Doctoral Thesis

Scheduling Algorithms for Data-Parallel Tasks on Multicore Architectures

September 2018

Doctoral Program in Advanced Electrical,

Electronic and Computer Systems

Graduate School of Science and Engineering

Ritsumeikan University

LIU Yang

Doctoral Thesis reviewed

by Ritsumeikan University

Scheduling Algorithms for Data-Parallel Tasks on Multicore Architectures

(データ並列タスクのマルチコアスケジューリングアルゴリズム)

September 2018

2018年9月

Graduate School of Science and Engineering,

Ritsumeikan University

立命館大学大学院理工学研究科

博士課程後期課程

LIU Yang

劉　陽

Supervisor: Professor TOMIYAMA Hiroyuki

研究指導教員：冨山　宏之　教授

Abstract

To meet the increasing demand for computational performance, the number of cores in embedded processors as well as general-purpose processors, has rapidly grown in recent years. How to fully utilize such processors with a high degree of parallelism has now become a more important issue than ever.

In general, the forms of parallelisms can be classified into task-parallelism and data-parallelism. The task-parallelism is achieved by the concurrent execution of different tasks on multiple cores in parallel, and the data-parallelism focuses on executing the same task with different input data sets on multiple cores. Many of existing task scheduling algorithms only consider the task-parallelism. In other words, each task is executed on a single core. However, most scientific and media applications often combine the two kinds of parallelism, which means, multiple data-parallel tasks are executed in a task-parallel fashion. This mixed-parallel approach significantly increases the scalability of parallelism. Many studies have shown that exploiting both task- and data-parallelisms often yields better performance than pure data- or task-parallelism. This paper addresses the task scheduling problem which takes into account both task- and data-parallelisms.

In this thesis, we provide an extensive survey on existing task scheduling algorithms. Since the scheduling problem is NP-hard, there are a large number of heuristics and meta-heuristics which aim to find near-optimal results in a practical time. List scheduling is one of the most popular heuristics for task scheduling problems, which assigns a particular priority to tasks, and schedules these tasks by the assigned priorities. In our thesis, we extend the traditional priority strategy to task scheduling for data-parallel tasks. We propose six list scheduling algorithms with different strategy of priority assignment. The experimental results demonstrate the effectiveness of the proposed algorithms against a commercial mathematical programming solver.

We also find that a specific static priority is hard to be effective against all applications. Next, we extend the simple list scheduling to use two static priorities switched during task scheduling. In our experiments, we compare the proposed algorithm with traditional list scheduling algorithms. The experimental results show that the proposed algorithm yields

shorter scheduling length, by 2% on average and up to 10%, than pure list scheduling with a single priority.

The advantages of list scheduling algorithms and their variants produce results in a very short time and are relatively simple to implement. However, their acquired scheduling results are often far from optimal ones. In recent years, many studies have turned to meta-heuristics to solve task scheduling problems. Meta-heuristics provide certain algorithmic frameworks to search the solution space and avoid local optimal results, which are effective ways to improve the quality of results. In this thesis, we present an introduction of several popular meta-heuristics for task scheduling. Furthermore, an efficient method based on a genetic algorithm (one kind of meta-heuristics) is proposed to solve the task scheduling problem which considers both task- and data-parallelism. Different from traditional genetic algorithms for task scheduling, we propose a novel representation for the chromosome of task scheduling and corresponding genetic operators, aiming to reduce the search space and improve the computing speed. In addition to the single-thread implementation, we parallelize our algorithm with OpenMP to speed up our algorithm. Our experiments show that the proposed genetic algorithm finds near-optimal schedules and outperforms the previously discussed list scheduling algorithms by 5% on average and up to 13%.

Although the heuristic and meta-heuristic algorithms produce sub-optimal scheduling lengths in a reasonable time, it is still desirable to obtain optimal scheduling lengths in some cases, for example, to evaluate heuristic algorithms. This thesis proposes an exact algorithm to find optimal results. The proposed algorithm is based on depth-first branch-and-bound search. We present four rules to prune non-optimal branches. The experiments show that our algorithm could find best schedules in a practical time. In our experiments with up to 100 tasks, the proposed algorithm successfully finds optimal schedules for 135 test cases out of 160 within 12 hours. Even in the case where optimal schedules are not found within 12 hours, the proposed algorithm finds better schedules than state-of-the-art heuristic algorithms.

As mentioned above, this thesis proposes broadly four algorithms for task scheduling with both task- and data-parallelisms. The four algorithms feature different characteristics on computational complexity and quality of results, and system designers can employ the one which best satisfies their requirements on computational complexity and quality of results.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

"Scheduling" is an ancient and important concept for our everyday life. It is used to set up daily personal agenda, organize staffs, allocate plant resources and plan aircraft landings. In computer science, we also need "scheduling" to allocate different tasks to limited computational resources, this process that significantly affects the performance of the overall computational system is usually called task scheduling.

Nowadays many-cores become more and more demanding because of their high performance. Even in the embedded system the number of cores also increased rapidly. How to design a more effective task scheduling algorithm to utilize all computational resources in such systems completely has become an increasingly critical topic.

The task scheduling algorithms are classified into two major categories. One is dynamic scheduling (also known as online scheduling) which is performed on-the-fly at the operation time of the systems. The other is static scheduling (also known as offline scheduling) which is done at the design time [6]. This thesis focuses on static scheduling. Because in many cases, embedded system design where characteristics of the tasks are known prior to the compiling stages, static scheduling is often preferred due to its low runtime overhead and high predictability.

In general, the static task scheduling problem tries to schedule a set of tasks and decides when and on which core each task is executed. The goal of scheduling algorithms is to minimize the overall scheduling length while the obtained scheduling result meets all flow dependencies and other constraints, if any.

Classic task scheduling problems for multi-core architectures assume that each task is executed on one of the cores. They try to perform as many tasks as possible in parallel on different cores. This execution scheme is called task-parallel execution. A large number of algorithms for task-parallel scheduling have been developed so far. Recent works include [7] [8] [9] and [10].

Meanwhile, data parallelism is another form of parallelism, which is achieved by executing the same task with different data on multiple cores simultaneously. 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 [35] [36]. This paper addresses task scheduling which takes into account both task parallelism and data parallelism. In other words, multiple data-parallel tasks can be executed simultaneously in a task-parallel fashion.

This thesis aims to provide a comprehensive study of task scheduling problem which schedules a set of data-parallel tasks on multiple cores. In this respect, the first part of our thesis presents the dominant existing algorithms for task scheduling problem. We also discuss the differences between them and compare their respective scope of application.

In the following chapters, we first show the definition of task scheduling problem with data-parallel tasks and some necessary notations used in this thesis. Then we introduce the proposed task scheduling algorithms.

The task scheduling is a kind of optimization problem. Because of the complex inter-task dependencies, the solution space of scheduling problem usually is discrete, highly non-convex and with a large number of discontinuities. This kind of problem is complicated to solve by simple local search algorithm, for example, the greedy algorithm or gradient descent. In general, to find the optimal solutions of task scheduling requires searching the overall solution space, which is very time-consuming. Therefore, heuristics or meta-heuristics are much practical ways to find good enough solutions (also be called as near-optimal solution) in an acceptable time.

We roughly divided existing algorithms for task scheduling problem into exact algorithms, heuristics and meta-heuristics in the following discussion. Exact algorithms guarantee to find the optimal solution. However, for most complex scheduling problems, searching the optimal solutions requires a long executing time. On the other hand, heuristics do not guarantee optimality, but can yield a near-optimal solution more quickly. Meta-heuristics also do not guarantee to reach an optimal solution. The main difference between heuristics and meta-heuristics is that heuristics are a problem-specific method. However, meta-heuristics is a framework that provides a set of guidelines or strategies to develop heuristic algorithms. The popular meta-heuristics including: the genetic algorithm (GA), the simulated annealing (SA) algorithm, and the ant colony optimization (ACO).

In chapter 4, we examined the existing state-of-the-art algorithms which based on list scheduling for task scheduling problem without data parallelism. List scheduling is one of the most popular heuristics for task scheduling problems, which assigns a particular priority to each task, and schedules tasks by the assigned priorities. We also extended the existing strategies of priority assignment, which makes list scheduling more adaptable to schedule data-parallel tasks. There are six algorithms with different priority strategy were proposed. In our experiments, the six algorithms, as well as an integer linear programming method are evaluated.

In chapter 5, we further improve the algorithms proposed in chapter 4. We find that the simple list scheduling algorithms tend to yield worse results, especially when the target system has more cores. To solve the problem, we propose a new list scheduling algorithm which employs two static priorities. The new algorithm switches two different priorities during the scheduling process. Thus, we call it as dual-mode algorithm. We use a set of experiments demonstrated that dual-mode algorithm yields better scheduling results than pure list scheduling algorithms.

Although the algorithms based on list scheduling are proposed in chapters 4 and 5 obtain good results in a short time. However, these kinds of deterministic algorithms generally find a priority strategy based on statistics or experiences. It often yields bad schedules for some specific problems. In chapter 6, we proposed a genetic algorithm for task scheduling problem. Different from heuristic approaches, genetic algorithm provides a set of mechanisms to search global solution space and escape from the local optimal solution more efficient. If the parameters are appropriately designed, a better solution is always available. Furthermore, we propose a novel chromosome representation for task scheduling problem. Our chromosome only encodes information about the order of task execution, does not represent which cores are assigned to which tasks. It greatly reduces the size of search space and improves the performance of the algorithm. Efficient genetic operators (i.e., selection, crossover and mutation) corresponding to the definition of chromosome also were presented. Although our genetic algorithm requires a much longer execution time than list scheduling algorithm proposed in chapters 4 and 5, since the computation is inherently parallel, we parallelize our algorithm with OpenMP to cover this problem. Our experiments show that the proposed genetic algorithm finds near-optimal schedules and outperforms the previously discussed list scheduling algorithms.

As we mentioned earlier, finding the optimal solution is very time-consuming especially for complex task scheduling problem. However, in some occasions, it is still desirable to obtain optimal schedules, for example, to evaluate heuristic algorithms. In chapter 6, we propose an exact task scheduling algorithm. The proposed algorithm is based on depth-first branch-and-bound search. In our experiments with up to 100 tasks, the proposed algorithm could successfully find optimal schedules for 135 test cases out of 160 within 12 hours. Even in the case where optimal schedules were not found within 12 hours, our experiments show this algorithm always found better schedules than heuristics and meta-heuristics proposed in chapters 4, 5 and 6.

Chapter 2.

Related Work

The task scheduling problem has been extensively studied for decades. From Table 1 we know that scheduling problems with tasks have arbitrary execution time and arbitrary precedence constraints are known to be NP-hard [1] [2] [3]. Pioneering researchers were proposed many heuristics and meta-heuristics, aim to find the approximate results in a reasonable amount of time. However, exact algorithms are still desirable to obtain optimal scheduling lengths in some case. This chapter presents a survey about the existing algorithms for solving task scheduling problem.

Table 1. The complexity of scheduling problems [55]

Number of Processors (m)	Task Processing Time Ti	Precedence Constraints	Complexity
Arbitrary	Equal	Tree	O(n)
2	Equal	Arbitrary	O(n^2)
Arbitrary	Equal	Arbitrary	NP-hard
Fixed (m>=2)	Ti=1or2 for all i	Arbitrary	NP-hard
Arbitrary	Arbitrary	Arbitrary	Strong NP-hard

2.1. Heuristic Algorithms

Since scheduling problem is known to be NP-hard, the most research efforts in this area are focused on heuristic algorithm to obtaining non-optimal results. The existing heuristics for task scheduling can be classified into three categories, list scheduling, cluster based scheduling and task duplication-based scheduling.

List Scheduling

The most important family of heuristics is based on list scheduling (e.g. [3] [4] [5] [11] [12] and [13]). The basic idea of list scheduling is to assign tasks with certain priority, and then allocate these tasks to free cores according to the priority repeatedly, until all the tasks are scheduled. List scheduling is generally accepted as an attractive approach since it pairs low complexity with good results. There are numerous variations of list scheduling using different ways to determine the priorities of each task, such as HLF (Highest Level First) [1]; LP (Longest Path) [1]; LPT(Longest Processing Time) [5]; and CP (Critical Path) [3].

Cluster-Based Scheduling

For scheduling with communication cost, the cluster-based scheduling schemes [14] are often employed. Cluster-based scheduling try to cluster of tasks based on certain criteria (e.g. tasks that need to communicate among themselves are grouped together to form a cluster). Tasks in same cluster are scheduled on the same processor. the methods can reduce inter-processor communication overhead significantly. However, if the available number of processors is less than the number of clusters, their solutions may not be very efficient.

Duplication-Based Scheduling

A general solution for the problem of cluster-based scheduling schemes is task duplication-based scheduling [15] [16] [17] [23]. Similar as cluster-based scheduling, task duplication is also tried to reduce the inter-processor communication overhead. The basic idea of task duplication is to duplicate the preceding task of the currently selected task onto the chosen processor. It aims to reduce or optimize the task's starting or finishing time.

The main weakness of duplication-based algorithms is their high complexity. The mainly target of duplication-based scheduling is to schedule a set of tasks to an unbounded number of computing machines. There are numerous variations of task duplication base algorithms using different strategies to determine which tasks to duplicate and on which cores to use for the tasks.

Cluster-based scheduling and duplication-based scheduling are considered be useful for systems with negligible communication cost between tasks which are allocated on different processors. (e.g. Distributed Computing).

2.2. Meta-Heuristic Algorithms

Meta-heuristic is a high-level problem-independent algorithmic framework that provides a set of guidelines or strategies to develop heuristic algorithms. Most meta-heuristic algorithms are designed based on some abstraction of nature. The most popular meta-heuristics including: genetic algorithm (GA), ant colony optimization (ACO), bee algorithms (BA), particle swarm optimization (PSO) and simulated annealing (SA).

Because of its effectiveness to solve combinational problems, meta-heuristics have gained massive popularity in the past years. In this section, we present a brief view of scheduling algorithms based on meta-heuristic algorithms.

Genetic Algorithms (GA)

The genetic algorithm was first invented by Holland [54]. This algorithm thinks of a set of candidate solutions for certain problem as biological population, in each step, the good individuals have a higher chance to pass its traits to next generations. And some traits of individuals may be mutated and altered. Better individuals will be found over successive generations.

In the past decades, Genetic algorithms have been widely used to evolve solutions for many task scheduling problems. Including [9] [18] [41] [42] [43] [44] [45]. How the definite the representation of individual for scheduling problem and corresponding genetic operators usually are the key issues for genetic task scheduling algorithm design.

Ant Colony Optimization (ACO)

Ant colony optimization (ACO) is another popular meta-heuristic algorithm for combinational problems, it was inspired by the behavior of real ants finding the food. When an real ant finds a food source, the ant will leave pheromones on the ways to its colony. Because other ants will attract to explore paths with more pheromones, as the time goes on, a better (shortest) path from the colony to the food source would normally be found.

Ant colony optimization is initially proposed in [38]. Although compared with the genetic algorithm, Ant colony optimization is relatively new. However, it has been successfully applied to the traveling salesman problem [51], the asymmetric traveling salesman problem [52], the quadratic assignment problem [53], and the transportation planning problems [49] [50].

There also are many works for task scheduling [39] [40] based on ant colony optimization. In general, scheduling algorithms based on ant colony optimization usually adopt the following steps:

1. Ants produce a scheduling resolution according to some information (generally known as pheromones) left by previous ants.
2. Evaluate scheduling solutions obtained by each ant. The better solutions will leave more pheromones.
3. Go to step 1.

2.3. Exact Algorithms

In this chapter, we introduce several exact methods for scheduling problem. Although most of task scheduling algorithms are based on heuristic algorithms to find sub-optimal results, however, on many occasions, it is still desirable to obtain optimal schedules, for example, to evaluate heuristic algorithms.

Branch-and-Bound Algorithms

The branch-and-bound algorithm (B&B) is the most frequently used exact method for task scheduling problem (i.e. [24] [25] [26] [27]). B&B explores all solution space which is represented as a branching tree. B&B prunes the branches if they have no candidates to furthermore improve the final results. DF/IHS (depth first/implicit heuristic search) [24] is one of task scheduling based on B&B. This method can reduce average computation time markedly by combining the branch-and-bound method with CP/MISF (critical path/most immediate successors first). J. Carlos [26] proposed a scheduling algorithm for heterogeneous system, which is multi-objective B&B algorithm based on Pareto dominance.

Integer Linear Programming

Linear programming tries to find a maximum or minimum solution while satisfying all given constraints. The integer linear programming (ILP) is a subset of linear programming. Its constraints and solution must be restricted to integers. When only some of the constraints are integer, the problem is called a mixed-integer linear program.

The exact results of scheduling problem can be acquired by ILP, because constraints in task scheduling problem in essence is a set of integer constraint functions.

Recently, Venugopalan [46] has proposed ILP based approach which aims to find exact results for task scheduling problem with communication delays. The contribution of this work is to use problem specific knowledge to eliminate the bi-linear forms arising out of communication delays, and to run all variable indices in the proposed MILP formulation independent of the number of processors which reduces the complexity significantly.

A* Search Algorithms

The A* search algorithm [47] is often used for finding the shortest path between two points for robot navigation or game. In [48], a new algorithm was reported to solve the problem of task scheduling by A* searching algorithm. A* task scheduling algorithm starts from a state where all tasks are not scheduled. At each iteration, A* choose one or more states with minimum cost to produce new states.

Usually, the cost is defined as:

$$cost(s) = g(s) + h(s) \tag{1}$$

Where s is a partial scheduling state, $g(s)$ is its scheduling length. $h(n)$ is a heuristic which estimates the scheduling length from the current state to the final state. An admissible $h(n)$ will significantly reduce the search space.

2.4. Algorithm for Data-Parallel Task

Unfortunately, the majority of works on task scheduling (The above mentioned algorithms) only consider the task parallelism. Many studies [35] [36] [37] have shown that, for a large class of large computational applications, exploiting both task and data parallelism yields better speedups compared to either pure task parallelism and pure data parallelism.

There are several research efforts for task scheduling problem with data-parallel tasks. Recent works include [28], [29] and [30]. In [28], Yang and Ha proposed a mapping and scheduling technique which is based on integer linear programming (ILP) formulation, and extended the technique in [29] by assuming that several pipeline stages can share a multi-core processor. Vydyanathan also proposed an algorithm for data-parallel tasks in [30], which reduces the overall scheduling length by a locality conscious scheduling strategy. There is a common assumption in [28], [29] [30] that the degree of data parallelism for each task is flexible, and increasing the number of cores assigned to a task decreases the execution time of the task. The exact execution time of the task on different cores is known before the task schedule starts. However, to acquire the execution times of all tasks on different degree of data parallelism may be very tedious and time-consuming. This thesis assumes that tasks have a fixed degree of data parallelism which was decided by human programmers.

Chapter 3.

The Problem Definition

The problem of task scheduling can be described as scheduling and mapping a set of tasks which belongs to a task graph onto a multicore system, with a goal of minimum scheduling length under constraints on inter-task dependency. This section presents the application model for the task scheduling problem with data-parallel task. Essential, the task scheduling is a kind of mathematical optimization. We also discuss that how to use ILP (integer linear programming) to define and solve this problem in this section.

3.1. Task Graph

The task graph (also be called acyclic directed graph) is an intuitive representation of parallel applications. It consists of a set of nodes and directed edges, in which the nodes represent tasks and edges represents the flow dependencies between different tasks. An example of task graph is shown in Figure 1.

Task graph has two dummy tasks S and E. The tasks S with no parent is the entry point, and the tasks E with no child is the exit point of an application. The common tasks have its own execution time and degree of data parallelism which are marked respectively behind the correspondent tasks. For example, the task 1 must be executed on 4 cores concurrently, and take 10 time units to finish its work.

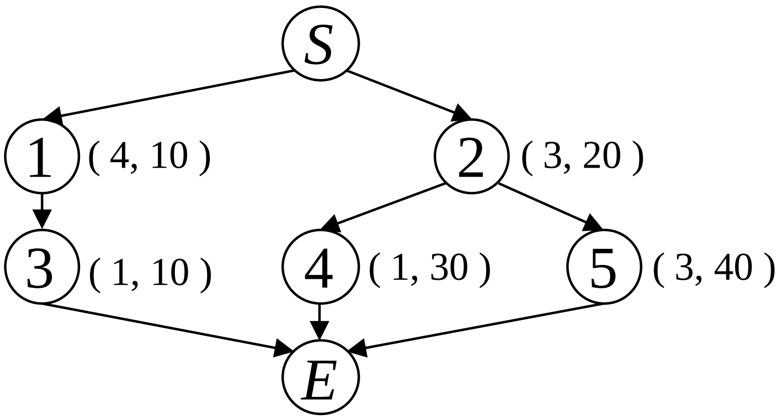


Figure 1. A task graph

In this thesis, we assume that individual tasks are implemented by human programmers using parallel programming techniques. The degree of parallelism for each task does not change during the task scheduling process. How to choose an appropriate degree of parallelism, and how to determine the execution time of the task on this degree of parallelism are beyond the scope of this article.

