Doctoral Thesis reviewed

by Ritsumeikan University

Scheduling Algorithms for Data-Parallel Tasks on Multicore Architectures

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

Yang Liu

劉　陽

Supervisor: Professor Hiroyuki Tomiyama

研究指導教員：冨山　宏之　教授

# Abstract

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

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

In this thesis, we provide an extensive survey of existing algorithms. Since the scheduling problem is NP-hard, there are a large number of heuristics and meta-heuristics aim to find near-optimal results. List scheduling is one of the most important 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 traditional priority strategy to task scheduling for data-parallel tasks. We propose six list scheduling algorithms with different strategy of priority assignment. The six algorithms, as well as an integer linear programming method, are evaluated.

We also find 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 algorithm. The experimental results show that the proposed algorithm yields shorter scheduling length than pure list scheduling.

The advantages of list scheduling algorithms and its variants are producing results in a very short time and relatively simple to implement. However, their acquired scheduling results are far from optimal. In recent years, many studies have turned to meta-heuristics to solve task scheduling problem. Meta-heuristics provide some algorithmic frameworks to search 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-heuristic) is proposed to solve task scheduling which considering both task- and data-parallelism. Different from traditional genetic algorithms for task scheduling, we also 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 based algorithms.

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 presented 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 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 algorithm found better schedules than state-of-the-art heuristic algorithms.

Contents

[1. Introduction 1](#_Toc514525119)

[2. Related Work 5](#_Toc514525120)

[2.1. Heuristic Algorithms 6](#_Toc514525121)

[2.2. Meta-Heuristic Algorithms 7](#_Toc514525125)

[2.3. Exact Algorithms 9](#_Toc514525128)

[2.4. Algorithm for Data-Parallel Task 11](#_Toc514525132)

[3. The Problem Definition 12](#_Toc514525133)

[3.1. Task Graph 12](#_Toc514525134)

[3.2. System Model 13](#_Toc514525135)

[3.3. ILP Formulation 15](#_Toc514525136)

[4. List Scheduling Algorithms 17](#_Toc514525137)

[4.1. A Motivating Example 18](#_Toc514525138)

[4.2. The Proposed Priorities 21](#_Toc514525139)

[4.3. Experiments 22](#_Toc514525140)

[4.3.1. Results for Random Task Graphs 23](#_Toc514525141)

[4.3.2. Results for Realistic Task Graphs 34](#_Toc514525142)

[5. Dual-mode algorithm 38](#_Toc514525143)

[5.1. The Problem of Pure List-Scheduling 38](#_Toc514525144)

[5.2. A Motivating Example 40](#_Toc514525145)

[5.3. The Overall Dual-mode Scheduling Algorithm 42](#_Toc514525146)

[5.4. Experiments 48](#_Toc514525147)

[5.4.1. Results for Random Task Graphs 48](#_Toc514525148)

[5.4.2. Results for Realistic Task Graphs 52](#_Toc514525149)

[6. Genetic Algorithm 55](#_Toc514525150)

[6.1. Genetic Algorithm Fundamentals 56](#_Toc514525151)

[6.2. The Proposed Genetic Algorithm 58](#_Toc514525152)

[6.2.1. Representation of a Chromosome 58](#_Toc514525153)

[6.2.2. Initialization 59](#_Toc514525154)

[6.2.3. Fitness Function 61](#_Toc514525155)

[6.2.4. Selection 62](#_Toc514525156)

[6.2.5. Crossover 63](#_Toc514525157)

[6.2.6. Mutation 64](#_Toc514525158)

[6.2.7. Parallelization of the Algorithm with OpenMP 66](#_Toc514525159)

[6.3. Experiments 67](#_Toc514525160)

[7. Branch-and-Bound Algorithm 75](#_Toc514525161)

[7.1. Depth-First Search 75](#_Toc514525162)

[7.2. Branch-and-Bound Methods 78](#_Toc514525163)

[7.2.1. Related Pattern Rule 78](#_Toc514525164)

[7.2.2. Exclusive Task Branch Rule 79](#_Toc514525165)

[7.2.3. Reducing Meaningless Idle Time 80](#_Toc514525166)

[7.2.4. Lower Bound Rule 82](#_Toc514525167)

[7.3. Selection Rule 83](#_Toc514525168)

[7.4. Experiments 84](#_Toc514525169)

[8. Conclusions 95](#_Toc514525170)

[Acknowledgements 96](#_Toc514525171)

[References 97](#_Toc514525172)

[Achievements 108](#_Toc514525173)

[Journal 108](#_Toc514525174)

[International Conference 108](#_Toc514525175)

# 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 fully 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]. In many cases of embedded system design where characteristics of the tasks are known at the design time, static scheduling is often preferred due to its low runtime overhead and high predictability. This thesis focuses on static scheduling. Given a set of tasks with data dependency, a task scheduling problem decides when and on which core each task is executed in such a way that the overall schedule length is minimized, while meeting constraints on flow dependency among tasks and the number of available cores.

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 the thesis survey different types of task scheduling algorithms and discuss their advantages and disadvantages, as well as the scope of application.

Next, we present the definition of task scheduling problem with data-parallel tasks and some necessary notations used in scheduling theory.

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.

In the following chapters, we roughly divided existing algorithms for task scheduling problem into exact algorithms, heuristics, and meta-heuristics. Exact algorithms guarantee to find the optimal solution. However, for most complex scheduling problems, searching the optimal solutions require 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 heuristic and meta-heuristic is that heuristic is a problem-specific method. However, meta-heuristic is a framework that provides a set of guidelines or strategies to develop heuristic algorithms. The popular meta-heuristics including: the genetic algorithms (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 tasks, and schedules those tasks by the assigned priorities. We also extended the existing strategy 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 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 based 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 schedule results than pure list scheduling algorithms.

Although the algorithms based on list scheduling are proposed in chapter 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 more efficient, and escape from the local optimal solution. 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 ordering of task execution, does not represent which cores are mapped by those 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 the calculation time is much longer than the list scheduling based algorithm, since the computation is inherently parallel, we can 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 based algorithms.

As we mentioned earlier, finding the optimal solution is very time-consuming especially for complex 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 algorithm for this problem. 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 chapter 4, 5 and 6.

# Related Work

The task scheduling problem has been extensively studied for decades. From Table 1 we know that scheduling problems tasks with arbitrary execution time and arbitrary precedence constraints are known to be NP-hard [1] [2] and [3]. Due to the nature of NP-hard, pioneering researchers were proposed many heuristics and meta-heuristics, and try 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 T*i* | Precedence Constraints | Complexity |
| Arbitrary | Equal | Tree | O(*n*) |
| 2 | Equal | Arbitrary | O(*n2*) |
| Arbitrary | Equal | Arbitrary | NP-hard |
| Fixed (*m*>=2) | T*i*=1or2 for all *i* | Arbitrary | NP-hard |
| Arbitrary | Arbitrary | Arbitrary | Strong NP-hard |

## 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 Heuristic 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 make a scheduling list (a sequence of tasks for scheduling) by assigning them certain priorities, and then allocate the task with the highest priority to free cores 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] is 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 of 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], and [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, aiming to reduce or optimize the task starting or finishing time. The main weakness of duplication-based algorithms is their high complexity and that they mainly target 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 while cores to duplicate the task.

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

## 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](http://scholarpedia.org/article/Algorithm) [54]. Most meta-heuristic algorithms are nature-inspired as they have been designed based on some abstraction of nature. The most popular meta-heuristics including: genetic algorithms (GA), ant colony optimization (ACO), bee algorithms (BA), particle swarm optimization (PSO) and simulated annealing (SA).