3.2. System Model

In this thesis, the target system is assumed to be a set of cores which is fully connected by high-speed bus. The architecture of these cores is same (homogeneous system). We also assume that:

- Task has the same execution time on arbitrary cores.
- The task which is being executed cannot interrupt or preempt by another task.
- The communication time between tasks is ignorable.

Time = 0	10	20	30	40	50	60	70	80
Core 0	T1	T2	T2	T5	T5	T5	T5	
Core 1	T1	T2	T2	T5	T5	T5	T5	
Core 2	T1	T2	T2	T5	T5	T5	T5	
Core 3	T1	T3		T4	T4	T4		

(a) A schedule for Figure 1 which with scheduling length equal to 70 time units

Time = 0	10	20	30	40	50	60	70	80
Core 0	T2	T2	T5	T5	T5	T5	T1	T3
Core 1	T2	T2	T5	T5	T5	T5	T1	
Core 2	T2	T2	T5	T5	T5	T5	T1	
Core 3			T4	T4	T4		T1	

(b) A schedule for Figure 1 which with scheduling length equal to 80 time units

Time = 0	10	20	30	40	50	60	70	80
Core 0	T1	T2	T2	T5	T5	T5	T5	
Core 1	T1	T3		T5	T5	T5	T5	
Core 2	T1	T2	T2	T5	T5	T5	T5	
Core 3	T1	T2	T2	T4	T4	T4		

(c) Another schedule for Figure 1 which with scheduling length equal to 70 time units

Figure 2. Some scheduling results for task graph in Figure 1

Figure 2 shows several scheduling resolutions for task graph in Figure 1. The target system is described in above with four cores. Obviously, the same task graph can be scheduled as totally different ways and have different scheduling length. The scheduling problem aims to find the minimal overall scheduling length, while meet all constraints which are described in this task graph.

3.3. ILP Formulations

The task scheduling is essentially a kind of mathematical optimization problem. It can be formulated by an integer linear programming (ILP) [22]. In this section, we use the following ILP formulation to describe the task scheduling problem mentioned in 3.1.

Before we go into the details of our ILP formulation, we first present the notation used in (2) ~ (6). The $time_i$ is the execution time of $task_i$, and par_i denotes the degree of data parallelism of $task_i$. If there is a data dependency between $tasks_{i1}$ and $tasks_{i2}$, the $flow_{i1,\,i2}$ equal to 1. Otherwise, the $flow_{i1,\,i2}$ equal to 0. $start_i$ and $finish_i$ denote the start time and finish time of $task_i$, respectively. $map_{i,j}$ denotes mapping information of $task_i$, that is, if the $map_{i,j}$ is equal to 1 means the $task_i$ is assigned to $core_j$.

Minimize:

$$\text{Max}(finsih_i) \tag{2}$$

Subject to:

$$\forall i \quad \textstyle\sum_{\text{j}} map_{i,j} = par_i \tag{3}$$

$$\forall i \quad finish_i = start_i + time_i \tag{4}$$

$$\begin{aligned} \forall i1, i2, j \qquad & map_{i1,j} + map_{i2,j} \le 1 \\ & \vee\ finish_{i1} \le start_{i2} \\ & \vee\ finish_{i2} \le start_{i1} \end{aligned} \tag{5}$$

$$\forall i1, i2 \quad flow_{i1,i2} = 1 \rightarrow finish_{i1} \le start_{i2} \tag{6}$$

The objective function (2) indicates that the goal of task scheduling is minimizing the overall scheduling results. The Formula (3) expresses that $task_i$ must be mapped onto par_i cores. The Formula (4) guarantee $task_i$ takes $time_i$ to complete. Formula (5) ensures that every task cannot run on same cores at the same time. Formula (6) describes that if $task_{i2}$ depend on $task_{i1}$, the $task_{i2}$ must be executed until the $task_{i1}$ finished.

Then, the task scheduling problem can be defined as follows: Given $time_i$, par_i and $flow_{i1,i2}$, try to find appropriate $start_i$, $finish_i$ and $map_{i,j}$ to minimize the overall scheduling length. Although at least in theory, solving the above ILP formulas can acquire the exact solution of task scheduling problem. However, it is very time-consuming may not practical for large task graphs.

Chapter 4.

List Scheduling Algorithms

List scheduling is a most important heuristic for task scheduling. In this section, we propose six heuristic algorithms for scheduling problem. All of the six algorithms are based on list scheduling, but their priority assignment strategies are different.

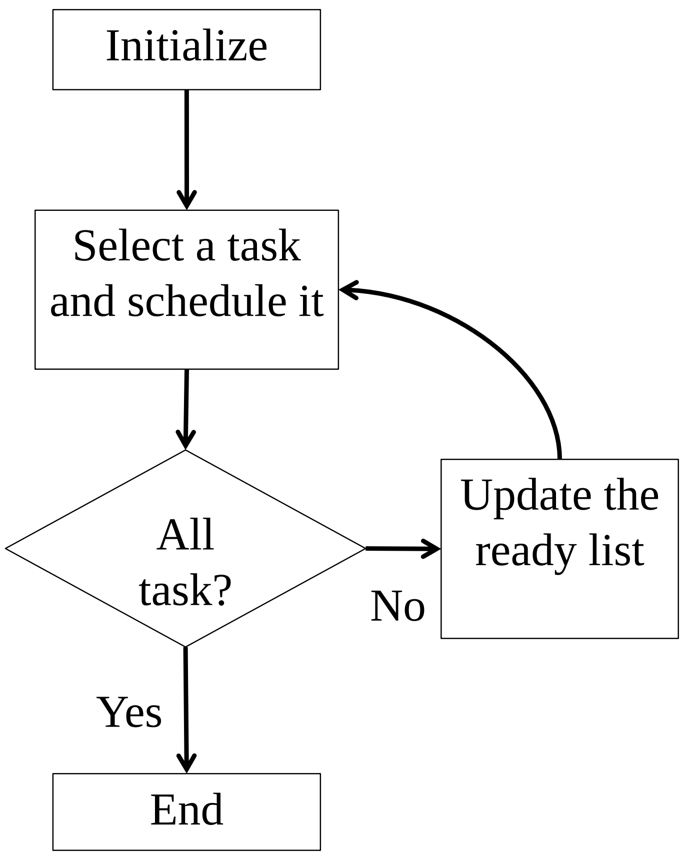


Figure 3. Flowchart for list scheduling algorithm

Many heuristics for task scheduling are based on list scheduling. The basic idea of list scheduling is to make a *Readylist*. The *Readylist* contains a sequence of tasks which can be scheduled immediately.

List scheduling repeatedly executes the following three steps until a valid schedule is obtained:

(1). Update the *Readylist*.

(2). select a task from *Readylist*.

(3). Assign this task to suitable processors.

The above steps finished until all tasks are scheduled. In step (2), the task with the highest priority will be selected first. There are a number of list-based algorithms employ different ways to determine the priority of tasks. How to define the priority of tasks is the most important issue for the design of list scheduling. The list scheduling described in above also summarized in Figure 3.

4.1. A Motivating Example

CP/MISF (critical path/most immediate successor first) is a list-based scheduling algorithm. It is designed to handle task scheduling without data parallelism by Kasahara and Narita in [24]. Although this algorithm was developed more than three decades ago, because of the high quality of results as well as the low computational complexity, CP/MISF still be considered as one of the best heuristics in practical use.

The CP/MISF algorithm, as implied by its name, uses two different factors to determine the priority of tasks: the critical path length and the number of immediate successors. The critical path length of a task is the longest distance from this task to the end point of the overall task graph. Figure 4 is the same task graph as Figure 1. In Figure 4 two numbers are written in red behind each task denote its critical path length and the number of immediate successors respectively. For example, the critical path length of task 2 is 60, by

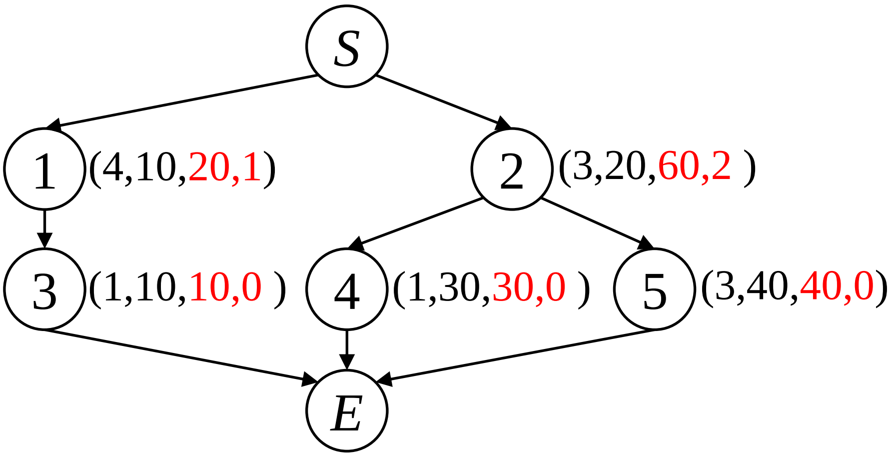


Figure 4. A task graph with critical path and most immediate successor number

passing through the task 2 and task 5. In the CP/MISF algorithm, the priority of tasks is defined according to the following two rules:

- If the critical path of task i is longer than that of task j, task i has a higher priority than task j.
- If two tasks have same critical path length, the task with more immediate successors has higher propriety than other.

Figure 6 illustrates the scheduling result of task graph in Figure 4 acquired by the CP/MISF algorithm. In the beginning, The tasks 1 and task 2 with no parent tasks except the task S are ready to schedule. The CP/MISF algorithm schedules task 2 first, because of its longer critical path. Next, tasks 4 and 5 become schedulable, task 5 is selected because it has the longest critical path between task 1, task 4 and task 5. Then, followed by tasks 2 and 5. The task 4, task 1 and task 3 are scheduled. We can see from Figure 6 the overall scheduling length acquired by the CP/MISF algorithm is 80 time units.

The CP/MISF algorithm is very efficient for task scheduling only considering task parallelism. However, the CP/MISF algorithm is not worked well for tasks with data parallelism. Actually, The scheduling result in Figure 6 is not optimal. Then, we try to schedule the same task graph in a better way. In Figure 5 we give the tasks which with larger data parallelism higher priority. According to this priority strategy, task 1 is scheduled first. Next, task 2 and task 3 can be scheduled at the same time with a task parallel fashion. This way works better than CP/MISF algorithm and shortens the overall scheduling length by 10 time unit. Of course, this policy is not always effective, but this example shows that the degree of data parallelism is an important factor for scheduling data-parallel tasks, and should be considered in the priority strategy.

Time=0	10	20	30	40	50	60	70	80
Core 0	T2	T2	T5	T5	T5	T5	T1	T3
Core 1	T2	T2	T5	T5	T5	T5	T1	
Core 2	T2	T2	T5	T5	T5	T5	T1	
Core 3			T4	T4	T4		T1	

Figure 6. Schedule obtained by the CP/MISF algorithm

Time=0	10	20	30	40	50	60	70	80
Core 0	T1	T2	T2	T5	T5	T5	T5	
Core 1	T1	T2	T2	T5	T5	T5	T5	
Core 2	T1	T2	T2	T5	T5	T5	T5	
Core 3	T1	T3		T4	T4	T4		

Figure 5.Schedule which takes into account the degree of data parallelism

4.2. The Proposed Priorities

According to the example in 4.1, we propose a set of algorithms based on list scheduling, the priority strategies of which take into account three factors, including the degree of data parallelism, the length of critical path and the number of immediate successors. For the convenience in writing, we use following notations:

- P: The degree of data parallelism
- C: The length of critical path
- S: The number of immediate successors

In this chapter, the first proposed algorithm adopts the following priority based on the three factors.

1. If task i has a larger data parallelism than task j, task i has a higher priority than task j.
2. In case tasks i and j has the same degree of data parallelism, if the critical path of task i is longer than that of task j, task i has a higher priority than task j.
3. In case tasks i and j has the same degree of parallelism and the same length of critical paths, if task i has more immediate successors than task j, task i has a higher priority than task j.

We named the above priority strategy as PCS, since the three factors (P, C and S) are prioritized in the order of P-C-S. In the PCS algorithm, each task has a priority value which is called $PriorityPCS_i$. A larger $PriorityPCS_i$ value indicates the task has a higher priority. We formal define the $PriorityPCS_i$ as follows:

$$PriorityPCS_i = \mathrm{U}^2 \cdot P_i + \mathrm{U} \cdot C_i + S_i \tag{7}$$

The P_i, C_i, and S_i in formula (7) are the values of P, C and S for task i, respectively. U is a constant value greater than any of P_i, C_i, and S_i for any i. The next five algorithms CPS, CSP, SCP, PSC and SPC are defined in the similar way, but with different priorities using the three factors. The task priorities in the five algorithms are defined as follows:

$$PriorityCPS_i = \mathrm{U}^2 \cdot C_i + \mathrm{U} \cdot P_i + S_i \tag{8}$$

$$PriorityCSP_i = \mathrm{U}^2 \cdot C_i + \mathrm{U} \cdot S_i + P_i \tag{9}$$

$$PrioritySCP_i = \mathrm{U}^2 \cdot S_i + \mathrm{U} \cdot C_i + P_i \tag{10}$$

$$PriorityPSC_i = \mathrm{U}^2 \cdot P_i + \mathrm{U} \cdot S_i + C_i \tag{11}$$

$$PrioritySPC_i = \mathrm{U}^2 \cdot S_i + \mathrm{U} \cdot P_i + C_i \tag{12}$$

An important common feature must note that all of the six algorithms use static priorities, which means the priorities are determined before scheduling, and they do not change while the scheduling process.

4.3. Experiments

In this section, we adopted the Standard Task Graph (STG) Set [31] [55] which was developed at Waseda University to evaluate the effectiveness of the six algorithms. Since task graphs in STG do not indicate the degree of data parallelism of each task, we randomly assigned it to all of the tasks. The number of cores was changed from two to sixteen.

We compare the six algorithms proposed in this chapter with integer linear programming (ILP) technique (defined in chapter 2). In our experiments, we use IBM ILOG CPLEX 12.5 to solve the ILP problems. Actually, In the majority of cases, CPLEX cannot find the exact results in a reasonable time. Therefore, we limited the CPU time of CPLEX in 60 minutes, and the best solutions found in the limited time were compared with the proposed algorithms. We conducted our experiments on dual Xeon processors (E5-2650, 2.00Hz) with 128GB memory.

4.3.1. Results for Random Task Graphs

First, we evaluated our algorithms using 20 random task graphs which with 50 tasks. Table 2-a, b, c and d show the scheduling lengths obtained by the proposed six algorithms as well as the ILP method. We use X to mark in the ILP column if the ILP method failed to find any feasible solution within 60 minutes in CPU time. For ease of comparison, in each test the best scheduling result is highlighted in red.

We can see from Table 2, in many benchmarks, the ILP method cannot find a solution within the limited CPU times. Even the feasible schedules were found by the ILP method, usually are much longer than the scheduling results of other algorithms.

Figure 7 shows the average scheduling lengths of task graphs with 50 tasks obtained by the six algorithms proposed in this paper. We normalized the scheduling lengths to the PCS algorithm. This figure shows that, on average, PCS algorithm find better solutions than other methods.

Next, we evaluated our algorithms using 20 random task graphs which with 100 tasks. Similar to Table 2, Table 3 shows the scheduling lengths which obtained by seven methods, but task graphs with 100 tasks. And Figure 8 shows the average scheduling lengths which were normalized to results of PCS. The Table 3 and Figure 8 demonstrated that PCS algorithm finds better solutions than other methods on average again.

From Table 2 and Table 3, we also note that, although PCS worked nice on average, in many cases, it still found longer scheduling results than other algorithms. We consider that pure list scheduling based algorithms are hard to work well for all task graphs.

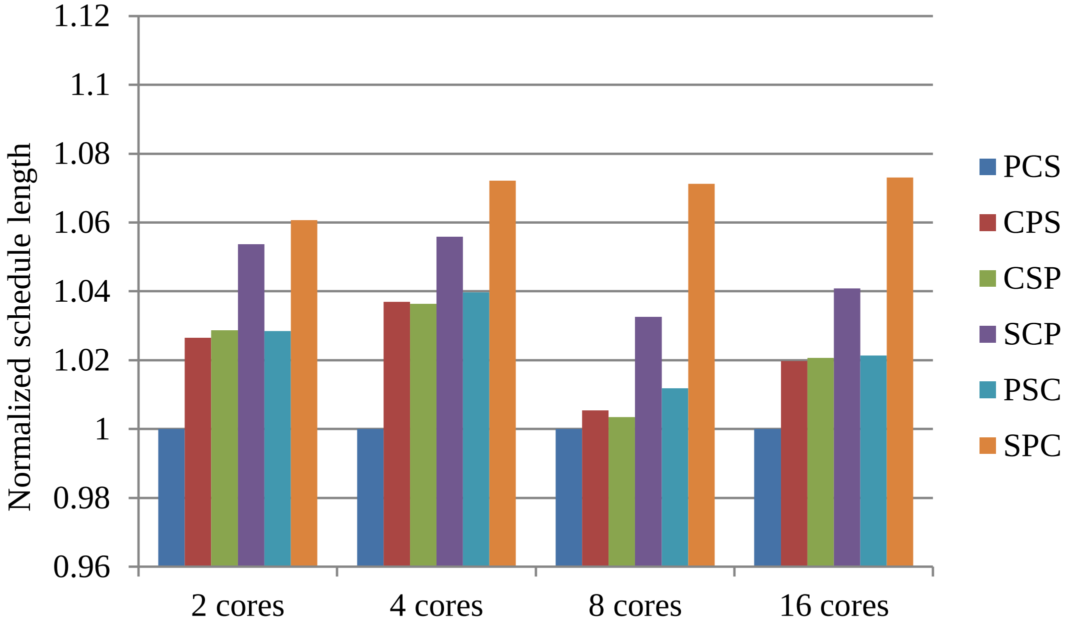


Figure 7. Averages of normalized schedule lengths for task graphs with 50 tasks.

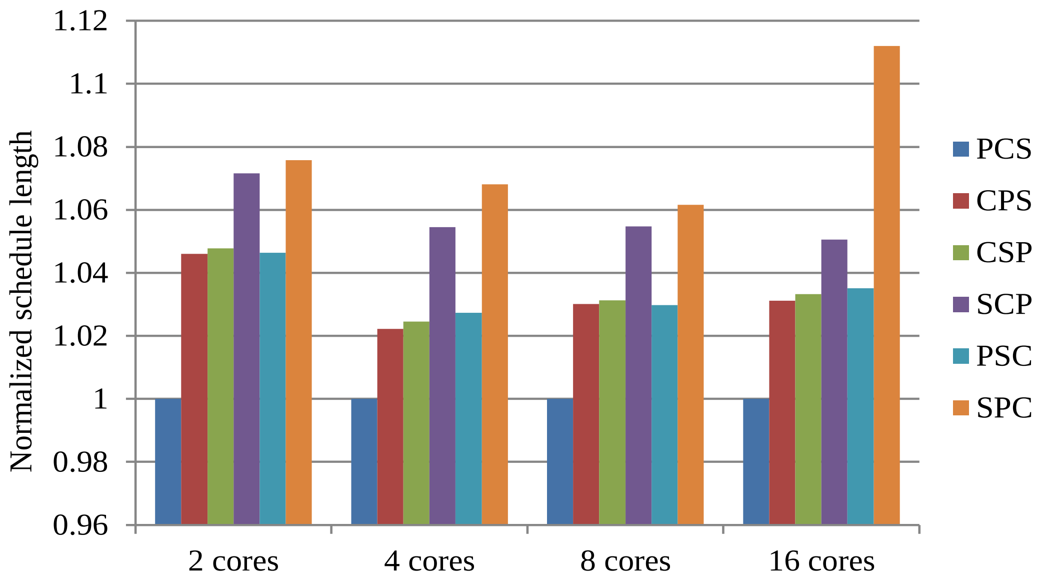


Figure 8. Averages of normalized schedule lengths for task graphs with 100 tasks.