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

### Genetic Algorithms (GA)

The genetic algorithm was first invented by Holland [1]. This algorithm thinks of a set of candidate solutions for a problem as biological population, in each step. genetic algorithm also eliminates the bad individuals, and the good ones are more likely selected and produce next generations. Over successive generations, more and better individuals will be found.

In the past decade, GA has been widely used to solve task scheduling problem. GAs 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 task scheduling problem and corresponding genetic operators usually are the key issues for genetic algorithm design.

### Ant Colony Optimization (ACO)

Ant colony optimization (ACO) is another popular meta-heuristic algorithm for combinational problems, it is a meta-heuristic 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 home. While the pheromones will attract other ants, as the time goes on, a better (shortest) path will be found.

Although compared with the genetic algorithm, Ant colony optimization is relatively new, it is initially proposed in [38]. 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 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 by pheromones left by previous ants.
2. Evaluate the obtained scheduling solutions.
3. The pheromones on the paths of achieving better solutions will be strengthened.
4. Go to step 1.

## Exact Algorithms

Although most of the studies 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. In this chapter, I will introduce several methods for scheduling problem.

### Branch-and-Bound Algorithms

The branch-and-bound algorithm (B&B) is the most frequently used method for task scheduling problem (i.e. [24] [25] [26] [27]). B&B explores all solution space which is represented as a branching tree, and prunes the branches which are not candidates to improve the final results during the search. DF/IHS (depth first/implicit heuristic search) [24] is one of task scheduling based on B&B. This method can reduce space complexity markedly and average computation time 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

The scheduling constraints in task scheduling problem can be represented as a set of integer constraint functions. Algorithms based on integer linear programming (ILP) try to minimize or maximize an object function while satisfying all the constraints. When only some of the variables are integer, the problem is called a mixed-integer linear program.

Recently, Venugopalan [46] has proposed ILP based approach which aims to find exact resolution 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 using A\* searching algorithm for solving the problem of task scheduling. A\* algorithm based task scheduling 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. Until all the tasks are scheduled.

Usually, the cost is defined as:

(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.

## Algorithm for Data-Parallel Task

Unfortunately, the majority of the works on task scheduling (The above mentioned algorithms) only consider task parallelism only. 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 exist several research efforts on task scheduling with data parallelism in the past. Recent studies include [28], [29] and [30]. In [28], Yang and Ha proposed a scheduling technique for data-parallel tasks based on integer linear programming (ILP) formulation, and extended the technique towards pipelined scheduling in [29]. Their techniques perform task scheduling and allocation simultaneously, where allocation means a design process which decides the number of cores assigned to each task. Vydyanathan also proposed a simultaneous scheduling and allocation algorithm for data-parallel tasks [30]. The common assumption in [28], [29] and [30] is that the degree of data parallelism in tasks, i.e., the number of cores assigned to the task, is flexible, and the execution time of the task for each parallelism is known prior to task scheduling decision. However, this assumption may not be practical in some cases. This thesis assumes that a task has a fixed degree of data parallelism. We use a variety of popular optimization algorithms to solve this problem and compare them comprehensively with their quality of results and runtime.

# 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 schedule length (makespan) 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.

## Task Graph

An application is modeled as an acyclic directed graph (DAG), also be called task graph, where a node represents a task and a directed edge represents a flow dependency between the two tasks. Figure 1 shows an example of a task graph. In this graph, tasks labeled S and E are dummy tasks which do not perform any meaningful computation. Tasks S and E denote an entry point and an exit point of the application. Tasks are associated with two numbers, the first number denote the degree of data parallelism of the tasks. The second one denotes the execution time of the task. For example, task 1 must assign to four cores, and take 10 time units to perform its work.

In this paper, we assume that individual tasks are written in a parallel programming language by human programmers, and that the programmers decide the degree of data parallelism. How to decide the degree of parallelism and how to know the execution time are up to the programmers, and are out of the scope of this paper.