Table 2-a. Scheduling lengths for task graphs with 50 tasks on 2 cores

Task graph IDs	PCS	CPS	CSP	SCP	PSC	SPC	ILP
rand0000	203	200	200	210	200	212	204
rand0001	232	233	233	249	233	251	232
rand0002	188	192	192	199	192	199	197
rand0003	224	224	224	230	225	228	241
rand0004	177	181	181	189	181	191	180
rand0005	495	496	496	520	496	531	504
rand0006	351	363	363	372	363	375	356
rand0007	384	387	387	394	391	400	430
rand0008	434	456	456	447	456	464	460
rand0009	386	397	397	412	397	410	398
rand0010	153	162	162	156	163	159	165
rand0011	205	213	213	208	213	210	198
rand0012	208	211	211	213	211	213	200
rand0013	238	252	252	282	252	287	248
rand0014	195	197	197	196	197	201	208
rand0015	425	448	448	452	448	444	427
rand0016	374	390	390	398	395	408	389
rand0017	439	448	467	492	456	491	471
rand0018	428	443	443	438	443	430	429
rand0019	393	409	409	416	403	407	404

Table 2-b. Scheduling lengths for task graphs with 50 tasks on 4 cores

Task graph IDs	PCS	CPS	CSP	SCP	PSC	SPC	ILP
rand0000	168	178	178	175	180	178	X
rand0001	220	214	214	229	214	232	X
rand0002	173	173	173	183	174	186	197
rand0003	194	202	202	211	202	201	X
rand0004	167	168	168	171	170	186	X
rand0005	439	443	438	448	449	448	464
rand0006	275	293	293	294	293	305	X
rand0007	357	348	348	358	349	367	X
rand0008	409	415	415	424	415	412	456
rand0009	327	373	373	368	373	363	X
rand0010	131	139	139	134	140	134	X
rand0011	181	192	192	177	192	177	191
rand0012	197	195	195	201	195	212	X
rand0013	186	214	214	239	214	254	X
rand0014	171	181	181	175	181	175	X
rand0015	376	377	377	383	373	386	382
rand0016	318	330	330	342	331	356	360
rand0017	377	396	396	414	396	414	X
rand0018	403	390	390	408	392	414	401
rand0019	342	368	368	368	369	373	X

Table 2-c. Scheduling lengths for task graphs with 50 tasks on 8 cores

Task graph IDs	PCS	CPS	CSP	SCP	PSC	SPC	ILP
rand0000	149	152	152	151	160	160	X
rand0001	203	210	210	197	210	212	X
rand0002	161	153	153	156	153	164	X
rand0003	175	180	180	183	180	189	X
rand0004	150	155	155	160	154	172	X
rand0005	432	402	402	438	402	439	X
rand0006	259	260	252	269	262	281	X
rand0007	336	325	325	324	324	338	X
rand0008	366	362	362	367	362	377	X
rand0009	323	324	324	338	324	349	X
rand0010	127	134	134	128	134	132	193
rand0011	180	173	173	178	173	195	X
rand0012	183	180	180	183	180	183	X
rand0013	171	170	169	215	170	233	X
rand0014	166	169	169	164	169	164	X
rand0015	304	314	314	307	314	307	X
rand0016	269	289	289	319	302	323	X
rand0017	306	305	305	326	310	342	X
rand0018	358	357	357	354	362	363	403
rand0019	361	373	373	371	373	371	X

Table 2-d. Scheduling lengths for task graphs with 50 tasks on 16 cores

Task graph IDs	PCS	CPS	CSP	SCP	PSC	SPC	ILP
rand0000	156	149	152	148	152	160	211
rand0001	195	204	205	213	204	213	227
rand0002	150	143	143	149	143	146	199
rand0003	169	174	174	171	174	184	219
rand0004	158	159	159	157	159	167	188
rand0005	406	399	399	413	399	451	463
rand0006	268	261	261	263	261	282	360
rand0007	301	283	283	298	283	288	431
rand0008	360	347	347	370	347	369	438
rand0009	289	303	303	309	303	286	382
rand0010	126	133	133	129	133	133	168
rand0011	135	155	155	172	155	186	175
rand0012	174	182	183	183	182	197	213
rand0013	154	174	174	199	174	201	243
rand0014	160	160	158	162	160	166	191
rand0015	325	336	336	331	336	343	445
rand0016	286	301	301	291	304	286	387
rand0017	333	337	337	319	338	336	481
rand0018	342	350	350	372	350	382	415
rand0019	334	332	332	319	332	334	401

Table 3-a. Scheduling lengths for task graphs with 100 tasks on 2 cores

Task graph IDs	PCS	CPS	CSP	SCP	PSC	SPC	ILP
rand0000	431	447	447	463	445	466	X
rand0001	401	411	411	416	411	418	X
rand0002	459	480	486	508	480	512	X
rand0003	406	419	419	427	416	431	501
rand0004	393	417	417	408	422	416	459
rand0005	814	833	833	868	842	873	X
rand0006	868	886	882	916	886	899	965
rand0007	861	872	872	888	869	929	997
rand0008	796	818	818	824	818	806	X
rand0009	947	963	963	958	963	974	X
rand0010	464	485	485	488	485	490	532
rand0011	445	464	466	456	466	455	X
rand0012	469	484	484	522	484	528	551
rand0013	480	502	502	513	502	513	X
rand0014	391	417	417	422	415	418	X
rand0015	781	792	792	873	792	866	X
rand0016	764	862	860	868	857	863	X
rand0017	860	920	922	936	922	927	X
rand0018	724	777	792	794	779	828	X
rand0019	749	825	825	860	825	844	856

Table 3-b. Scheduling lengths for task graphs with 100 tasks on 4 cores

Task graph IDs	PCS	CPS	CSP	SCP	PSC	SPC	ILP
rand0000	388	396	396	399	392	406	X
rand0001	348	361	366	381	362	380	X
rand0002	413	429	429	448	429	466	X
rand0003	341	363	363	375	365	375	X
rand0004	454	369	376	387	382	396	X
rand0005	704	707	698	739	698	753	X
rand0006	785	778	778	790	782	813	X
rand0007	760	773	773	797	773	806	X
rand0008	701	726	726	750	726	739	X
rand0009	783	806	810	852	810	843	X
rand0010	385	402	402	405	402	417	X
rand0011	394	406	410	400	416	400	X
rand0012	432	450	450	477	450	490	X
rand0013	404	435	440	426	437	431	X
rand0014	354	353	357	370	359	369	X
rand0015	706	695	694	721	697	734	X
rand0016	667	700	700	722	700	730	X
rand0017	746	796	798	828	798	818	X
rand0018	628	669	662	651	669	686	X
rand0019	700	725	726	802	743	814	X

Table 3-c. Scheduling lengths for task graphs with 100 tasks on 8 cores

Task graph IDs	PCS	CPS	CSP	SCP	PSC	SPC	ILP
rand0000	356	355	355	357	361	368	X
rand0001	326	345	347	350	346	366	X
rand0002	380	380	382	387	382	387	X
rand0003	338	354	354	371	353	365	X
rand0004	340	355	344	342	350	360	X
rand0005	713	701	701	764	701	759	X
rand0006	712	732	730	730	730	731	X
rand0007	675	728	728	712	728	709	X
rand0008	637	669	669	671	669	674	X
rand0009	785	754	754	748	754	774	X
rand0010	338	354	375	358	356	358	X
rand0011	353	382	384	389	381	398	X
rand0012	431	435	435	441	435	443	X
rand0013	382	402	405	395	402	406	X
rand0014	327	344	343	342	343	347	X
rand0015	697	671	658	714	658	692	X
rand0016	625	649	649	705	657	721	X
rand0017	730	770	770	816	770	783	X
rand0018	657	668	668	673	668	679	X
rand0019	679	705	701	801	701	775	X

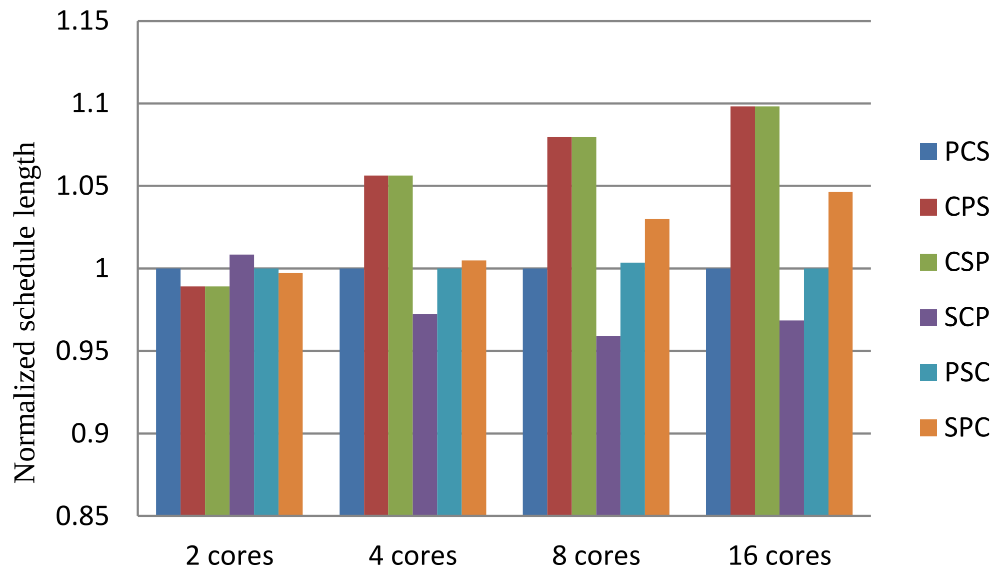
Table 3-d. Scheduling lengths for task graphs with 100 tasks on 16 cores

Task graph IDs	PCS	CPS	CSP	SCP	PSC	SPC	ILP
rand0000	335	351	346	354	358	368	494
rand0001	307	327	327	317	326	366	483
rand0002	365	352	353	381	353	387	501
rand0003	314	329	327	336	331	365	449
rand0004	317	314	320	324	320	360	489
rand0005	668	690	690	699	690	759	920
rand0006	687	705	701	719	705	731	789
rand0007	665	694	694	690	696	709	945
rand0008	607	618	618	620	618	674	900
rand0009	728	742	742	786	742	774	944
rand0010	362	370	372	362	361	358	501
rand0011	336	342	342	344	351	398	480
rand0012	410	394	397	437	414	443	541
rand0013	375	395	399	431	394	406	556
rand0014	313	337	338	325	338	347	473
rand0015	606	625	625	613	597	692	978
rand0016	648	670	670	671	670	721	876
rand0017	677	727	727	750	727	783	1024
rand0018	591	644	652	615	652	679	832
rand0019	676	682	686	731	690	775	796

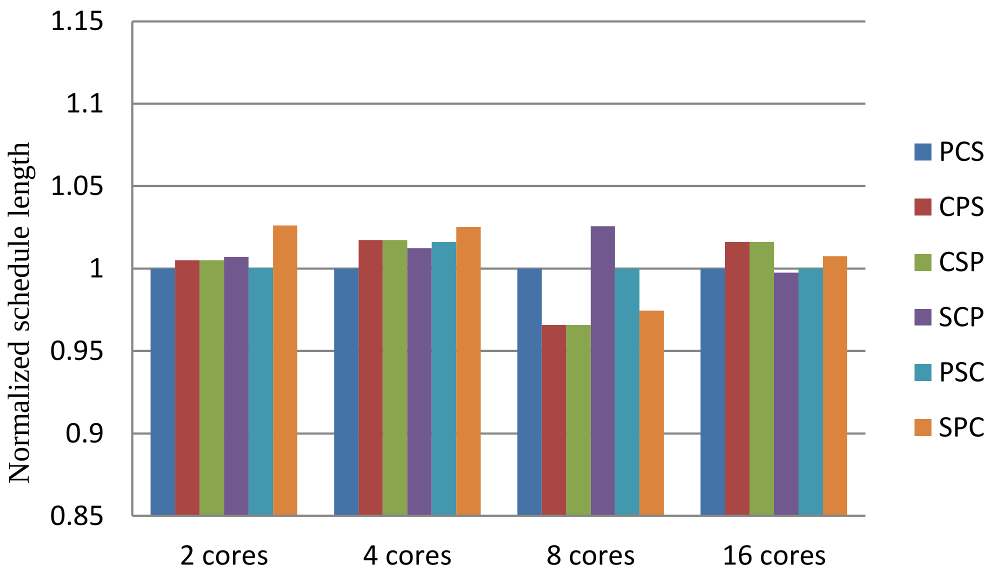
4.3.2. Results for Realistic Task Graphs

The next experiments, we used three task graphs developed from the realistic applications, i.e., (a) a part of fpppp from in the SPEC benchmarks, (b) robot control and (c) sparse matrix solver [55]. The task graphs are generated by the OSCAR Parallelizing Compiler, [32], [33] and [34]. The task graphs of fpppp, robot and sparse contain 334 tasks, 88 tasks, and 96 tasks, respectively. Table 4 shows the scheduling lengths obtained by six proposed algorithms. We can see in general, PCS yielded better scheduling results more efficiently than other algorithms.

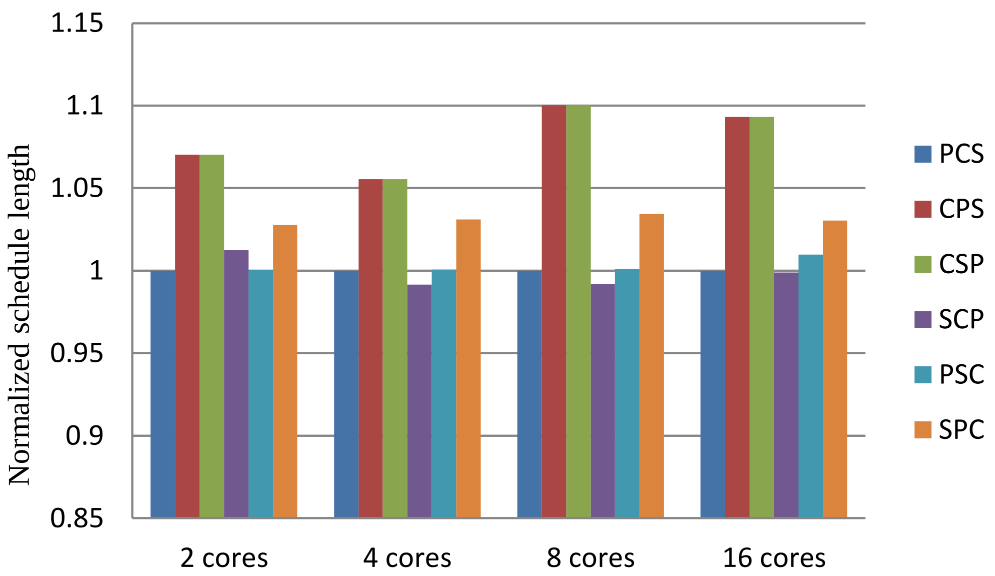
Figure 9-(a), (b) and (c) show the normalized the scheduling lengths of three realistic task graphs, respectively. We found although PCS algorithm obtains good schedules in general. However, for robot on eight cores and sparse on two cores, some others algorithms perform better than PCS.



(a) robot



(b) sparse



(c) fpppp

Figure 9. Normalized schedule lengths for realistic task graphs.

Table 4. Scheduling lengths for realistic task graphs

Robot				
	2 cores	4 cores	8 cores	16 cores
PCS	1951	1739	1731	1615
CPS	1961	1769	1672	1641
CSP	1961	1769	1672	1641
SCP	1975	1791	1715	1637
PSC	1952	1767	1731	1615
SPC	2002	1783	1687	1627
Sparse				
	2 cores	4 cores	8 cores	16 cores
PCS	1458	1242	1132	1038
CPS	1442	1312	1222	1140
CSP	1442	1312	1222	1140
SCP	1454	1276	1172	1104
PSC	1458	1242	1136	1038
SPC	1454	1248	1166	1086
Fpppp				
	2 cores	4 cores	8 cores	16 cores
PCS	5361	4881	4533	4487
CPS	5738	5152	4987	4905
CSP	5738	5152	4987	4905
SCP	5809	5108	4946	4899
PSC	5363	4884	4538	4531
SPC	5509	5032	4689	4623

Chapter 5.

Dual-Mode Algorithm

In chapter 4, the experimental results show that the PCS algorithm yields the best scheduling results on average. However, due to the static priority assigned by list scheduling algorithm, it is difficult to produce a good scheduling result for all task graphs.

In this chapter, we proposed a new algorithm for task scheduling with data-parallel tasks. This algorithm uses two static priorities and applies different priority strategies during the task scheduling. Generally, this kind of flexible priority strategy helps the new algorithm to achieve better scheduling length on average. In our experiments, the experimental results show that the proposed algorithm yields the better scheduling results than PCS.

5.1. The Problem of Pure List-Scheduling

In chapter 4, we proposed six list scheduling algorithms to solve task scheduling problem with data parallelism. The list scheduling algorithms use a ready list to contain tasks of whose parent tasks are scheduled. The tasks in ready list will be scheduled according to certain priority. After a task is scheduled, the ready list will be updated. The approach continues until the ready list is empty.

Among the six proposed algorithms, the PCS algorithm yields the shortest scheduling lengths on average. The effectiveness of list scheduling depends the most on how to define the priority.

Generally, the PCS algorithm has the following advantages: the PCS algorithm schedules tasks which occupy more free cores first. In the fixed degree of data parallelism system, a task with higher data parallelism have fewer chances do task parallelism with another task. If a task cannot do task parallelism, performing it at any time will not affect the overall scheduling length. However, scheduling such task earlier may activate some sub-tasks, and subsequent scheduling process has more candidate tasks to utilize multi-cores fully. Due to the above advantages, the PCS algorithm yields the good scheduling results in many cases.

Many studies (for example [1] [2] [24]) have shown that task with longer critical path is scheduled later may make the overall scheduling result become longer. The PCS algorithm does not always yield good schedules. Especially in system have more cores. We investigated the reason carefully and found that, although critical paths have been concerned in the PCS algorithm, some small-scale parallel tasks with a longer critical path still have lower priority. Such tasks will be executed late and make the overall scheduling length longer.

According to the above theory, scheduling task with longer critical path obtains good results in some case. Table 2 and Table 3 shows that CPS or CSP is better than PCS in some cases. To solve this problem, we design an new algorithm which has the advantage of PCS and CSP/CPS simultaneously, named dual-mode scheduling algorithm.

5.2. A Motivating Example

In this section, we use a simple example to show that the PCS algorithm failed to yield good scheduling results for some task graph. Figure 10 shows a task graph, and the critical path length and the number of immediate successors of this task graph are written in red. Figure 11 is the scheduling results of this task graph obtained by PCS algorithm. We assume that the processor has four available cores

In the beginning, the tasks 1 and 2 with no parent tasks (except task S) are ready to schedule. The PCS algorithm schedules task 2 first, because task 2 has a higher degree of data parallelism. In next step, the task 1 and tasks 3 is schedulable. The task 3 is selected according to the PCS priority. Then, tasks 3, Task 5, task 4 are subsequently scheduled. We can see from Figure 11 the total scheduling length is 80 time units.

We notice that PCS priority tend to schedule a task which can fully utilize the degree of parallelism of CPU. However, it does not consider whether there are enough tasks to run in parallel with the currently scheduled task. Actually, in Figure 11, scheduling task 1 first will lose the opportunity to execute task 1 with the other tasks.

Next, we try to schedule task 2 first. And then, task 1 and task 3 can be executed concurrently. Task 1 and task 4 can be executed at the same time. This way shortens the overall scheduling length to 60 time units. And the scheduling result is shown in Figure 12.

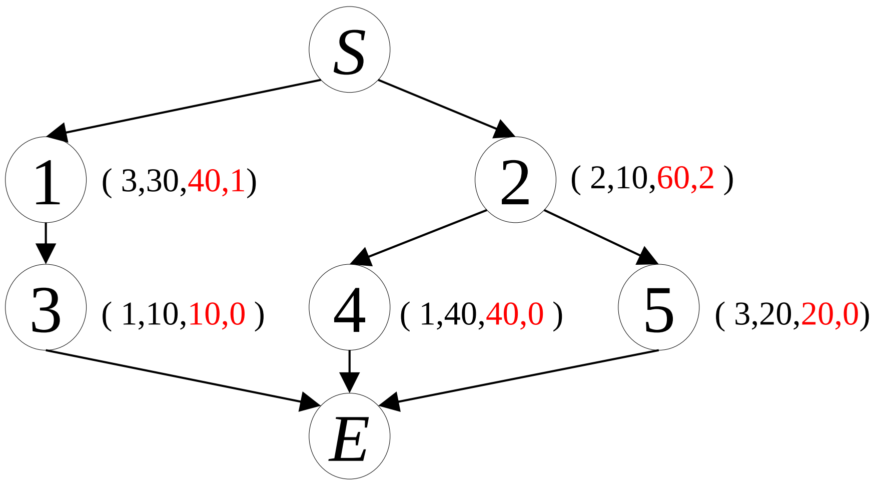


Figure 10. A task graph which was marked the CP and MISF

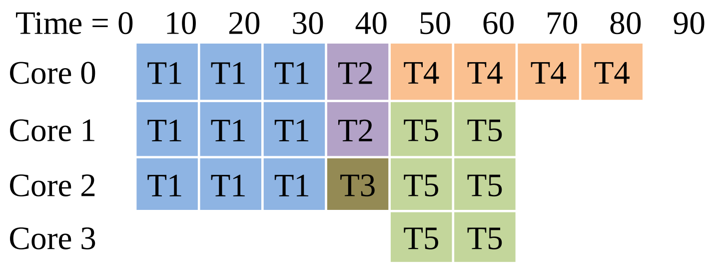


Figure 11. The Scheduling result obtained by PCS of task graph in Figure 10.