Figure 1. A task graph

## 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.

C:\Users\scomup\AppData\Local\Microsoft\Windows\INetCache\Content.Word\c.emf

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

Figure 2 shows some 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 schedule length, and meet all constraints which are described in this task graph.

## ILP Formulation

The task scheduling is essentially a kind of mathematical optimization problem. It can be formulated by an integer linear programming (ILP) [22].

Let timei starti, and finishi denote the execution time, start time and finish time of task i, respectively. pari denotes the data parallelism, meaning that task i must be mapped onto pari cores. Flowi1,i2 denotes a flow dependency between tasks i1 and i2. Flowi1,i2 is 1 if task i1 must proceed task i2. Mapi,j denotes mapping of tasks on cores. Mapi,j is 1 if task i is mapped to core j.Then, the task scheduling problem is formally defined as follows: Given timei, pari and Flowi1,i2, decide starti, finishi and mapi,j which minimize the objective function (2), while meeting the constraints (3), (4), (5) and (6).

Minimize:

(2)

Subject to:

(3)

(4)

(5)

(6)

Optimal scheduling results can be obtained by solving the above ILP formulas, but it is not practical for large task sets.

# List Scheduling Algorithms

List scheduling is a most important task scheduling heuristic. 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.

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 execute the following two steps until a valid schedule is obtained:

1. Update the *Readylist* , and select a task from *Readylist*.
2. Assign this task to suitable processors.

In step (1), the task with the highest priority will be selected first. How to define the priority of tasks is the most important issue for the design of list scheduling. There are a large number of variations of list scheduling depending on how to define the priority. The list scheduling described in above also summarized in Figure 3.



Figure 3. Flowchart for list scheduling algorithm

## A Motivating Example

In [19], Kasahara and Narita propose a list-based scheduling algorithm, named CP/MISF (critical path/most immediate successor first). The CP/MISF algorithm is designed for task scheduling without data parallelism. Although it was proposed three decades ago, it is still recognized as one of the best heuristic algorithms because of the high quality of results as well as the low computational complexity. As the name of the algorithm indicates, the CP/MISF algorithm takes into account two factors to define the priority of tasks; the critical path length and the number of immediate successors. The critical path length of a task is the length of the longest path from the node to the exist node. For example, the critical path length of task 2 is 60, by adding the execution time of task 2 and that of task 5. In the CP/MISF algorithm, the priority of tasks is defined according to the following two rules:



Figure 4. Schedule obtained by the CP/MISF algorithm



Figure 5.Schedule which takes into account the 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.

In case tasks i and j has the same critical path length, if task i has more immediate successors than task j, task i has a higher priority than task j.

Figure 4 shows the schedule when the CP/MISF algorithm is applied to the task graph in Figure 1. At time t=0, tasks 1 and 2 are executable, but task 2 is scheduled first because it has a longer critical path. Then, tasks 5 and 4 are scheduled, followed by tasks 1 and 3. The total schedule length is 80 time units.

The CP/MISF algorithm works nice for tasks without data parallelism. However, the CP/MISF algorithm is not always efficient for tasks with data parallelism. Actually, the schedule in Figure 4 is not optimal. Figure 5 shows a better schedule for the same task set. The policy of this scheduling is that a task with the largest data parallelism has a priority. Due to this policy, task 1 is scheduled first, and then, task 3 is enabled to run in parallel with another task. Of course, this policy is not always optimal, but this example demonstrates that the degree of data parallelism should be taken into account in the priority.

## The Proposed Priorities

We propose six algorithms, all of which are based on list scheduling, but their priority strategies are different. In order to define the priority, we take into account three factors as follows:

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

Based on the three factors, the first algorithm proposed in this paper defines the priority of tasks as follows:

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.

The algorithm based on the above priority strategy is named PCS, since the three factors (P, C and S) are prioritized in the order of P-C-S. Let *PriorityPCSi* denote the priority of task i in the PCS algorithm, where a higher value means a higher priority. A simple formula to define *PriorityPCSi* is as follows.