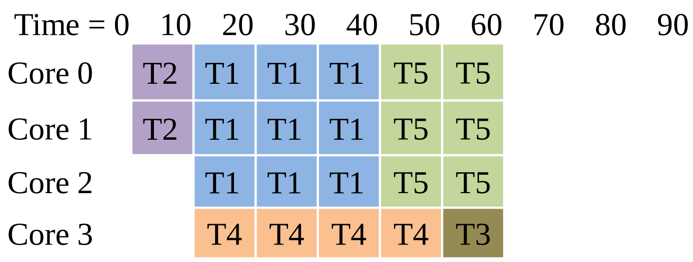


Figure 12. The optimal scheduling result of task graph in Figure 10

5.3. The Overall Dual-mode Scheduling Algorithm

The list based task scheduling algorithms such as the PCS have a *ReadyList*, The *ReadyList* contains all tasks whose parent tasks have been scheduled. At each step, list based algorithms schedule a task from the *ReadyList* according to a static priority and update its *ReadyList*. The scheduling process finished until all tasks in task graph are scheduled.

In this chapter, we propose a new algorithm based on a variation of list scheduling. The major difference between pure list scheduling algorithms is the new algorithm has two ready lists. Here, we denote the two types of ready lists as *ReadyList1* and *ReadyList2*. *ReadyList1* and *ReadyList2* are similar to the ready list in the pure list scheduling algorithm which contains a set of schedulable tasks. However, there is a restriction of tasks contained in *ReadyList1*, that is, the tasks in *ReadyList1* must be with the degree of data parallelism between *R***IdleCores* and *IdleCores*. The *IdleCores* denotes the current number of idle cores, and *R* (filling ratio) is a constant between 0 and 1. The task priority strategies of the above two ready lists are as follow.

- Tasks in *ReadyList1* are scheduled by the PCS priority
- Tasks in *ReadyList2* are scheduled by the CS priority.

The PCS priority is the same priority strategy in PCS algorithm, and CS priority means:

1. If the critical path of task i is longer than that of task j, task i has a higher priority than task j
2. In case tasks i and j has the same length of critical paths, if task i has more immediate successors than task j, task i has a higher priority than task j.

We can use formal definition of CS as:

$$PriorityCS_i = U \cdot C_i + S_i \tag{13}$$

Below is a fundamental algorithm of the dual-mode algorithm.

1. Initialize *ReadyList1*, *ReadyList2* and *IdleCores*, as well as calculate the two priorities for all tasks.
   - *ReadyList1* = Ø
   - *ReadyList2* = Ø
   - *IdleCores* = the number of total cores.

2. Select a task at *ReadyList1* which with the highest PCS priority.
3. If *ReadyList1* does not contain any task, select a task at *ReadyList2* which with the highest CS priority.
4. Finish if all tasks have been scheduled. Otherwise, update *ReadyList1*, *ReadyList2* and *IdleCores*.
5. Go back to step 2.

The *R* is a parameter which uses to limit the tasks which are contained in *ReadyList1*, the behavior of dual-mode algorithm will be changed by using different *R*. For example, when $R = 0$, the dual-mode algorithm produces same scheduling result as PCS, because the *ReadyList1* contains all schedulable tasks, then never go to the step 3. On the other hand, when $R = 1$, *ReadyList1* only contains the tasks can fully utilize the idle cores in current time, this is, the dual-mode algorithm tries to schedule the task with highest PCS priority value, and can completely utilize the current degree of data parallelism. If no such tasks exist, the dual-mode algorithm schedules tasks by CS priority. The dual-mode scheduling described above also summarized in Figure 13.

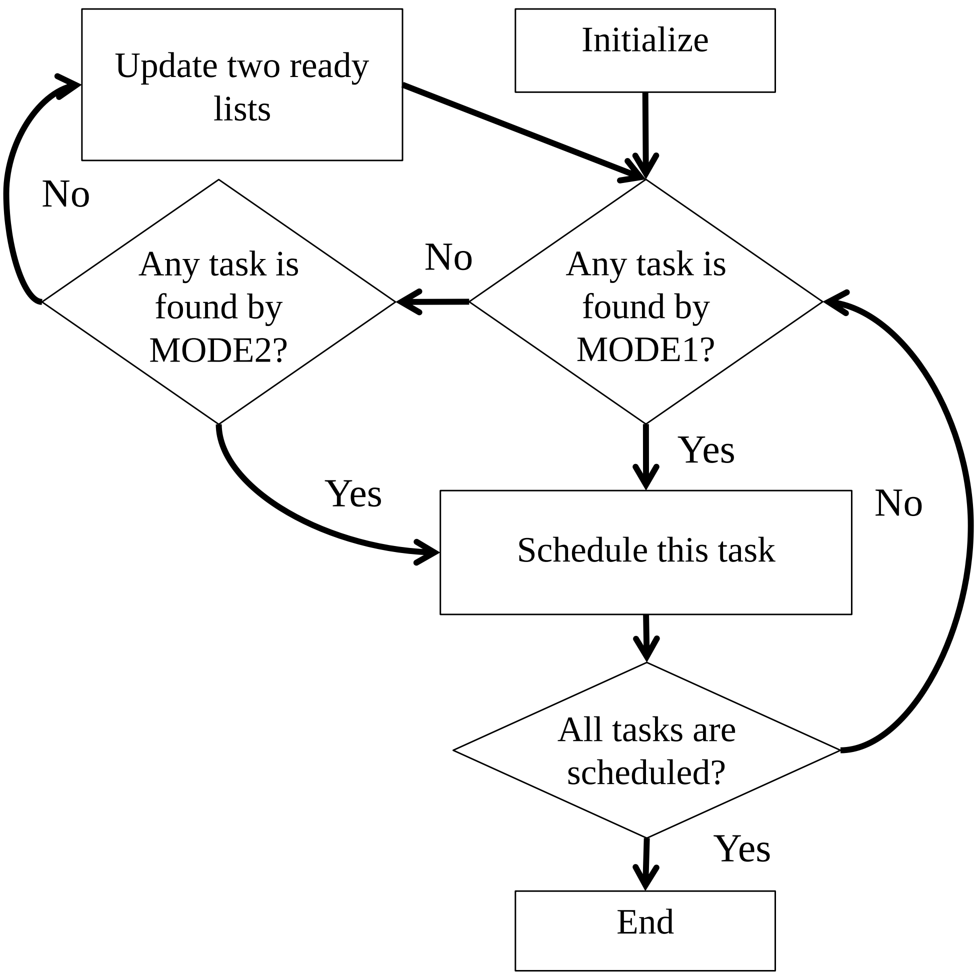


Figure 13. Flowchart for Dual-mode Scheduling Algorithm

To summarize, dual-mode algorithm tends to find a task to utilize the degree of data parallelism completely at the beginning. The parameter *R* indicates the lowest resource usage. This process is called MODE1. A more detailed description of MODE1 is available in the Listing 2. If the dual-mode algorithm fails to find any suitable task in MODE1, it will switch to MODE2, in this mode, dual-mode algorithm tends to schedule tasks with longer critical path primarily. The Listing 3 outlines the MODE2.

Listing 1 presents the pseudo-code for the overall dual-mode algorithm. Listing 2 to 5 give precise descriptions of all subroutines of Listing 1. The time complexity of the dual-mode algorithm is $O(N^2)$. Here, N denotes the number of tasks of a task graph. First, The algorithm for calculating the critical path lengths of each task takes $O(N^2)$. And to compute the two different priority values using three factors (P, C and S) takes O(N) time to run. The two ready lists update and select one task to schedule at each loop takes O(N). The main loop of the dual-mode algorithm shown in Listing 1 will repeat N times to schedule all the tasks. Therefore, the overall complexity of the dual-mode algorithm is $O(N^2)$.

Listing 1.Dual-algorithm

```
Calculate C (critical path) for all tasks;
Calculate S (number of immediate successors) for all
tasks;
Calculate priority PCS of all tasks;
Calculate priority CS of all tasks;
Initialise
do
   begin
    task = MODE1;
    if task not exist
      begin
      task = MODE2;
      endif
    if task not exist
      begin
      INCREASE_IDLE_CORE;
      continue
      endif
    SCHEDULE;
    end
while there are unscheduled tasks exist
```

Listing 2 MODE1

```
for i from 1 to n
begin
if all preceding tasks of task i are
   completed AND R*IdleCores < P
   (the degree of data parallelism)
   of task i < = IdleCores
       begin
       Add task i to ReadyList1;
       end
end
return task in ReadyList1 with highest priority_PCS
```

Listing 3. MODE2

```
for i from 1 to n
begin
if all preceding tasks of task i are
   completed and IdleCores < P
   (the degree of data parallelism)
   of task i <= IdleCores
       begin
       Add task i to ReadyList2;
       end
end
return task in ReadyList2 with highest priority_CS
```

Listing 4. INCREASE_IDLE_CORE

```
t = the second smallest occupied times of all cores;
idle_cores_number = 0;
for i from 1 to m
    begin
     if t <= occupied time of core i
          begin
          occupied time of core i = t
          idle_cores_number =
          idle_cores_number + 1;
          end
    end
```

Listing 5. SCHEDULE

```
t = the smallest occupied times of all cores;
p = 0;
for i from 1 to m
      begin
      if t == occupied time of core i AND p < the
            P of task i
            begin
            Schedule task on core i;
            Update the occupied time of core i;
            p = p + 1;
            end
end
```

5.4. Experiments

The proposed dual-mode algorithm was implemented in C. We used Standard Task Graph (STG) Set [31] [55] which was developed at Waseda University to evaluate the effectiveness of the proposed algorithms. The scheduling results obtained by the dual-mode algorithm were compared with PCS algorithm (defined in chapter 4). Since task graphs in STG do not indicate the degree of data parallelism for each task, we randomly assigned it to all of the tasks. The number of cores was changed from two to sixteen. We conducted all of our experiments on dual Xeon processors (E5-2650, 2.00Hz) with 128GB memory.

5.4.1. Results for Random Task Graphs

First, we evaluated our algorithm using 20 random task graphs which with 50 tasks. Because using different parameter *R* greatly affect the following scheduling procedures of dual-mode algorithm. we conducted same experiments but the *R* value is set to 0.7, 0.8, 0.9 and 1.0.

Figure 14 shows the average scheduling lengths of task graphs with 50 tasks. We compared the proposed dual-mode algorithm with PCS algorithm. And the all scheduling results normalized to the results by the result of PCS. From Figure 14 we can find that the dual-mode algorithm always gets better results than PCS, especially when the CPU with more cores.

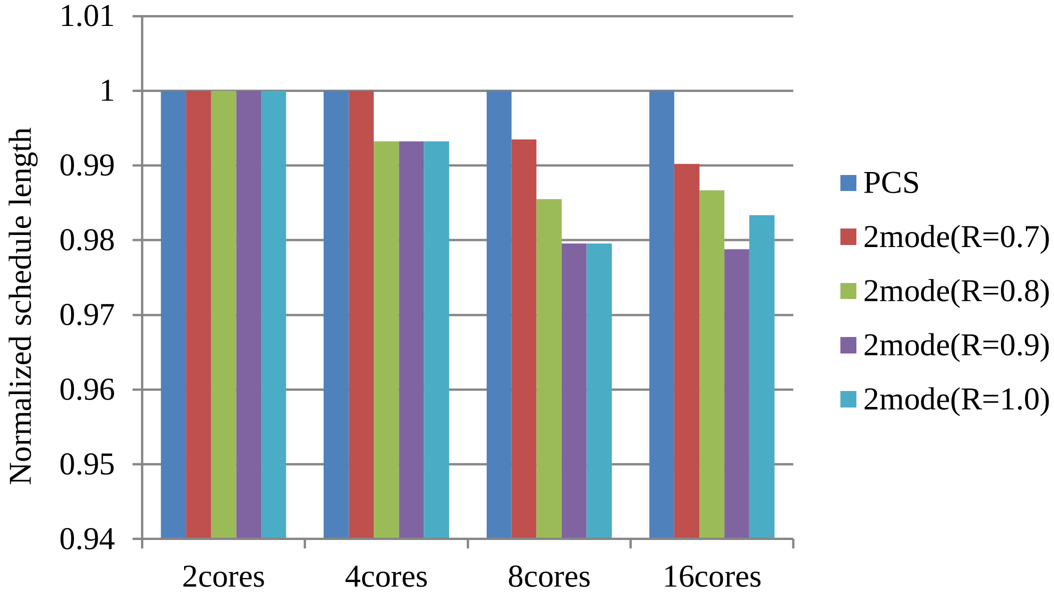


Figure 14. Averages of normalized schedule lengths for task graphs with 50 tasks.

Next, we conducted similar experiments as Figure 14, but using 20 random task graphs which with 100 tasks. As you can see from Figure 15, the dual-mode algorithm yields better scheduling results than PCS. The two figures Figure 14 and Figure 15 show very clear on that the effectiveness of dual-mode algorithm is significantly improved when compared with PCS.

However, the above experiments cannot clearly determine the best *R* value. In order to find the best *R* value on average, we evaluated our dual-mode algorithm while the *R* value changed from 0 to 1.

In Figure 16 and Figure 17, we scheduled 20 task graphs which consist of 50 tasks and 100 tasks respectively. The *R* value of dual-mode algorithm is changed between 0 and 1 by 0.05 increment.

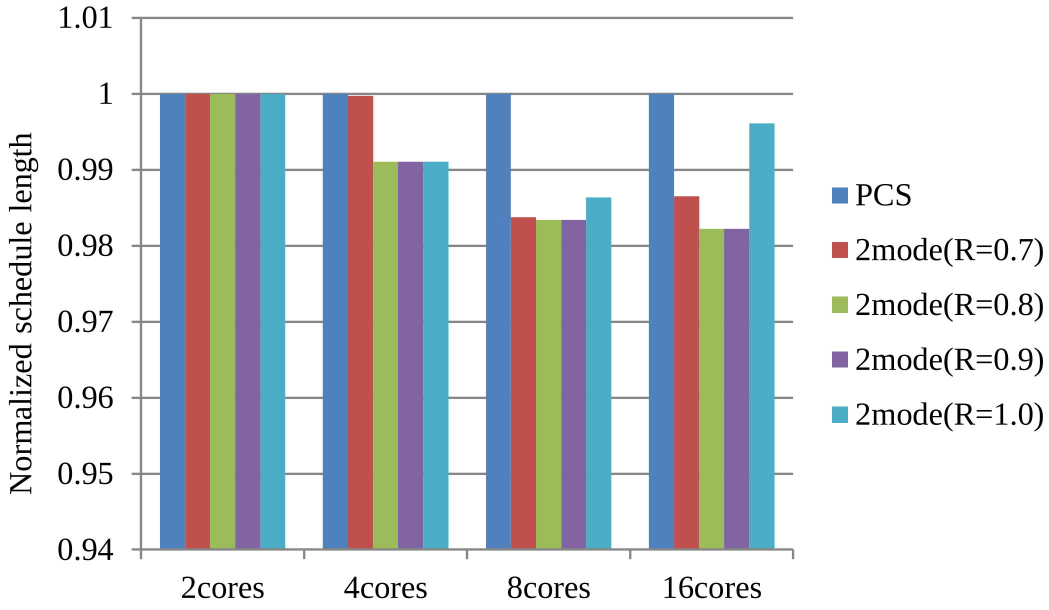


Figure 15. Averages of normalized schedule lengths for task graphs with 100 tasks.

As we can see from Figure 16 and Figure 17, if the *R* is small, the dual-mode algorithm produces exactly the same scheduling result obtained by the PCS algorithm. A small *R* helps MODE1 to contain more tasks, and fewer tasks are scheduled by MODE2. In this case, most tasks scheduled by MODE1 using the PCS priority, therefore, the dual-mode algorithm with smaller *R* behaves similarly to the PCS algorithm.

On the other hands, if the *R* is large, MODE1 only schedules tasks which can completely utilize the current idle cores. If no such tasks exist, the dual-mode algorithm switches to MODE2 and schedules tasks by CS priority.

We attribute the good results achieved by dual-mode to the fine balance between different priority strategies. By adjusting the value of *R*, we can change the percentage of task scheduled by PCS and CS strategies. The experimental results show, on average, *R* approximately equals to 0.85, the dual-mode algorithm yields good scheduling results.

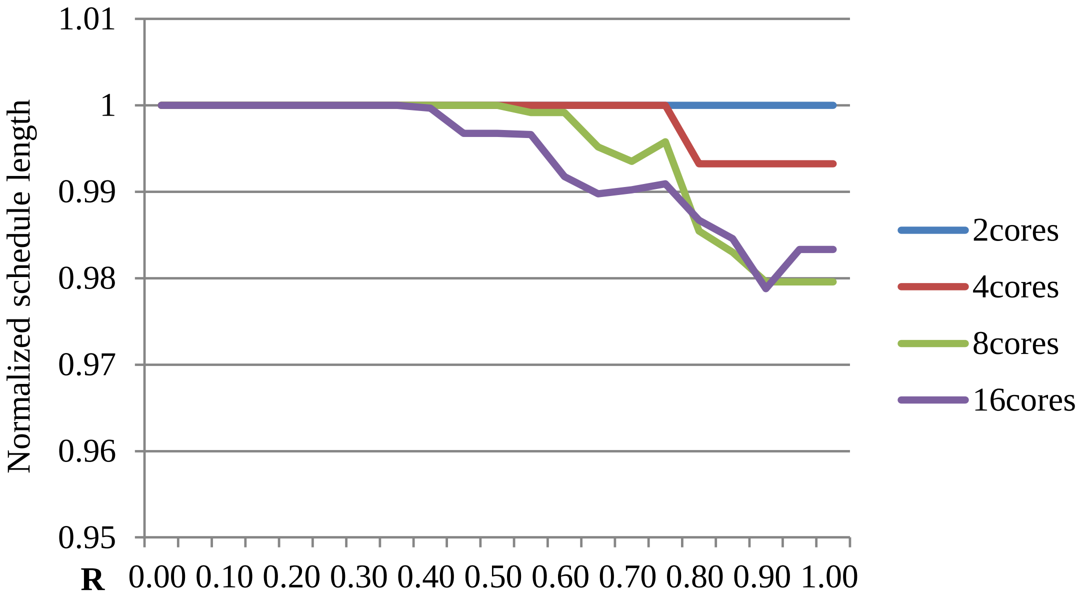


Figure 16. Schedule lengths with 50 tasks (R=0~1)

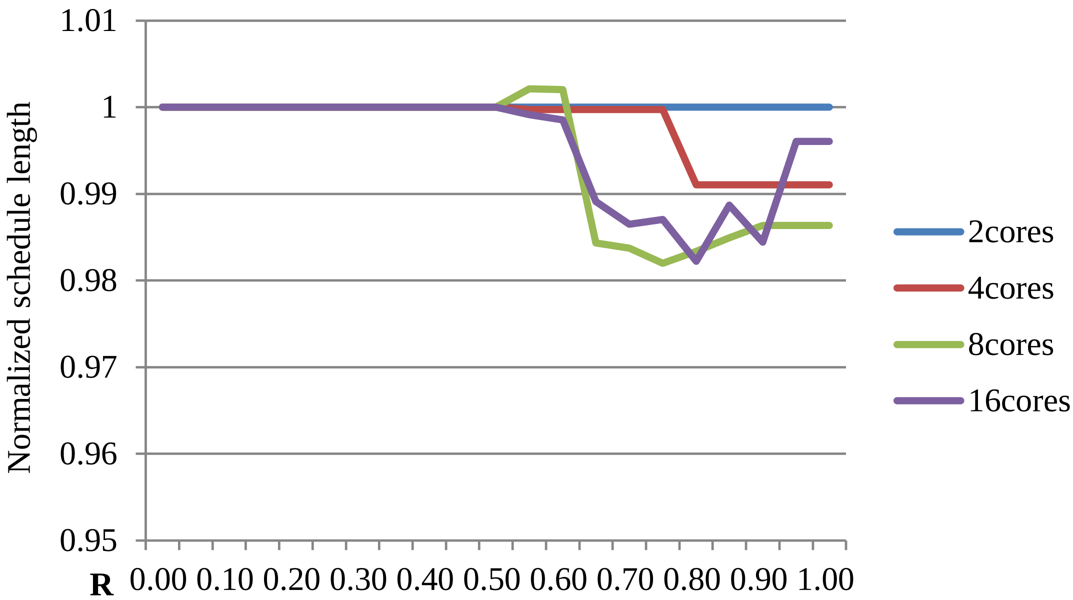
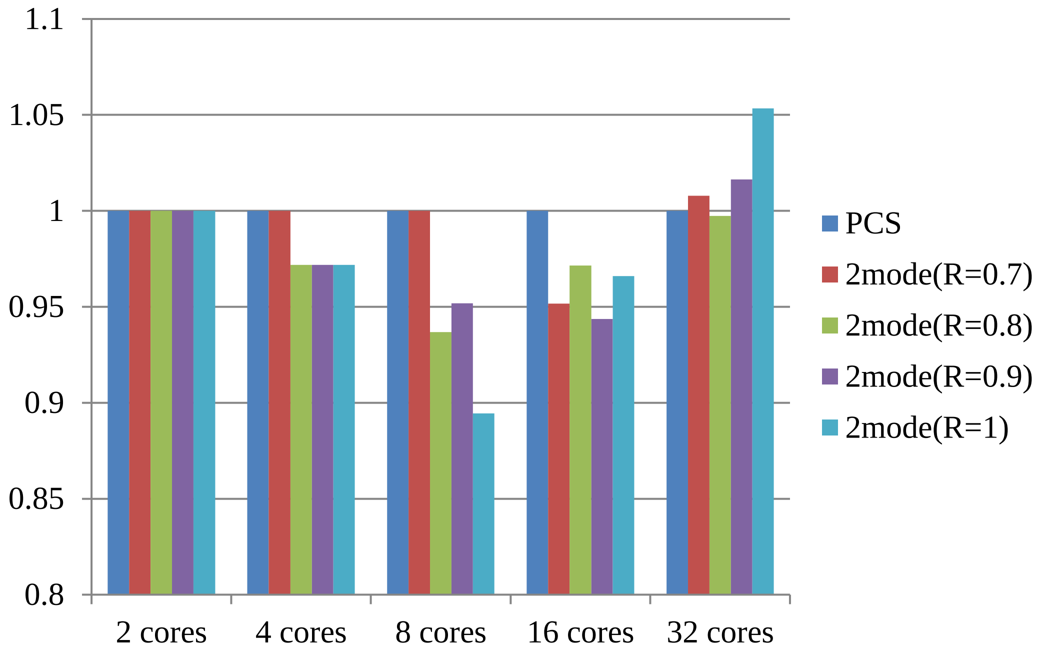


Figure 17. Schedule lengths with 100 tasks (R=0~1)

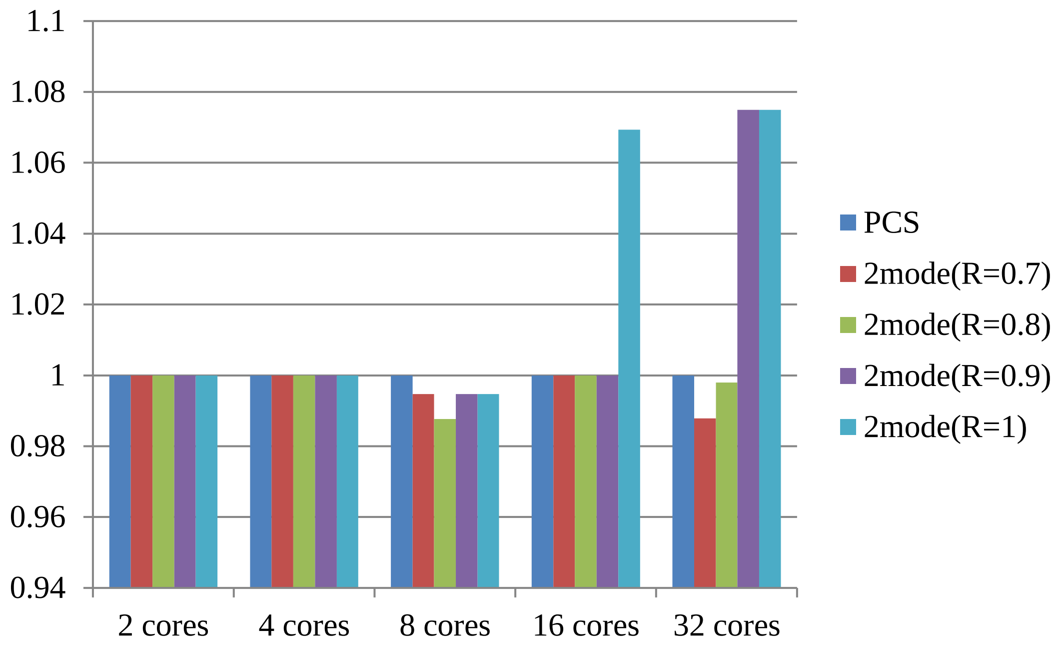
5.4.2. Results for Realistic Task Graphs

The next experiments, we used three task graphs developed from the realistic applications, i.e., (a) a part of fpppp from in the SPEC benchmarks, (b) robot control and (c) sparse matrix solver [55].

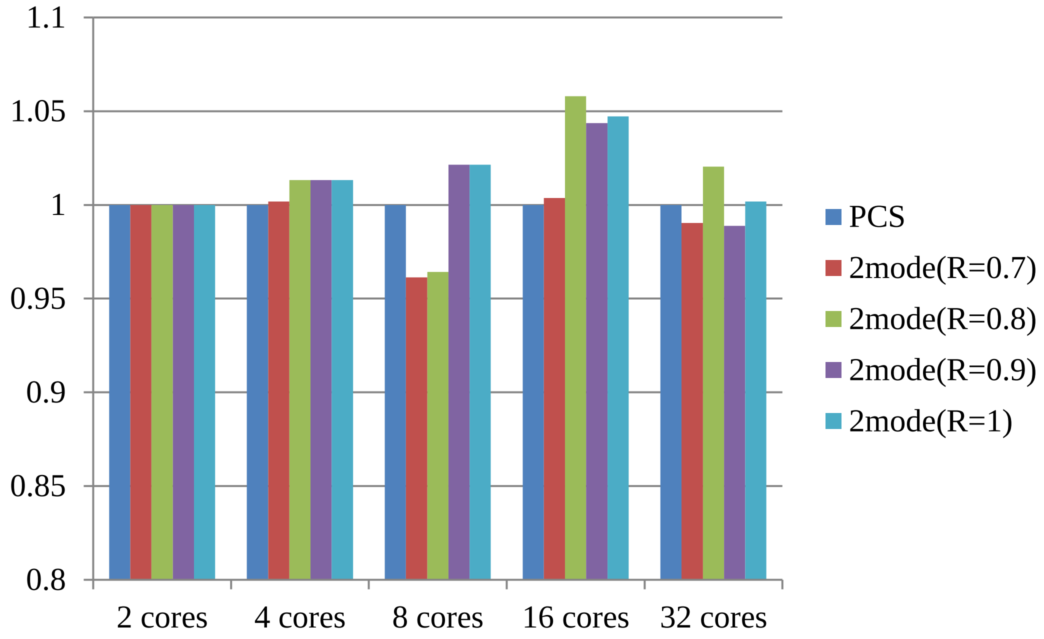
Figure 18 shows the average scheduling lengths of three realistic task graphs. Same as Figure 14 and Figure 15, we normalized all the results to the result of PCS We found the dual-mode algorithm obtained good schedules when *R* value equal to 0.7 or 0.8. However, for fpppp on 16 cores, sparse on 16 and 32 cores, dual-mode algorithm failed to get shorter scheduling result than PCS. The quality of dual-mode largely depends on the structure of task graph and the target system model. In generally the effectiveness of heuristic need evaluate by a lot of experiments. Therefore, we attribute the poor results of sparse on 16 and 32 cores to isolated experiments.



(a) robot



(b) Sparse



(c) fpppp

Figure 18. Normalized schedule lengths for realistic task graphs.

Chapter 6.

Genetic Algorithm

In chapter 5, dual-mode algorithm has greatly improved the list scheduling base method for task scheduling. But in essence, list scheduling algorithms use static rule based on experience or statistics. If the static rules are over-optimized by specific task graph, it may difficult to produce optimal solutions or near-optimal to other problems.

In contrast, meta-heuristics provide a framework for solving for the optimization problem. They adopt some random strategies to search larger solution space, and usually provide a mechanism to avoid local-optimal resolution. The genetic algorithm is one of the most famous meta-heuristics which inspired by natural selection. Due to its efficiency to solve combinatorial optimization problems, there are many task scheduling algorithms are based on genetic algorithm. Unfortunately, majority of those works only consider the data task parallelism. Many studies have shown that, for a large class of large computational applications, exploiting both task and data parallelism yields better speedups compared to either pure task parallelism and pure data parallelism.

In this chapter, we present an approach of task scheduling based on a genetic algorithm to solve the scheduling problem with both task and data parallelism. Different from traditional genetic algorithms for task scheduling [19] [21] [20], we propose a novel chromosomal representation for task scheduling and corresponding genetic operators to reduce the search space and improve the computing speed. Because the genetic algorithm needs to generate and evaluate a large number of chromosomes, it usually requires a long execution time. In this chapter, we also parallelize our algorithm with OpenMP.

6.1. Genetic Algorithm Fundamentals

Genetic algorithms are a kind of meta-heuristic algorithms inspired by the processes observed in natural selection [54]. Genetic algorithms think of a set of candidate solutions for a problem as biological population, and the fitness of each individual is evaluated according to Darwin's theory: "Survival of the fittest". The fitter ones are more likely selected and produce next generations. During this breeding process, the spontaneous mutations occur, creating individuals that are better adaptable to the environment. The basic terms of genetic algorithms used in this paper are shown and defined in Table 5.

Table 5. Basic terms of a genetic algorithm.

Terms	Meaning
Environment	Problem
Individual	Solution to a problem
Chromosome	Representation for a solution
Population	Set of solutions represented by chromosome
Gene	The basic element in chromosome
Fitness	The degree of adaptation for individual to the environment
Selection	The operation of choosing parents
Crossover	The operation of producing child
Mutation	The operation of randomly alter genes

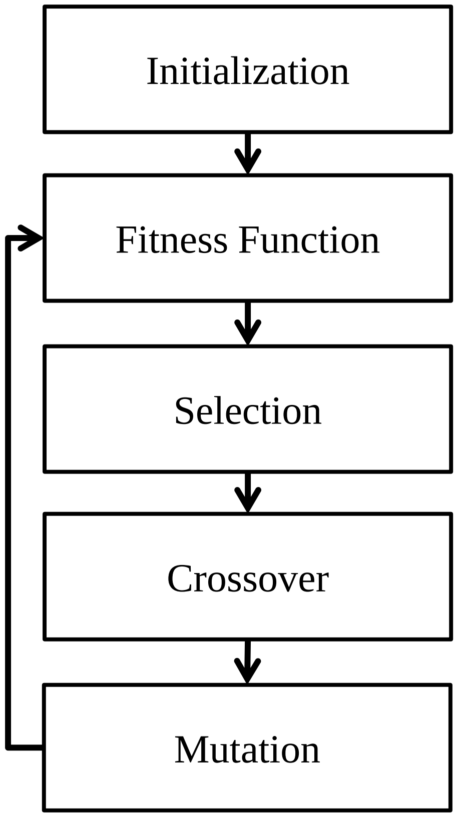


Figure 19. The flow chart of Genetic algorithm

Typically, a genetic algorithm can conclude as Figure 19, which consists of the following steps.

- Initialization: Generate the initial population.
- Calculation of the fitness: The fitness of each individual is calculated according to the definition of the problem.
- Selection: Select the adapted individuals as parents for the next generation.
- Crossover: Vary the programming of a chromosome (or chromosomes) from one generation to the next generation.
- Mutation: Alter genes for individuals.
- Go to step 2 until the stopping criteria is reached.

6.2. The Proposed Genetic Algorithm

This section proposes a new algorithm for the task scheduling problem defined in Chapter 3. In principle, our algorithm is based on the basic genetic algorithm described in Section 6.1. This section presents details of each step of the genetic algorithm tailored for our scheduling problem.

6.2.1. Representation of a Chromosome

In genetic algorithms, a chromosome is a set of strings, which represent a potential solution for the problem. Defining an adequate chromosome is one of the most important issues for a successful application of genetic algorithms. Since all genetic operators are defined on chromosomes, a good chromosome representation will make the genetic operators easier to implement and limit the unnecessary search space.

Several different types of chromosomes for task scheduling problems were proposed in previous works. All of them contain the information on both tasks scheduling and mapping, which means that both the ordering of task execution and the mapping between tasks and cores are encoded. This kind of chromosomes may not be very efficient for task scheduling with task and data parallelism, because the tasks can be mapped on multiple cores, therefore, the length of chromosomes may tend to be very long. We intend to find a more condensed representation of chromosomes. Our proposed chromosome only encodes information about the ordering of task execution, while ignoring the mapping between tasks and cores. This representation also reduces greatly the size of search space and improves the performance of the algorithm.

The proposed chromosome representation is an array of N elements where N represents the number of tasks. This array determines the sequence of the processing of the tasks. Figure 20 shows an example of the proposed chromosome. In Figure 20, task1 (T1) will be scheduled first, the next one is task2 (T2), and so on.

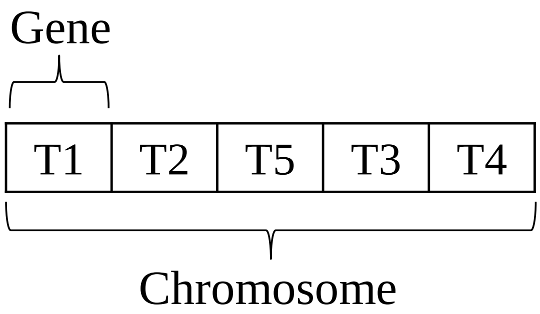


Figure 20. An example of chromosome

Another important issue on the chromosome representation is that the precedence relation between tasks must be maintained. A chromosome is called valid if the scheduling solution represented by the chromosome satisfies the precedence relation among the tasks.

6.2.2. Initialization

Our algorithm begins with a set of randomly generated candidate solutions represented by chromosome which is defined in Section 6.2.1. Our algorithm of initialization guarantees that all the generated chromosomes are valid.

The pseudocode of initialization is shown in Listing 6.

Listing 6. The algorithm for initialization.

```
1  order[0] = 0; // dummy task S
2  for i = 1 to N   // N is the number of tasks in task graph
3      min = MAX(order[Ti_parent]) + 1;
4      order[i] = RANDOM_BETWEEN(min, i); // random number between min
5          and i
6      for j = 0 to (i-1)
7          if(order[j] >= order[i]) then
8              order[j] = order[j] + 1;
9          endif
10     endfor
11 endfor
12 for i = 1 to N
13     C[order[i]] = i;
14     endfor
```

The initialization algorithm assumes that a task with a larger ID is not a parent for tasks with smaller ID. If the task graph does not satisfy this assumption, we need to reorder the tasks before the initialization algorithm. In the algorithm, array C[] represents a chromosome, and its elements represent genes. As shown in Figure 20, the i-th gene, i.e., C[i], represents the i-th scheduled task. Ti_parent indicates a parent task for task i. The scheduling order of task i, i.e., order[i] in Listing 6, is randomly generated, but is guaranteed to be later than its parent tasks (lines 3 and 4). Thus, the chromosome generated by the algorithm is valid.

6.2.3. Fitness Function

The fitness function is used to decode a chromosome and assign it a fitness value. The fitness value in our genetic algorithm represents the scheduling length. We propose a deterministic algorithm to schedule the tasks according to the chromosome and the task graph. This algorithm also restores the mapping information, that is, on which cores the tasks are mapped. The algorithm of our fitness function is as follows.

1. Ti = the first gene in the chromosome.
2. Remove Ti from the chromosome.
3. Calculate start time of Ti as follows:
    3.1. a = MAX(finished time of Ti's parents).
    3.2. b = earliest time at which an enough number of cores for executing Ti become free.
    3.3. Start time of Ti = MAX(a, b).
4. Finish time of Ti = start time of Ti + execution time of Ti.
5. Assign the cores which were selected at step 2.2 to Ti.
6. Update the occupied time of the cores.
7. Go back to step 1 until the chromosome is empty.
8. Fitness value = MAX(finish times of all tasks).

In essence, the above algorithm schedules tasks as early as possible in the order specified by the chromosome.

6.2.4. Selection

The selection operator is guided by the fitness value of each chromosome calculated by the process presented in Section 4.3. Chromosomes with better fitness value have a larger probability to survive. In the past work on genetic algorithms, different approaches were used in the selection operators such as roulette wheel selection, rank selection, and steady-state selection. Our algorithm uses the roulette wheel.

In roulette wheel selection, each chromosome in the population is allocated a segment on a virtual roulette wheel of a size proportional to its fitness. The adapter chromosomes have a larger segment; it means such chromosomes are more likely to be selected when the wheel is spin. This size of the segment for each task is calculated as below:

$$p_i = \frac{\exp(-a(f_i - f_{min}))}{\sum_j^N p_j} \tag{14}$$

f_{min} denotes the minimum fitness value in population, and f_i denotes the fitness value of current chromosome. The part of denominator is a normalization factor. The parameter α must be greater than 0, and the larger α is, the more likely to select the chromosome with higher fitness value (If α is 0, the chromosomes with different fitness values will have same chances of being selected).

6.2.5. Crossover

The crossover operator is analogous to the biological crossover. Two chromosomes are chosen from the population, and the child chromosomes are produced from them.

Since our chromosome represents the order of task execution, simply exchanging part of genes between two chromosomes may produce invalid chromosomes which violate precedence constraints among the tasks. Therefore, we propose the following algorithm to ensure the generated chromosomes are valid.

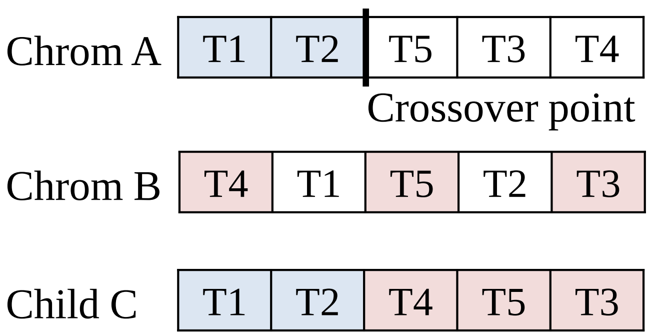


Figure 21. An example of crossover.

1. Select two chromosomes, A and B, from the population.

2. Randomly choose a crossover point in chromosome A.
3. Copy the genes in the left segment of the crossover point in chromosome A, to a new chromosome C.
4. Copy the genes which were not selected in step 3 to the child C in the order of chromosome B

This algorithm is illustrated in where a new chromosome C is generated from two chromosomes A and B by the crossover operation.

In Figure 21, two genes T1 and T2 in chromosome A are copied to chromosome C, and three genes T4, T5 and T3 are copied from chromosome B to C. As long as the two parent chromosomes, i.e., A and B, are valid, the child chromosome C is also valid.

6.2.6. Mutation

The mutation operator randomly alters one or more genes. In genetic algorithms, selection operators remove inferior chromosomes, but lose the diversity in the population. Mutation is a very important mechanism to recover the diversity. Hence, the mutation operator gives us the possibility of producing better children than their parents. Our mutation operator also guarantees that the chromosomes after mutation are valid.

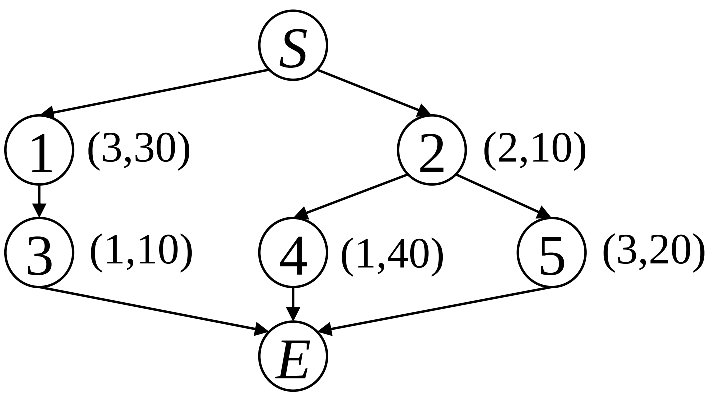


Figure 22. A task graph.

In the proposed chromosome, the value of i-th gene indicates the task whose execution order is i-th. Our mutation changes the order of execution of the task by the following algorithm.

1. Generate a random number *p* (from 0 to 1) for each task.
2. Go to step 2 if $p > m$, where *m* is a given threshold that the chromosome is subjected to be mutated. Otherwise, go back to step 1.
3. Calculate the new location of the selected task as follows:
   3.1. *upper* = the current location of the task.
   3.2. *lower* = MAX(locations of its parents) + 1.
   3.3. New location of the task = RANDOM_BETWEEN(*lower, upper*).
4. Move the task to the new location and slide other tasks accordingly.