(7)

Here, Pi, Ci, and Si denote the values of P, C and S factors for task i, and U is a constant integer number which is larger than any of Pi, Ci, and Si for any i. In the similar manner, we can define five algorithms CPS, CSP, SCP, PSC and SPC with different ordering of the three factors. The task priorities in the five algorithms are defined as follows:

(8)

(9)

(10)

(11)

(12)

A common important feature in the six algorithms is that priorities are static. The priorities can be computed prior to scheduling, and they do not change during scheduling.

## Experiments

We implemented the six algorithms in the C language, and tested their effectiveness. We used 40 task graphs from Standard Task Graph (STG) Set [31] [55] developed at Waseda University. Since tasks in STG do not assume data parallelism, we randomly assigned the degree of data parallelism to the tasks. The number of cores was changed from two to sixteen. In addition to the six algorithms presented in this paper, an integer linear programming (ILP) technique (as defined in section 2.2) was evaluated. In order to solve the ILP problems, IBM ILOG CPLEX 12.5 was used. Since exact solutions could not be found in a practical time, we limited the CPU time of CPLEX up to 60 minutes on dual Xeon processors (E5-2650, 2.00Hz, 128GB memory), and the best solution found at that time was compared with the six algorithms.

### Results for Random Task Graphs

First, we conducted experiments using 20 random task graphs, each task graph consists 50 tasks. Figure 6 shows the average schedule lengths of the 20 task graphs obtained by the six algorithms proposed in this paper. The schedule lengths are normalized to the PCS algorithm. This graph clearly shows the effectiveness of the PCS algorithm.

Figure 6 shows detailed results for individual task graphs. The first column shows the task ID, and the following columns show the lengths of the schedules obtained by the seven methods (the six algorithms proposed in this paper and the ILP method). For each benchmark, the best solution is shaded in yellow. X in the ILP column means that no feasible solution was found within 60 minutes in CPU time. In many cases, the ILP method failed to find a feasible schedule within the limited time. Even when the ILP method found feasible schedules, they are lengthy. Although the PCS algorithm yields the best schedule results on average, Figure 6 also shows that the effectiveness of the six algorithms highly depends on the task graph，which means an algorithm is hard to work well for all task graphs.



Figure 6. Averages of normalized schedule lengths for task graphs   
with 50 tasks.

Table 2-a. Schedule 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. Schedule 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. Schedule 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. Schedule 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 |

Next, we conducted experiments using 20 random task graphs, each of which consists of 100 tasks. Figure 7 shows the average schedule lengths of the 20 task graphs obtained by the six algorithms proposed in this paper. Again, this graph clearly shows the effectiveness of the PCS algorithm.

Table 4 shows detailed results for individual task graphs with 100 tasks.



Figure 7: Averages of normalized schedule lengths for task graphs  
with 100 tasks.

Table 3-a. Schedule 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 4-b. Schedule 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. Schedule 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. Schedule 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 |

### Results for Realistic Task Graphs

In addition to the random task graphs, we used three task graphs which are derived from realistic applications. The STG contains three task graphs based on realistic application programs, 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 5 shows the average schedule lengths for the three realistic task graphs. In order to understand more easily, we normalized the all results by the result of PCS, and converted the data to bar charts as Figure 8 (a), (b) and (c). We found the PCS algorithm yields good schedules in general. However, for robot on 8 cores and sparse on 2 cores, some others algorithms perform better than PCS.

Table 5. Schedule 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 |



(a) robot



(b) sparse



(c) fpppp

Figure 8: Normalized schedule lengths for realistic task graphs.

# 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 which are switched during task scheduling. Generally, this kind of flexible priority strategy helps the new algorithm to achieve better scheduling length for more task graphs. In our experiments, the experimental results show that the proposed algorithm yields the better scheduling results than our previous algorithm.