Figure 23 shows an example of mutation for the task graph in Figure 22. Assume that T4 in the chromosome in Figure 23(a) is selected for mutation in steps 1 and 2. According to Figure 22, T2 is a parent of T4. Therefore, T4 cannot be moved before T2, and there

T1	T2	T5	T3	T4

(a) A chromosome

T1	T2	T4	T5	T3
T1	T2	T5	T4	T3
T1	T2	T5	T3	T4

(b) Possible mutations

Figure 23. An example of mutation.

exist three possibilities for mutation of T4 as shown in Figure 23 (b). Our mutation algorithm chooses one of the three mutations randomly.

6.2.7. Parallelization of the Algorithm with OpenMP

The genetic algorithm may require an unacceptably long execution time because a large number of chromosomes must be generated and evaluated. Therefore, we use the parallelization technique to improve computational efficiency on multicore platforms. There are various types of parallelization technologies such as Pthreads, C++11 STL threads, OpenMP, Intel TBB, CUDA, and OpenCL. We have chosen OpenMP because of its easiness and flexibility on popular multicore platforms running on Linux or MS-Windows.

OpenMP is an API for writing multi-threaded applications on shared memory multi-processor architecture. In our genetic algorithm, a data dependency occurs when calculating the normalization factor in the selection operator, but otherwise, all of the genetic operators can be performed independently. Based on the above observation, we propose the parallelization framework of the algorithm as shown in Figure 24.

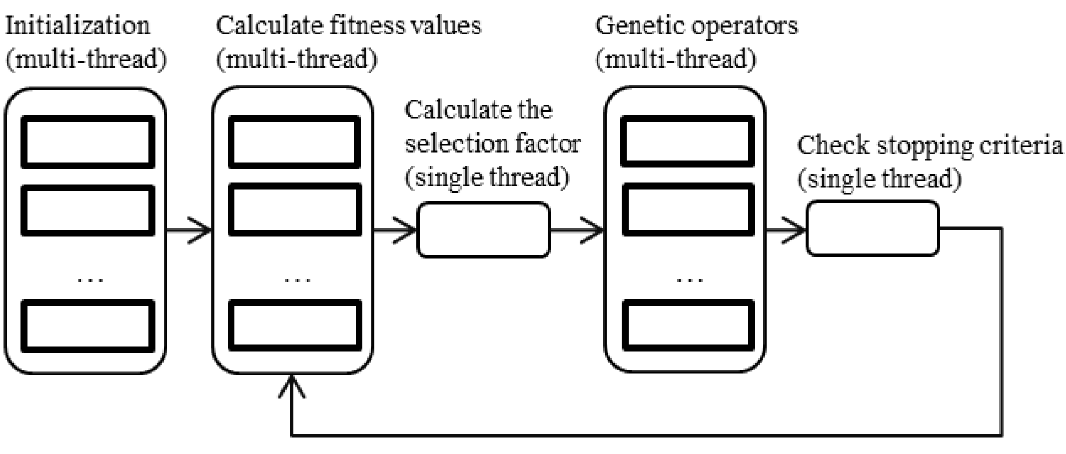


Figure 24. The parallelization framework.

6.3. Experiments

The proposed algorithm was implemented in C++. We evaluated our algorithm with the Standard Task Graph (STG) [55]. We used 20 sets of 50 tasks and another 20 sets of 100 tasks. The number of cores was changed from two to sixteen.

We conducted all experiments on Intel Core i7 (Core i7-4790K, 4 cores / 8 threads) and 32GB memory on Ubuntu 14.04. In the discussion in Section 6.2, we have presented a set of important parameters. The parameters have strong effects on the execution time and the quality of results. Finding the optimal set of parameters is another important and hard mission, but these are not included in the scope of this article. We just set the parameters as summarized in Table 6.

The results of scheduling for task graphs with 50 tasks are shown in Table 9-a and Table 9-b, PCS and Dual-mode are compared with the proposed algorithm. For each benchmark, the best solution is marked in red. We can find that the proposed algorithm could successfully find best schedules for 157 test cases out of 160 within 12 hours.

Table 6. The list of parameters.

Terms	Value
Population size	16384
α (selection rate)	0.6
m (mutation rate)	0.05
Max generations	50

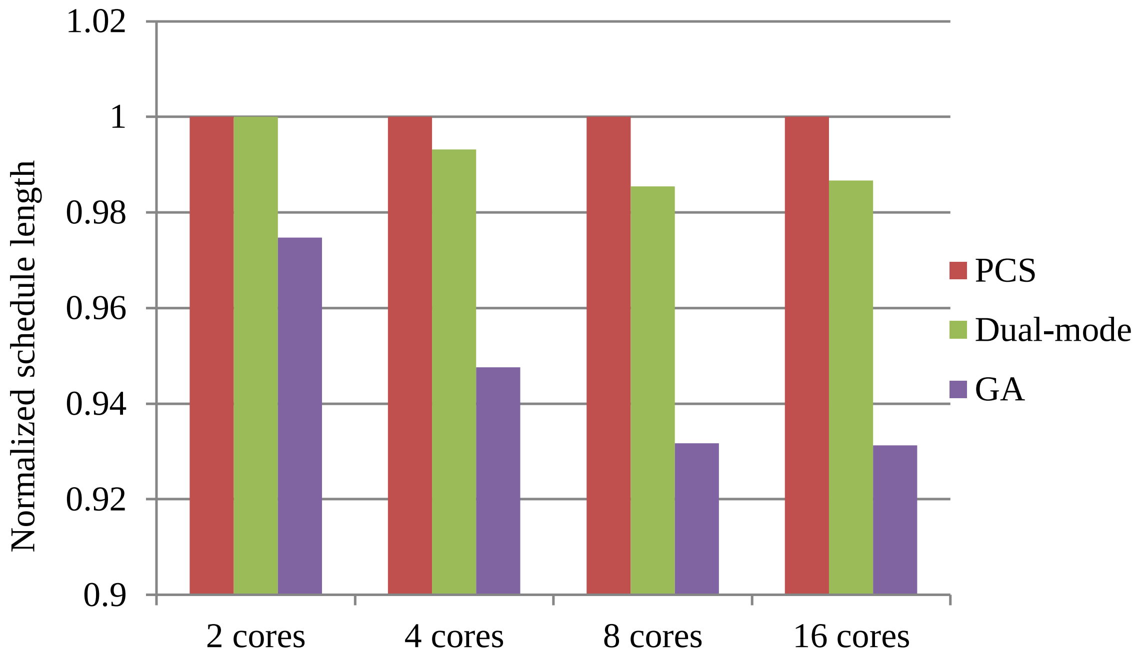


Figure 25. Results of three algorithms for task graphs with 50 tasks

For ease of comparison between the other algorithms, we normalize all results to PCS in Figure 25 and Figure 26. Figure 25 shows that for task graphs with 50 tasks, our genetic algorithm achieves 2.5%, 5.2%, 6.8% and 6.8% reduction in the scheduling length on 2, 4, 8 and 16 cores, respectively, compared with the PCS algorithm. And Figure 26 shows that for task graphs with 100 tasks, our genetic algorithm achieves 2.5%, 5.2%, 6.8% and 6.8% reduction in the scheduling length on 2, 4, 8 and 16 cores, respectively, compared with the PCS algorithm. Both figures show that our proposed genetic algorithm significantly improves the quality of the results.

The runtimes of the four scheduling algorithms are compared in Table 7. Because a large number of chromosomes need be generated and evaluated. The single-threaded implementation of the genetic algorithm is much slower than PCS or dual-mode algorithm. However, we use the parallelization technique to improve computational efficiency on

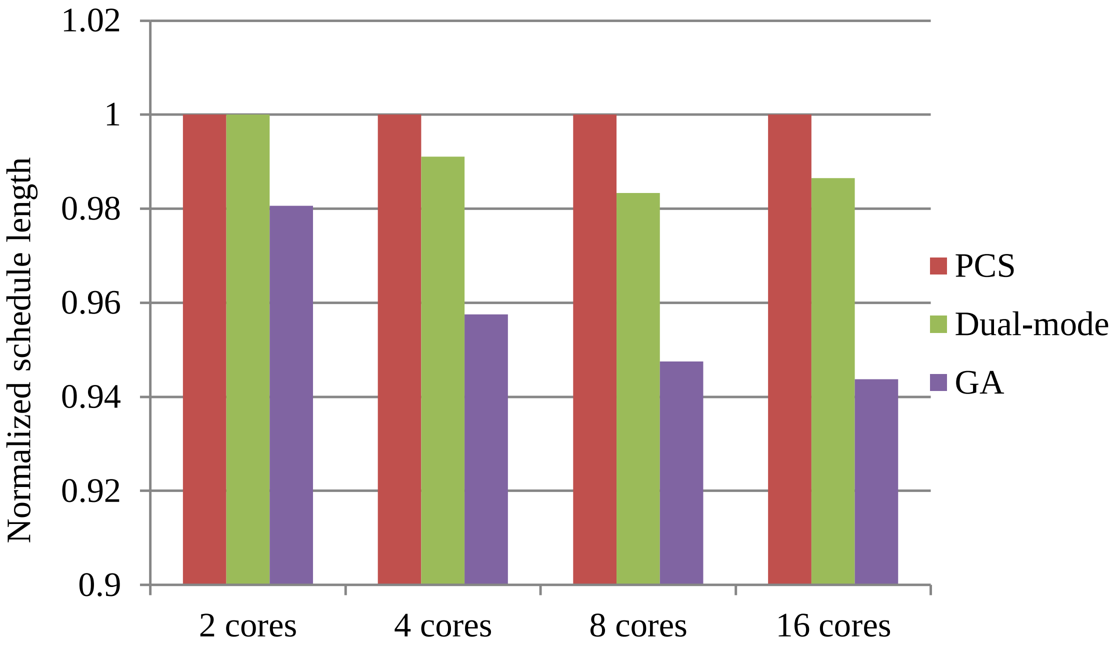


Figure 26. Results of three algorithms for task graphs with 100 tasks

Table 7. Runtimes of three scheduling algorithms (seconds).

	50 tasks	100 tasks
PCS	< 0.01	< 0.01
Dual-mode	< 0.01	< 0.01
GA	4.01 – 4.64	9.21 – 10.21
Parallelized GA	0.50 – 0.62	1.12 – 1.38

multicore platforms. The parallelized implementation achieved approximately seven times speed-up.

Table 8-a. Scheduling lengths for task graphs with 50 tasks on 2 and 4 cores

Task graph IDs	PCS	Dual-mode	GA	PCS	Dual-mode	GA
	2 Cores			4 Cores		
50-0000	203	203	196	168	167	159
50-0001	232	232	223	220	211	203
50-0002	188	188	186	173	170	165
50-0003	224	224	224	194	194	185
50-0004	177	177	174	167	167	166
50-0005	495	495	465	439	426	404
50-0006	351	351	340	275	270	266
50-0007	384	384	384	357	354	340
50-0008	434	434	429	409	407	390
50-0009	386	386	382	327	356	318
50-0010	153	153	153	131	131	129
50-0011	205	205	190	181	176	170
50-0012	208	208	193	197	192	179
50-0013	238	238	235	186	192	182
50-0014	195	195	195	171	167	161
50-0015	425	425	404	376	373	347
50-0016	374	374	368	318	319	292
50-0017	439	439	434	377	378	365
50-0018	428	428	423	403	396	363
50-0019	393	393	376	342	330	326

Table 8-b. Scheduling lengths for task graphs with 50 tasks on 8 and 16 cores

Task graph IDs	PCS	Dual-mode	GA	PCS	Dual-mode	GA
	8 Cores			16 Cores		
50-0000	149	148	144	156	151	140
50-0001	203	201	186	195	198	193
50-0002	161	153	143	150	146	131
50-0003	175	180	172	169	165	158
50-0004	150	155	147	158	157	145
50-0005	432	406	385	406	388	373
50-0006	259	246	239	268	249	246
50-0007	336	312	305	301	279	273
50-0008	366	354	337	360	345	327
50-0009	323	326	296	289	292	265
50-0010	127	125	121	126	127	123
50-0011	180	172	161	135	146	129
50-0012	183	178	171	174	169	166
50-0013	171	171	160	154	155	147
50-0014	166	163	148	160	147	147
50-0015	304	307	290	325	347	318
50-0016	269	266	254	286	293	259
50-0017	306	314	294	333	312	310
50-0018	358	354	329	342	344	326
50-0019	361	365	345	334	336	306

Table 9-a. Scheduling lengths for task graphs with 100 tasks on 2 and 4 cores

Task graph IDs	PCS	Dual-mode	GA	PCS	Dual-mode	GA
	2 Cores			4 Cores		
100-0000	431	431	431	388	376	367
100-0001	401	401	397	348	343	340
100-0002	459	459	448	413	424	403
100-0003	406	406	391	341	338	336
100-0004	393	393	393	354	366	346
100-0005	814	814	780	704	682	666
100-0006	868	868	826	785	737	721
100-0007	861	861	847	760	735	739
100-0008	796	796	792	701	706	694
100-0009	947	947	912	783	779	763
100-0010	464	464	446	385	392	366
100-0011	445	445	441	394	377	371
100-0012	469	469	451	432	434	405
100-0013	480	480	474	404	427	394
100-0014	391	391	386	354	334	329
100-0015	781	781	765	706	683	675
100-0016	764	764	751	667	646	614
100-0017	860	860	857	746	755	740
100-0018	724	724	722	628	626	601
100-0019	749	749	736	700	709	661

Table 9-b. Scheduling lengths for task graphs with 100 tasks on 8 and 16 cores

Task graph IDs	PCS	Dual-mode	GA	PCS	Dual-mode	GA
	8 Cores			16 Cores		
100-0000	356	337	332	335	333	317
100-0001	326	330	326	307	304	305
100-0002	380	372	354	365	355	335
100-0003	338	342	326	314	309	298
100-0004	340	327	322	317	307	303
100-0005	713	698	666	668	676	638
100-0006	712	703	680	687	666	654
100-0007	675	657	630	665	637	622
100-0008	637	638	618	607	610	597
100-0009	785	737	710	728	713	692
100-0010	338	327	317	362	354	324
100-0011	353	349	354	336	331	310
100-0012	431	423	380	410	421	387
100-0013	382	385	378	375	372	369
100-0014	327	319	319	313	306	305
100-0015	697	646	621	606	557	541
100-0016	625	657	607	648	645	601
100-0017	730	730	696	677	692	657
100-0018	657	642	625	591	595	567
100-0019	679	679	646	676	672	631

Chapter 7.

Branch-and-Bound Algorithm

In chapters 4 and 5, we discussed that use heuristics to find acceptable solutions in a short execution time. In chapter 6, the proposed genetic algorithm provides a robust approach to obtain higher quality solutions. However, the above methods are lack of ability to guarantee that the solutions are always optimal. Finding optimal solutions is indispensable to evaluate the quality of the algorithms. Also, optimal solutions also provide an in-depth understanding of the structure of the scheduling problem, which is very useful for theoretical research and the development of heuristic.

This section proposes an exacting algorithm for the scheduling problem with data parallelism. The proposed algorithm basically enumerates all possible solutions and explores them in a depth-first way with pruning non-optimal solution spaces.

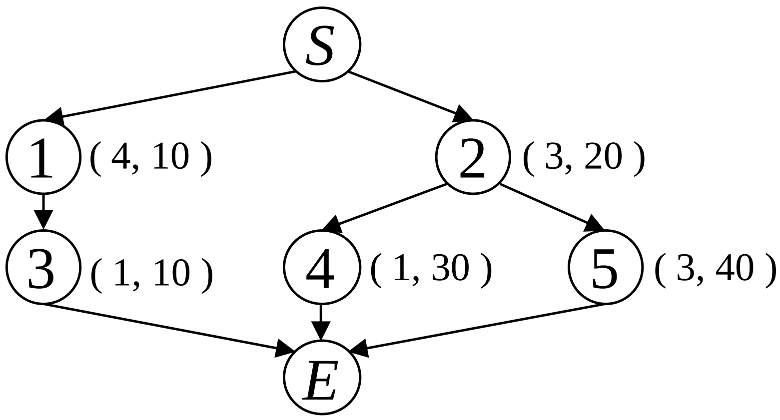


Figure 27. A task graph

7.1. Depth-First Search

Our algorithm uses a branching tree to enumerate all possible schedules systematically. For example, Figure 28 shows a branching tree for the task graph in Figure 27. 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.

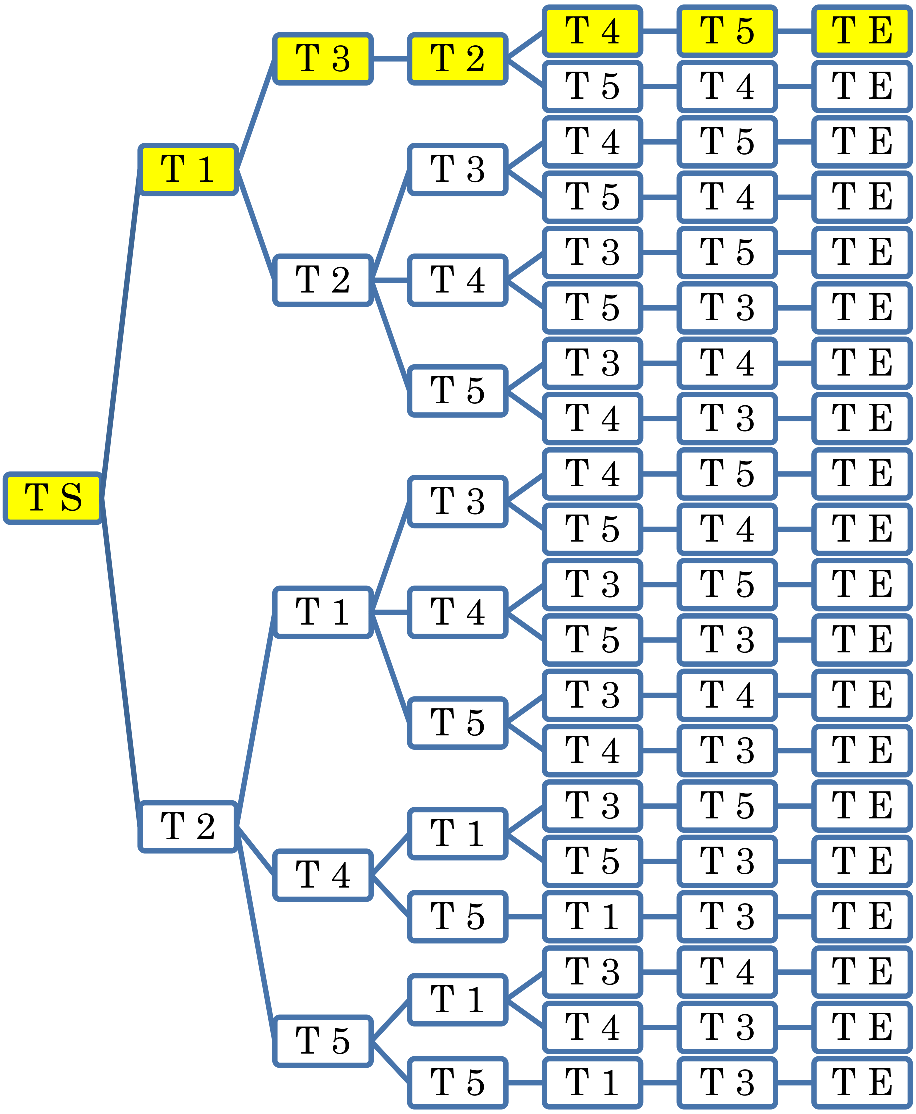


Figure 28. The tree enumerates all possible solutions.

Time =	0	10	20	30	40	50	60	70	80	90
Core 0	T1	T1	T1	T2	T4	T4	T4	T4		
Core 1	T1	T1	T1	T2	T5	T5				
Core 2	T1	T1	T1	T3	T5	T5				
Core 3					T5	T5				

(a). One schedule generated from a path (S → 1 → 3 → 2 → 4 → 5 → E)

Time =	0	10	20	30	40	50	60	70	80	90
Core 0	T1	T1	T1		T5	T5				
Core 1				T2	T5	T5				
Core 2	T1	T1	T1	T3	T4	T4	T4	T4		
Core 3	T1	T1	T1	T2	T5	T5				

(b). One schedule generated from a path (S → 1 → 3 → 2 → 4 → 5 → E)

Time =	0	10	20	30	40	50	60	70	80	90
Core 0	T1	T1	T1	T2	T5	T5				
Core 1	T1	T1	T1	T3	T4	T4	T4	T4		
Core 2				T2	T5	T5				
Core 3	T1	T1	T1		T5	T5				

(c) One schedule generated from a path (S → 1 → 3 → 2 → 4 → 5 → E)

Figure 29. Valid scheduling results for path (S → 1 → 3 → 2 → 4 → 5 → E)

The path in Figure 28 which is highlighted in yellow can be scheduled by many ways (e.g. Figure 29 (a), (b) and (c)). To be more precise, a path may denote more than one schedule, For example, path (S → 1 → 3 → 2 → 4 → 5 → E) may leads several different schedules which with same scheduling length. on the other hand, multiple paths also can generate the same schedule. For example, paths (S → 1 → 2 → 3 → 4 → 5 → E), (S → 1 → 3 → 2 → 5 → 4 → E) and (S → 1 → 3 → 2 → 4 → 5 → E) also result in the same schedule as shown in Figure 28 also result in the same schedule as shown in Figure 29 (a). The important point is that, for a given path, one of its optimal schedules can be found by a simple as-soon-as-possible (ASAP) strategy.