## The Problem of Pure List-Scheduling

In our chapter 4, six list scheduling algorithms were proposed to solve task scheduling problem with data parallelism. List scheduling algorithms use a simple way to assign tasks. A list contains a set of executable tasks whose preceding tasks are completed. List scheduling chooses a task in the list which has the highest priority, and allocates the task to available cores. The approach continues until all the tasks are scheduled.

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 fully utilize multi-cores. 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 schedule length longer.

According to the above theory, scheduling task with longer critical path obtains good results in some case. Table 2 and Table 4 shows that CPS or CSP is better than PCS in some cases. To solve the problem, we designed an algorithm which has the advantage of PCS and CSP/CPS simultaneously, named dual-mode scheduling algorithm.

## A Motivating Example

Below is a simple example to illustrate that the PCS algorithm does not yield good schedules in some certain cases. Figure 10 shows the schedule when the PCS algorithm is applied to the task graph Figure 9, and a multi-core processor which has 4 cores is available.



Figure 9. Task graph

In Figure 9, as more understandable way, we have added two numbers to each task to denote the critical path length and the number of immediate successors. Task 1 and task 2 can be executed at the beginning of the scheduling process. Because task 1 has a higher degree of data parallelism than task 2 does, task 1 is scheduled first, and task 2 is scheduled in the next step. According to this schedule policy, task 3 is scheduled follow by task 2. Tasks 5 and task 4 are scheduled at last. The overall schedule length is 80 time units.

Although scheduling task 1 first can get the maximum level to use CPU idle time at the beginning, the algorithm loses the opportunity to execute task 1 with the other tasks simultaneously.

Next, assume that task 2 is scheduled first. After that, task 4 and task 5 are activated. Task 1 and task 4 can be executed at the same time. The optimal scheduling result is shown in Figure 11, and it shortens the total schedule length to 60 time units.



Figure 10. Schedule obtained by the PCS algorithm.



Figure 11. The optimal scheduling result.

## The Overall Dual-mode Scheduling Algorithm

In the common list scheduling algorithms like the PCS, each task has only one priority type. Those algorithms always try to execute tasks which have the highest priority. Tasks in *ReadyList* are sorted in order of the priority, and the *ReadyList* contains a set of schedulable tasks. A task which is called schedulable means its preceding tasks are completed, and the system has enough idle cores to allocate this task. The list scheduling algorithm allocates the task to the available idle cores. The algorithm continues until all tasks are scheduled.

In this chapter, the proposed algorithm is also based on a variation of list scheduling. Different from the common list scheduling algorithms, it prepares two ready-list types. The ready-list types include ReadyList1 and *ReadyList2*. *ReadyList2* is same as *ReadyList* in the common list scheduling algorithm. But ReadyList1 has an additional restriction, i.e., tasks in *ReadyList1* must be with the degree of data parallelism between **R**\**IdleCores* and *IdleCores*. The *IdleCores* denotes how many cores can be used at this moment, and **R** (filled ratio) is a float number ranged from 0 to 1. The two *ReadyLists* have different task-priority.

* Tasks in ReadyList1 are sorted in order of PCS priority
* Tasks in ReadyList2 are sorted in order of CS priority.

As the naming convention motioned above, CS 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.

Also we can formal defined CS as:

(13)

Below is a fundamental algorithm of dual-mode algorithm.

1. Initialise ReadyList1, ReadyList2 and IdleCores, as will as calculate the two priorities for all tasks.

* ReadyList1 = Ø •
* ReadyList2 = Ø •
* IdleCores = the number of total cores.

1. Select a task at ReadyList1 with has highest PCS priority.
2. If ReadyList1 does not contain any task. Select a task at ReadyList2 with has highest CS priority.
3. Finish if all tasks have been scheduled. Otherwise, update ReadyList1, ReadyList2 and IdleCores after go back to step 2.