7.2. Branch-and-Bound Methods

Our algorithm travels the branching tree from the root to leaves in a depth-first order. However, travelling 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 presents four rules to prune unnecessary branches.

7.2.1. Related Pattern Rule

Let us consider the branching tree in Figure 30, Assume that our algorithm already visited partial schedule (1 → 2) and now we have reached (2 → 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 → 1) cannot be better than (1 → 2), and thus, we can prune further branches under (2 → 1). How to compare the two partial schedules is as follows. Figure 31(a) and Figure 31(b) show time charts of partial schedules (1 → 2) and (2 → 1), respectively. In Figure 31(a), one of the four cores are available at time ten, and then, task 3 is schedulable. Here, a task is schedulable if both of the following two conditions hold:

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

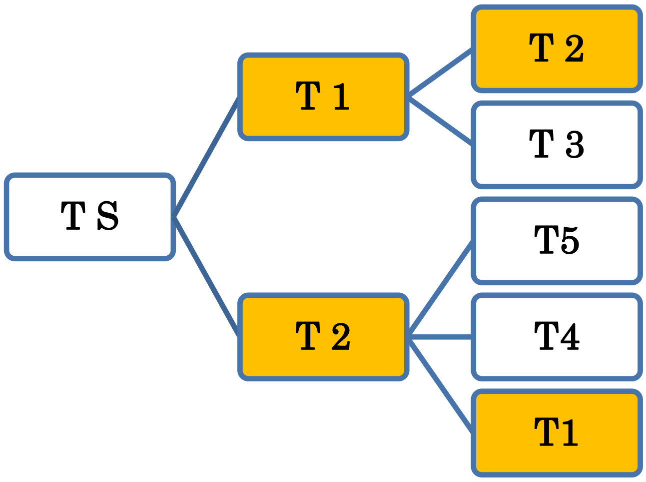


Figure 30: Related patterns

Similarly, tasks 3, 4 and 5 are schedulable at time 30 in Figure 31(a). In Figure 31(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 → 1) is a subset of that in partial schedule (1 → 2). For example, at time ten, a set of schedulable tasks in partial schedule (2 → 1) is empty, which is a subset of {3}. Then, it is guaranteed that no schedule under partial schedule (2 → 1) is better than the best schedule under (1 → 2), and therefore, branches under (2 → 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.

7.2.2. Exclusive Task Branch Rule

Let us consider the task graph in Figure 31. Initially, either task 1 or 2 is schedulable at time 0. In this case, scheduling task 1 first leads to an optimal schedule for 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.

There are two types of *exclusive* task.

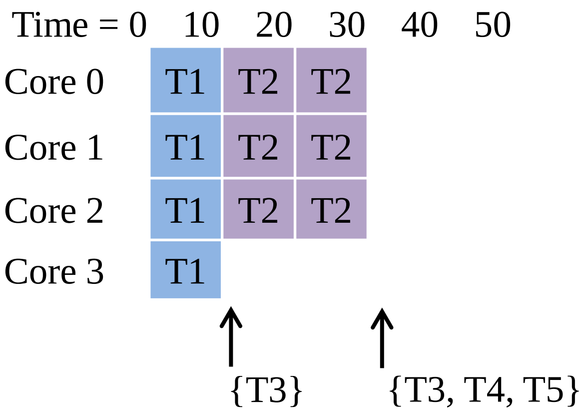
- A task has no parallelizable tasks.
- All of the parallelizable tasks of the task have been executed.

Delaying execution of exclusive tasks which can be scheduled at the earliest cannot minimize the scheduling 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.

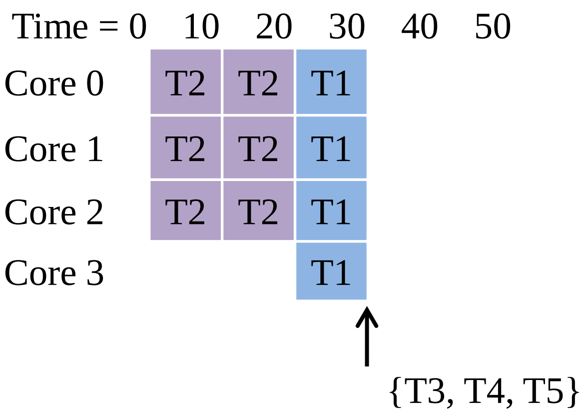
7.2.3. Reducing Meaningless Idle Time

Let us consider partial schedule in the branching tree shown in Figure 30. There are three branches from task 2, going to tasks 3, 4 and 5. If we look at the time chart in Figure 31 (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.

When 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.



(a) Partial schedule (1 → 2)



(b) Partial schedule (2 → 1)

Figure 31. Partial schedules with same tasks

7.2.4. Lower Bound Rule

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 scheduling 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.

$$\sum_j AT_j + \sum_{i\epsilon\boldsymbol{\varphi}} P_i \times T_i \geq N \times \mathrm{TOP} \qquad (15)$$

$$\sum_j AT_j - \sum_{i\epsilon\boldsymbol{\omega}} P_i \times T_i \geq \mathrm{MB} \qquad (16)$$

In the formulas, denotes the available time of core j. For example, in Figure 31 (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 (15) holds, the scheduling length under this node cannot be shorter than TOL, and therefore further branches are pruned.

In formula (16), 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.

$$\mathrm{TIT} = N \times \mathrm{TOL} - \sum_{i\epsilon\boldsymbol{all\ tasks}} P_i \times T_i \qquad (17)$$

7.3. Selection Rule

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 are multiple tasks with the same start time, we select a task based on the PCS strategy which was presented in Chapter 4.

Table 10. Optimal results for graphs with 10 tasks on 4 cores

Task graph ID	Scheduling length		Runtime (sec)	
	ILP	B&B	ILP	B&B
10-0000	32	32	6,823	< 1
10-0001	43	43	21,788	< 1
10-0002	26	26	60,012	< 1
10-0003	30	30	71,678	< 1
10-0004	36	36	2,588	< 1
10-0005	75	75	40,054	< 1
10-0006	70	70	46,245	< 1
10-0007	94	94	50,019	< 1
10-0008	121	121	6,115	< 1
10-0009	79	79	58,830	< 1

7.4. Experiments

We implemented our proposed scheduling algorithm in C++, and conducted two sets of experiments to test the effectiveness of the proposed algorithm. The experiments were conducted on dual Xeon processors (E5-2650, 2.00Hz) with 128GB memory. CPLEX fully utilized 16 cores on the host computer, while our algorithm ran on a single core as a single thread program.

In the first experiments, we use 10 sets of 10 tasks, derived from Standard Task Graph (STG) [55]. An integer linear programming (ILP) technique (see Section 3.3) was compared. In order to solve the ILP problems, IBM ILOG CPLEX 12.5 was used. The environment of experiments is dual Xeon processors (E5-2650, 2.00Hz, 128GB memory).

Table 10 shows scheduling results for 20 task graphs with 10 tasks on four cores. ILP and B&B denote the ILP technique using CPLEX and our branch-and-bound algorithm, respectively. The results in the table show that our algorithm yields the same scheduling length as the ILP technique 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 most 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 hours for CPLEX to find the optimal schedule for 10 tasks.

In the next set of experiments, we compared our branch-and-bound algorithm with three algorithms, the PCS, dual-mode and genetic algorithm which were introduced in Chapter, 4, 5 and 6 respectively. We used 20 sets of 50 tasks and another 20 sets of 100 tasks from STG. The number of cores was changed from two to sixteen. The runtimes of our branch-and-bound algorithm are limited to 12 hours or 1 second. When the runtime of our branch-and-bound algorithm exceeded the limited time, we suspended the algorithm and used the best schedule found by that time. The runtime of the PCS and dual-mode algorithms was less than 1 second in any case.

Table 11-a. Scheduling lengths for task graphs with 50 tasks on 2 cores

Task graph IDs	Scheduling length					B&B Runtime 12 hours (sec)
	PCS	Dual-mode	GA	B&B 12 hours	B&B 1 sec	
50-0000	203	203	196	196	196	2
50-0001	232	232	223	222	222	< 1
50-0002	188	188	186	186	186	< 1
50-0003	224	224	224	224	224	16
50-0004	177	177	174	174	174	< 1
50-0005	495	495	465	465	465	< 1
50-0006	351	351	340	338	338	< 1
50-0007	384	384	384	384	384	38
50-0008	434	434	429	428	428	< 1
50-0009	386	386	382	382	382	< 1
50-0010	153	153	153	153	153	4
50-0011	205	205	190	190	190	< 1
50-0012	208	208	193	192	192	< 1
50-0013	238	238	235	234	234	< 1
50-0014	195	195	195	195	195	< 1
50-0015	425	425	404	402	402	< 1
50-0016	374	374	368	366	366	< 1
50-0017	439	439	434	434	434	20
50-0018	428	428	423	421	421	< 1
50-0019	393	393	376	376	376	< 1

Table 11-b. Scheduling lengths for task graphs with 50 tasks on 4 cores

Task graph IDs	Scheduling length					B&B Runtime 12 hours (sec)
	PCS	Dual-mode	GA	B&B 12 hours	B&B 1 sec	
50-0000	168	167	159	155	157	8
50-0001	220	211	203	202	202	< 1
50-0002	173	170	165	162	162	< 1
50-0003	194	194	185	181	186	114
50-0004	167	167	166	166	166	< 1
50-0005	439	426	404	397	397	< 1
50-0006	275	270	266	258	260	6
50-0007	357	354	340	339	340	X
50-0008	409	407	390	387	387	< 1
50-0009	327	356	318	314	314	3
50-0010	131	131	129	128	130	50
50-0011	181	176	170	170	170	< 1
50-0012	197	192	179	179	179	2
50-0013	186	192	182	178	178	7
50-0014	171	167	161	159	159	462
50-0015	376	373	347	345	345	< 1
50-0016	318	319	292	292	292	< 1
50-0017	377	378	365	359	362	6,800
50-0018	403	396	363	363	363	< 1
50-0019	342	330	326	323	323	< 1

Table 11-c. Scheduling lengths for task graphs with 50 tasks on 8 cores

Task graph IDs	Scheduling length					B&B Runtime 12 hours (sec)
	PCS	Dual-mode	GA	B&B 12 hours	B&B 1 sec	
50-0000	149	148	144	139	142	1,250
50-0001	203	201	186	184	184	2
50-0002	161	153	143	139	139	1
50-0003	175	180	172	165	170	7,210
50-0004	150	155	147	147	147	< 1
50-0005	432	406	385	379	379	< 1
50-0006	259	246	239	231	236	306
50-0007	336	312	305	296	300	13,700
50-0008	366	354	337	333	333	2
50-0009	323	326	296	289	291	13
50-0010	127	125	121	118	120	1,380
50-0011	180	172	161	159	159	< 1
50-0012	183	178	171	170	171	15
50-0013	171	171	160	158	161	294
50-0014	166	163	148	144	148	1,860
50-0015	304	307	290	289	289	2
50-0016	269	266	254	245	248	24
50-0017	306	314	294	286	290	X
50-0018	358	354	329	328	328	< 1
50-0019	361	365	345	343	343	6

Table 11-d. Scheduling lengths for task graphs with 50 tasks on 16 cores

Task graph IDs	Scheduling length					B&B Runtime 12 hours (sec)
	PCS	Dual-mode	GA	B&B 12 hours	B&B 1 sec	
50-0000	156	151	140	136	141	4,680
50-0001	195	198	193	192	193	2
50-0002	150	146	131	128	129	9
50-0003	169	165	158	158	159	X
50-0004	158	157	145	144	144	< 1
50-0005	406	388	373	360	360	9
50-0006	268	249	246	243	254	354
50-0007	301	279	273	260	269	X
50-0008	360	345	327	319	319	< 1
50-0009	289	292	265	260	270	149
50-0010	126	127	123	122	125	X
50-0011	135	146	129	129	129	1
50-0012	174	169	166	164	164	27
50-0013	154	155	147	144	148	251
50-0014	160	147	147	143	144	X
50-0015	325	347	318	309	309	< 1
50-0016	286	293	259	254	259	38
50-0017	333	312	310	308	308	X
50-0018	342	344	326	326	326	1
50-0019	334	336	306	299	299	5

The detailed results for task graphs with 50 tasks are shown in Table 11. The tables show not only the scheduling length but also the runtime of the branch-and-bound algorithm. The 'X' mark in the right most column means that our branch-and-bound algorithm could not find the optimal result within 12 hours. In such cases, the length of the best schedule found in 12 hours is written in the tables. For 73 test cases out of 80, our branch-and-bound algorithm successfully found optimal schedules within 12 hours. Even when optimal schedules are not found, our branch-and-bound algorithm always found better schedules than the other three algorithms.

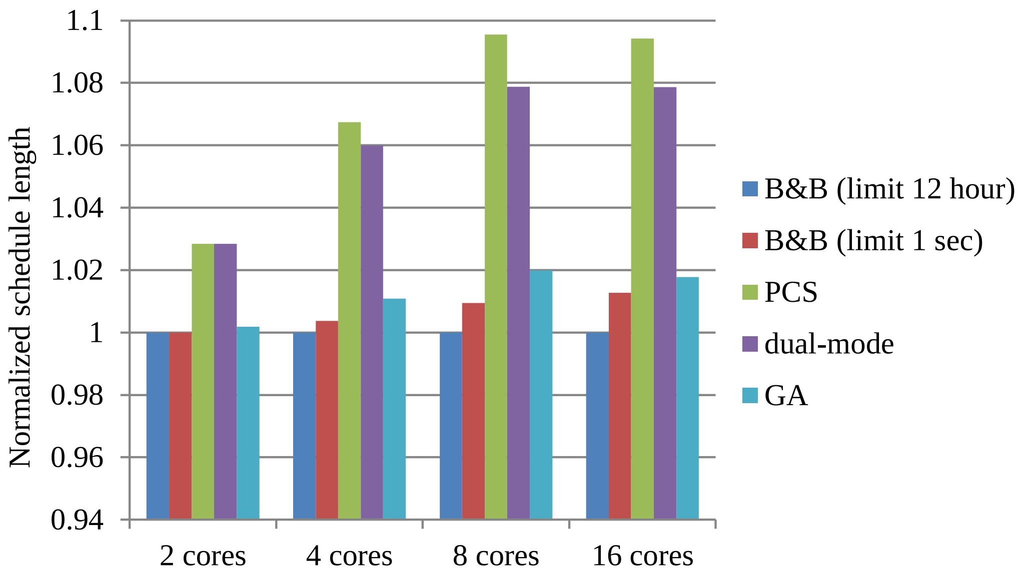


Figure 32. Average schedule length normalized by B&B for task sets with 50 tasks

Table 12-a. Scheduling lengths for task graphs with 100 tasks on 2 cores

Task graph IDs	Scheduling length					B&B Runtime 12 hours (sec)
	PCS	Dual-mode	GA	B&B 12 hours	B&B 1 sec	
100-0000	431	431	431	431	431	18
100-0001	401	401	397	396	396	< 1
100-0002	459	459	448	446	446	< 1
100-0003	406	406	391	391	391	< 1
100-0004	393	393	393	393	393	73
100-0005	814	814	780	774	774	< 1
100-0006	868	868	826	820	820	< 1
100-0007	861	861	847	845	845	7
100-0008	796	796	792	792	792	< 1
100-0009	947	947	912	910	910	< 1
100-0010	464	464	446	445	445	1
100-0011	445	445	441	440	440	227
100-0012	469	469	451	451	451	< 1
100-0013	480	480	474	472	472	< 1
100-0014	391	391	386	386	386	36
100-0015	781	781	765	763	763	< 1
100-0016	764	764	751	748	748	< 1
100-0017	860	860	857	857	857	6
100-0018	724	724	722	720	720	29
100-0019	749	749	736	736	736	< 1

Table 12-b. Scheduling lengths for task graphs with 100 tasks on 4 cores

Task graph IDs	Scheduling length					B&B Runtime 12 hours (sec)
	PCS	Dual-mode	GA	B&B 12 hours	B&B 1 sec	
100-0000	388	376	367	358	377	3,610
100-0001	348	343	340	335	340	11,500
100-0002	413	424	403	390	391	9
100-0003	341	338	336	325	330	33
100-0004	354	366	346	340	351	8,470
100-0005	704	682	666	655	655	2
100-0006	785	737	721	706	718	24
100-0007	760	735	739	711	734	9,310
100-0008	701	706	694	694	701	X
100-0009	783	779	763	747	767	89
100-0010	385	392	366	363	368	72
100-0011	394	377	371	364	367	165
100-0012	432	434	405	405	405	< 1
100-0013	404	427	394	390	393	32
100-0014	354	334	329	316	325	34
100-0015	706	683	675	650	671	102
100-0016	667	646	614	603	606	9
100-0017	746	755	740	705	720	135
100-0018	628	626	601	571	595	19,700
100-0019	700	709	661	659	659	4

Table 12-c. Scheduling lengths for task graphs with 100 tasks on 8 cores

Task graph IDs	Scheduling length					B&B Runtime 12 hours (sec)
	PCS	Dual-mode	GA	B&B 12 hours	B&B 1 sec	
100-0000	356	337	332	316	337	1,330
100-0001	326	330	326	317	319	X
100-0002	380	372	354	346	353	45
100-0003	338	342	326	320	336	211
100-0004	340	327	322	306	321	X
100-0005	713	698	666	644	644	8
100-0006	712	703	680	659	690	190
100-0007	675	657	630	622	654	X
100-0008	637	638	618	614	628	X
100-0009	785	737	710	684	712	135
100-0010	338	327	317	315	321	131
100-0011	353	349	354	346	349	X
100-0012	431	423	380	380	380	< 1
100-0013	382	385	378	363	380	1,270
100-0014	327	319	319	314	325	X
100-0015	697	646	621	621	685	X
100-0016	625	657	607	599	611	88
100-0017	730	730	696	684	709	4,750
100-0018	657	642	625	604	635	X
100-0019	679	679	646	631	640	9

Table 12-d. Scheduling lengths for task graphs with 100 tasks on 16 cores

Task graph IDs	Scheduling length					B&B Runtime 12 hours (sec)
	PCS	Dual-mode	GA	B&B 12 hours	B&B 1 sec	
100-0000	335	333	317	311	319	X
100-0001	307	304	305	302	305	X
100-0002	365	355	335	335	340	11
100-0003	314	309	298	289	306	712
100-0004	317	307	303	297	305	X
100-0005	668	676	638	623	630	14
100-0006	687	666	654	629	655	562
100-0007	665	637	622	604	638	X
100-0008	607	610	597	597	590	X
100-0009	728	713	692	663	696	394
100-0010	362	354	324	315	328	581
100-0011	336	331	310	299	315	X
100-0012	410	421	387	387	387	< 1
100-0013	375	372	369	353	372	2,060
100-0014	313	306	305	305	305	X
100-0015	606	557	541	540	565	X
100-0016	648	645	601	594	611	74
100-0017	677	692	657	632	668	23,400
100-0018	591	595	567	563	579	X
100-0019	676	672	631	631	633	5

The detailed results for task graphs with 100 tasks are shown in Table 12. For 62 test cases out of 80, our branch-and-bound algorithm successfully found optimal schedules within 12 hours.

At the same time, the B&B also helps us to evaluate other algorithms more accurately. In Figure 32 and Figure 33, each bar indicates the average scheduling length of 20 task graphs which is normalized to the B&B (limited to 12 hours). As the number of overall cores and tasks increase, PCS or dual-mode algorithm is less likely to achieve good results. For task graphs with 50 tasks, the genetic algorithm is only 0.1%, 1.0%, 1.9% and 1.7% worse, while the PCS is 2.8%, 6.7%, 9.5% and 9.4% worse than B&B on 2, 4, 8 and 16 cores respectively. For task graphs with 100 tasks, the genetic algorithm is only 0.1%, 2.2%, 2.2% and 1.3% worse, while the PCS is 2.1%, 6.8%, 7.9% and 7.4% worse than B&B on 2, 4, 8 and 16 cores respectively.

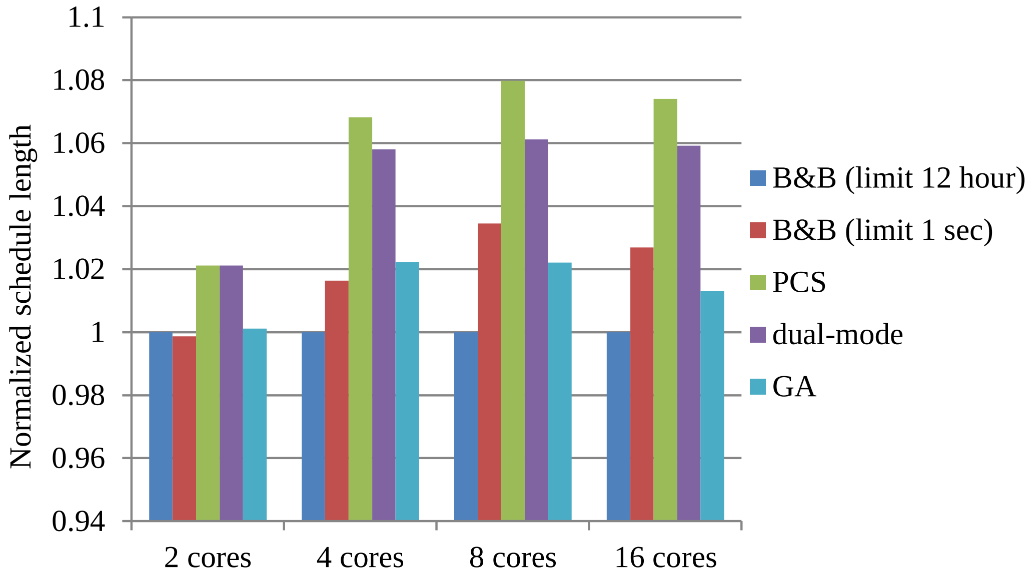


Figure 33. Average schedule length normalized by B&B for task sets with 100 tasks

Chapter 8.

Conclusions

Task scheduling is a very important problem to exploit the maximum capability of multicore processors. In order to deal with different challenges in practical use, we presented a series of different algorithms for task scheduling for data-parallel tasks on multicore architectures.

In Section 4, we proposed six algorithms base on list scheduling. The experimental results show that, among the six algorithms, the PCS algorithm yields the best scheduling results on average. Furthermore, according to the shortcomings of the PCS (using static priority), we proposed a new algorithm for task scheduling which is called dual-mode algorithm. Different from common list scheduling algorithms, the proposed algorithm has two priority types, and changes its behavior under the available cores conditions of the system. This algorithm has achieved 2%, reduction in the scheduling length on average.

For more powerful systems, we presented a genetic algorithm for the task scheduling problem which takes into account both task parallelism and data parallelism. Moreover, we proposed a new chromosome representation and corresponding genetic operators which aim to minimize the execution time and search space. We also proposed a parallelization method for the genetic algorithm. Our experiments show that the proposed genetic algorithm significantly improved the scheduling lengths over the PCS and dual-mode algorithm.

In order to deeper understand the scheduling problem and better evaluate the effectiveness of proposed algorithms. The study of finding optimal solutions for task scheduling is also indispensable. We proposed an exact algorithm for the scheduling problem with data parallelism. The proposed algorithm enumerates all possible solutions and explores them in a depth-first way. We presented four rules to prune non-optimal branches. The experiments show that our algorithm could find best schedules in a practical time for large task sets (the number of tasks is up to 100).

There are several works we plan to conduct in the future: (i) considering the cost of the communication; (ii) using CUDA to speed up our genetic algorithm further. (iii) comparing our genetic algorithm with other meta-heuristics, e.g., ACOs for task scheduling.

Considering the cost of the communication; we assume the communication cost between 2 tasks is scheduled on different cores can ignore. This assumption may not be practical in some case, e.g., task scheduling on distributed computing system. In the future, we plan to study task scheduling with communication cost.

Using CUDA to speed up our genetic algorithm further; the fewer compute units of CPU limit the parallel version of our genetic algorithm, so we hardly achieve furthermore speed up by OpenMP. Since GPU has much more compute units than CPU, CUDA is an ideal tool for our algorithm. In future, we plan to design the CUDA version of our genetic algorithm.

Comparing our genetic algorithm with other meta-heuristic; there are a large number of works for task scheduling with other meta-heuristics, e.g., ACOs. We plan to extend those methods to our problem and compare them with our genetic scheduling algorithm.

Acknowledgements

Firstly, I would like to express my heartfelt gratitude to my supervisor Professor Hiroyuki Tomiyama, His patient guidance, encouragement and advice helped me in all the time of my Ph.D. study life and writing of this thesis. He not only taught me how to do scientific research and present my ideas in a better way, but also profoundly impressed me by his modesty, carefulness and gentleness. I feel honoured to be his student, and I am sure I still have much to learn from him in the future.

I sincerely acknowledge my gratefulness to Professor Lin Meng, as both a teacher and friend. He helped me a lot in research and life. He also was the first person who encouraged me to obtain the Ph.D. I would also like to thank Professor Ittetsu Taniguchi for giving me various suggestions throughout the work.

I would like to express my special thanks to Professor Katsuhiro Yamazaki and Professor Tomonori Izumi for their kind acceptance to be examiners of my Ph.D. thesis.

Finally, I am forever grateful to my parents and wife for their endless support, understanding and love. Without my lovely family, I would not be able to finish this work.

References

[1] G. Coffman, "Computer and Job-shop Scheduling Theory," *New York*, *Wiley*, 1976.

[2] M. R. Garey and D. S. Johnson, "Computers and Intractability: A Guide to the Theory of NP Completeness, " *San Francisco*, *CA*, *W. H. Freeman*, 1979.

[3] R. L. Graham, L. E. Lawler, J. K. Lenstra and A. H. Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey", *Annuls of discrete Mathematics*, vol. 5, no. 10, pp. 287-326, December 1979.

[4] T. L. Adam, K. M. Chandy, and J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems," *Commun. Ass. Comput. Ma*ch., vol. 17, no. 12, pp. 685-690, December 1974.

[5] T. Gonzalez, O.H. Ibarra and S. Sahni, "Bounds for LPT Schedules on Uniform Processors," *SIAM Journal on Computing*, vol. 6, no. 1, pp. 155-166, 1977.

[6] Y.K. Kwok, and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors," *ACM Computing Surveys (CSUR)*, vol. 31, no.4, pp. 406-471, December 1999.

[7] M.R. Devi, and A. Anju, "Multiprocessor Scheduling of Dependent Tasks to Minimize Makespan and Reliability Cost Using NSGA-II," *International Journal in Foundations of Computer Science & Technology*, vol. 4, no. 2, pp. 27-39, March 2014.

[8] Y. Xu, K. Li, L. He, L. Zhang, and K. Li, "A Hybrid Chemical Reaction Optimization Scheme for Task Scheduling on Heterogeneous Computing Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 12, pp. 3208-3222, December 2015.

[9] Y. Xu, K. Li, L. He, L. Zhang, and K. Li, "A Genetic Algorithm for Task Scheduling on Heterogeneous Computing Systems Using Multiple Priority queues," *Information Sciences*, vol. 270, no. 20, pp. 255-287, June 2014.

[10] P. Tendulkar, P. Poplavko, I. Galanommatis, and O. Maler, "Many-Core Scheduling of Data Parallel Applications Using SMT Solvers," *Digital System Design (DSD), 17th Euromicro Conference* on, pp. 27-29, August 2014.

[11] H. El-Rewini, H. Ali and Ted Lewis, "Task Scheduling in Parallel and Distributed Systems," *New Jersey, Prentice-Hall, Inc. Upper Saddle River*, 1995.

[12] G. Sinevriotis and T. Stouraitis, "A Novel List-Scheduling Algorithm for the Low-Energy Program Execution," *IEEE International Symposium on Circuits and Systems*, pp. 26-29, May 2002.

[13] J. Brest and V. Zumer, "A Performance Evaluation of List Scheduling Heuristics for Task Graphs Without Communication Costs," *Proc. International Workshop on Parallel Processing*, pp. 21-24, August 2000.

[14] H.B. Chen, B. Shirazi, K. Kavi and A.R. Hurson, "Static Scheduling Using Linear Clustering and Task Duplication," *Proc. ISCA Int'l Conf. Parallel and Distributed Computing and Systems*, pp. 285-290, 1993.

[15] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 9, pp. 951-967, September 1994.

[16] D. Darbha and D. P. Agrawal "Optimal Scheduling Algorithm for Distributed-Memory Machines," *IEEE Trans. Parallel and Distributed Systems*, vol. 9, no. 1, pp.87-95, February 1998.

[17] C.I. Park and T.Y. Choe, "An Optimal Scheduling Algorithm Based on Task Duplication," *IEEE Trans. Computers*, vol. 51, no. 4, pp. 444-448, April 2002.

[18] L. Wang, H. Siegel, V. Roychowdhury and A. Maciejewski "Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach," *J. Parallel Distrib. Comput.*, vol. 47, no. 1, pp. 8-22, November 1997.

[19] H.M. Ghader, K. Fakhr, M. Javadi, G. Bakhshzadeh, "Static Task Graph Scheduling Using Learner Genetic Algorithm," *Int. Conf. on Soft Computing and Pattern Recognition*, pp. 7-10, December 2010.

[20] S. Gupta, G. Agarwal, and V. Kumar, "Task Scheduling in Multiprocessor System Using Genetic Algorithm," *Int. Conf. on Machine Learning and Computing*, pp. 9-11 February 2010.

[21] O. Sathappan, P. Chitra, P. Venkatesh and M. Prabhu "Modified Genetic Algorithm for Multi Objective Task Scheduling on Heterogeneous Computing System", *Int. J. Inform. Technol. Commun*, vol. 1, no. 2, pp.146-158, 2011.

[22] Y. Yi, W. Han, X. Zhao, A. T. Erdogan, and T. Arslan, "An ILP Formulation for Task Mapping and Scheduling on Multi-Core Architectures," *Int. Conf. on Design Automation and Test in Europe Conference & Exhibition*, pp. 20-24, April 2009.

[23] T. Hagras and J. Janecek. "A High Performance, Low Complexity Algorithm for Compile-Time Task Scheduling in Heterogeneous Systems," *Proc. the International Parallel and Distributed Processing Symposium*, vol. 31, no. 7, pp.653-670, July 2005.

[24] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. on Computers*, vol. 33, no. 11, pp. 1023-1029, November 1984.

[25] S. Fujita, "A Branch-and-Bound Algorithm for Solving the Multiprocessor Scheduling Problem with Improved Lower Bounding Techniques," *Computers, IEEE Transactions on*, vol. 60, no. 7, pp. 1006-1016, June 2011.

[26] J.C. Soto-Monterrubio, A. Santiago, H. J. Fraire-Huacuja, J. Frausto-Solís1 and J. David Terán-Villanueva "Branch and Bound Algorithm for the Heterogeneous Computing Scheduling Multi-Objective Problem" *International Journal of Combinatorial Optimization Problems and Informatics*, vol. 7, no. 3, pp. 7-19, December 2016.

[27] O. Sinnen, A. V. Kozlov, and A. Z. S. Shahul, "Optimal Scheduling of Task Graphs on Parallel Systems," *Proc. Ninth Int. Conf. on Parallel and Distributed Computing Applications and Technologies*, pp. 323-328, December 2008.

[28] H. Yang and S. Ha, "ILP Based Data Parallel Multi-Task Mapping/Scheduling Technique for MPSoC," *International SoC Design Conference*, pp. 134-137, December 2008.

[29] H. Yang and S. Ha, "Pipelined Data Parallel Task Mapping/Scheduling Technique for MPSoC," *Design Automation and Test in Europe*, pp. 20-24, April 2009.

[30] N. Vydyanathan, S. Krishnamoorthy, G.M. Sabin, U.V. Catalyurek, T. Kurc, P. Sadayappan, and J.H. Saltz, "An Integrated Approach to Locality-Conscious Processor Allocation and Scheduling of Mixed-Parallel Applications," *IEEE Trans. on Parallel and Distributed Systems*, vol. 20, no. 8, pp. 1158-1172 October 2008.

[31] T. Tobita and H. Kasahara, "A Standard Task Graph Set for Fair Evaluation of Multiprocessor Scheduling Algorithms," *Journal of scheduling*, vol. 5, no. 5, September 2002.

[32] H. Kasahara, H. Honda and S. Narita, "Parallel Processing of Near Fine Grain Tasks Using Static Scheduling on OSCAR," *Proc. IEEE ACM Supercomputing*, pp. 856-864, November 1990.

[33] H. Kasahra, H. Honda, A.Mogi, A. Ogura, K. Fujiwara and S. Narita, "A Multi-grain Parallelizing Compilation Scheme for OSCAR," *Proc. 4th Workshop on Languages and Compilers for Parallel Computing*, pp. 283-293, August 1991.

[34] A. Yoshida, K. Koshizuka and H. Kasahara, "Data-Localization for Fortran Macrodataflow Computation Using Partial Static Task Assignment," *Proc. 10th ACM Int. Conf. on Supercomputing*, pp. 61-68, May 1996.

[35] S. B. Hassen, H. E. Bal, and C. J. H. Jacobs, "A Task and Data-Parallel Programming Language Based on Shared Objects," *ACM Trans. Program. Lang. Syst*., vol. 20, no. 6, pp. 1131-1170, November 1998.

[36] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers," *IEEE Trans. Parallel Distrib. Syst*., vol. 8, no. 11, pp. 1098-1116, November 1997.

[37] A. R adulescu, C. Nicolescu, A. J. C. van Gemund, and P. P. Jonker, " CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems." *International Parallel and Distributed Processing Symposium*, pp. 23-27, April 2000.

[38] M. Dorigo, "Optimization, Learning and Natural Algorithms", *Ph.D. Thesis, Politecnico di Milano, Italy*, 1992.

[39] F. Ferrandi, P.-L.Lanzi, C.Pilato, D. Sciuto and A. Tumeo. "Ant Colony Heuristic for Mapping and Scheduling Tasks and Communications on Heterogeneous Embedded Systems." *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.*, vol. 29, no 6, pp. 911-924, June 2010.

[40] Li, K., Xu, G., Zhao, G., Dong, Y. and Wang, D. "Cloud Task Scheduling Based on Load Balancing Ant Colony Optimization." *In Chinagrid Conference (ChinaGrid), Sixth Annual. IEEE*, pp. 3-9, 2011,

[41] A. J. Page and T.J. Naughton, "Dynamic Task Scheduling Using Genetic Algorithms for Heterogeneous Distributed Computing," *19th International Parallel and Distributed Processing Symposium*, pp.189-197, April 2005.

[42] E.S. H. Hou, N. Ansari and H. Ren, "A Genetic Algorithm For Multiprocessor Scheduling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5 no. 2, pp. 113-120, February 1994.

[43] F. Omara and M. Arafa, "Genetic Algorithms for Task Scheduling Problem," *Journal of Parallel and Distributed Computing*, vol. 70, no. 1, pp.13-22, January 2010.

[44] P. Roy, M. Mejbah and N. Das. "Heuristic Based Task Scheduling in Multiprocessor Systems with Genetic Algorithm by Choosing the Eligible Processor," *International Journal of Distributed and Parallel Systems (IJDPS)*, vol. 3, no.4, pp. 111-121, August 2012.

[45] R. Entezari-Maleki and A. Movaghar, "A Genetic-Based Scheduling Algorithm to Minimize the Makespan of the Grid Applications," *International Journal of Grid and Distributed Computing*, vol. 4, no. 2, pp. 11-24, June 2011.

[46] S. Venugopalan and O. Sinnen, "ILP Formulations for Optimal Task Scheduling with Communication Delays on Parallel Systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, January 2015.

[47] R. Dechter and J. Pearl, "Generalized Best-First Search Strategies and the Optimality of A*," *Journal of the ACM (JACM)*, vol.32, no.3, pp. 505-536, July 1985.

[48] A.Z. Semar Shahul and O. Sinnen, "Scheduling Task Graphs Optimally with A*," *The Journal of Supercomputing*, vol.51 no.3, pp. 310-332, March 2010.

[49] Z. Yang, B. Yu and C. T. Cheng, "A Parallel Ant Colony Algorithm for Bus Network Optimization," *Computer-Aided Civil and Infrastructure Engineering*, vol.22, no.1, pp. 44-55, January 2007.

[50] B. J.Vitins, and K. W. Axhausen, "Optimization of Large Transport Networks Using the Ant Colony Heuristic," *Computer-Aided Civil and Infrastructure Engineering*, vol.24, no.1, pp. 1-14, January 2009.

[51] M. Dorigo, "Ant Colonies for the Travelling Salesman Problem" *BioSystems*, vol.43, no. 2, pp. 73-81, July 1997.

[52] J. Bai, GK Yang, Y.W. Chen, LS Hu and C.C. Pan," A Model Induced Max-Min Ant Colony Optimization for Asymmetric Traveling Salesman Problem," *Applied Soft Computing*, vol.13, no. 3, pp. 1365-1375, March 2013.

[53] L.M. Gambardella and É.D. Taillard," Ant Colonies for the Quadratic Assignment Problem," *Future Generation Computer Systems*, vol.17, no. 4, pp. 441-449, January 2001.

[54] J. H. Holland "Genetic Algorithms," *Scientific American*, vol. 267, pp. 66-72, July 1992.

[55] http://www.kasahara.elec.waseda.ac.jp/schedule/
(Last accessed: April, 2018)

Publications

Journal Publications

- 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, pp. 279-290, July 2014.
- Yining Xu, Yang Liu, Junya Kaida, Ittetsu Taniguchi, and Hiroyuki Tomiyama, "Static Mapping of Multiple Parallel Applications on Non-Hierarchical Manycore Embedded Systems," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E99-A, no. 7, pp. 1417-1419, July 2016.
- Y. Liu, L. Meng, I. Taniguchi, and H. Tomiyama, "A Dual-Mode Scheduling Approach for Task Graphs with Data Parallelism," *International Journal of Embedded Systems,* Inderscience Publishers, vol. 9, no. 2, pp. 147-156, April 2017.
- Y. Liu, L. Meng, I. Taniguchi, and H. Tomiyama, "A Branch-and-Bound Approach to Scheduling of Data-Parallel Tasks on Multicore Architectures," Accepted for publication in *International Journal of Embedded Systems*, Inderscience Publishers.

International Conference Publications

- Y. Liu, I. Taniguchi, H. Tomiyama, and L. Meng, "List Scheduling Strategies for Task Graphs with Data Parallelism," In *Proc. of International Symposium on Computing and Networking (CANDAR),* pp. 168-172, Matsuyama, Dec. 2013.
- Y. Liu, L. Meng, I. Taniguchi, and H. Tomiyama, "A Dual-Mode Scheduling Algorithm for Task Graphs with Data Parallelism," In *Proc. of Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 371-374, Ishigaki, Japan, November 2014.
- Yining Xu, Junya Kaida, Yang Liu, Ittetsu Taniguchi, Hiroyuki Tomiyama, "Static Task Mapping for Non-Hierarchical Manycore SoCs," In *Proc. of International Technical Conference on Circuits/Systems, Computers and Communications (ITC-*

CSCC), pp. 519-521, Seoul, Korea, June-July 2015.

- Y. Liu, L. Meng, I. Taniguchi, and H. Tomiyama, "A Branch-and-Bound Algorithm for Scheduling of Data-Parallel Tasks," In *Proc. of Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI)*, pp. 96-100, Kyoto, October 2016.
- Y. Liu, L. Meng, H. Tomiyama, "A Genetic Algorithm for Scheduling of Data-Parallel Tasks," *International Symposium on Advanced Technologies and Applications in the Internet of Things (ATAIT)*, Osaka, April 2018.

Domestic Workshops and Meetings

- Yang Liu, Ittetsu Taniguchi, Hiroyuki Tomiyama and Lin Meng, "List Scheduling Algorithms for Task Graphs with Data Parallelism," 電子情報通信学会 VLD/DC/情報処理学会 SLDM 研究会, 鹿児島, 2013 年 11 月.
- Yang Liu, Ittetsu Taniguchi, Hiroyuki Tomiyama and Lin Meng, "List Scheduling Strategies for Task Graphs with Data Parallelism," 第 14 回留日中国人研究成果報告会論文集, pp. 265-268, 大阪, 2013 年 11 月.
- Yang Liu, Lin Meng, Ittetsu Taniguchi and Hiroyuki Tomiyama, "A Dual-Mode Scheduling Strategy for Task Graphs with Data Parallelism," 電子情報通信学会 VLD/CPSY/RECONF/情報処理学会 SLDM 研究会, 横浜, 2015 年 1 月.
- Yang Liu, Yining Xu, Lin Meng, Ittetsu Taniguchi and Hiroyuki Tomiyama, "A Fast and Exact Algorithm for Scheduling of Data-Parallel Tasks," 電子情報通信学会総合大会, 草津, 2015 年 3 月.