The parameter **R** determines what kinds of tasks are allowed in *ReadyList1*, and changes the behavior of dual-mode algorithm. For example, when **R** = 0, *ReadyList1* contains all tasks which can be scheduled. Therefore, dual-mode algorithm would never go to step 3 and work as same as PCS does. By contrast, when **R** = 1, *ReadyList1* only contains the tasks which utilize the multi-core processor completely. In other words, dual-mode algorithm tries to find a task which occupied all idle cores. If failed, dual-mode algorithm schedule tasks according to CS priority. The dual-mode scheduling described in above also summarized in Figure 12.



Figure 12. Flowchart for Dual-mode Scheduling Algorithm

To summarize, dual-mode algorithm always tries to find a task to fully utilize the available cores. The parameter *R* indicates the lowest resource usage. This process is called MODE1. A more detailed description is available in the Listing 2. If the dual-mode algorithm cannot find qualified tasks in MODE1, it will shift to MODE2, and try to find a task which has the longest critical path. 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(N2), where N is the number of tasks, assuming that the number of cores is constant. First, it takes O(N2) to compute the critical path lengths of the nodes. After that, we use three factors (P, C and S) to calculate the two types of priority. It takes O(N) time to run. Each mode of dual-mode algorithm takes O(N) to update the ready list and chooses the best task. The main part of dual-mode algorithm shown in Listing 1 is repeated N times to schedule all the tasks. Therefore, we get the overall complexity of O(N2).

Listing 1.Dual-algorithm

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | 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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **for** i = 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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **for** i = 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 ReadyList1 with highest priority\_CS |

**Listing 4.** INCREASE\_IDLE\_CORE

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | t = the second smallest occupied times of all cores;  idle\_cores\_number : = 0;  **for** i = 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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | t = the smallest occupied times of all cores;  p = 0;  **for** i = 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** |

To summarize, the dual-mode algorithm always tries to find a task to utilize the available cores fully. The parameter *R* indicates the lowest resource usage. If the dual-mode algorithm cannot find any qualified tasks, it will shift to another mode, and try to find a task which has the longest critical path. The time complexity of the dual-mode algorithm is O(N2), where N is the number of tasks, assuming that the number of cores is constant. First, it takes O(N2) to compute the critical path lengths of the nodes. After that, we use three factors (P, C and S) to calculate the two types of priority. It takes O(N) time to run. Each mode of the dual-mode algorithm takes O (N) to update the ready list and choose the best candidate. The main part of the dual-mode algorithm is repeated N times to schedule all tasks. Therefore, we get the overall time complexity of O(N2)

## Experiments

We implemented the dual-mode algorithm in C. To test the effectiveness, we compared it with the PCS algorithm. We used 40 task graphs from Standard Task Graph (STG) Set. Out of 40 task graphs, 20 graphs contain 50 tasks, and another 20 graphs contain 100 tasks. Since tasks in STG do not assume data parallelism, we randomly assigned the degree of data parallelism to the tasks. The number of cores was changed from two to sixteen.

### Results for Random Task Graphs

We conducted experiments using 20 random task graphs. Each task graph consists of 50 tasks. Figure 13 shows the average schedule lengths of the 20 task graphs obtained by the dual-mode scheduling algorithms. In dual-mode scheduling algorithm, the *R* changes from 0.7 to 1. In order to make the results obtained by dual-mode algorithm easier to compare with the PCS algorithm, we normalized the all results by the result of PCS.

Figure 14 shows similar experiments with Figure 13. The experiments using 20 random task graphs which consist of 100 tasks. The two figures clearly demonstrate that the dual-mode algorithm improves the effectiveness of the algorithm significantly in comparison with PCS.



Figure 13. Averages of normalized schedule lengths for task graphs with 50 tasks.

However, we found the best R value is different in difference task sets. In order to determine the best R value for most situations, we designed an experiment and increase the R value between 0 to 1 by 0.05 increments. In Figure 15, we scheduled 20 task graphs which contain 50 tasks by dual-mode algorithm, and the R is changed between 0 and 1 by 0.05 increments. In Figure 16, we did similar experiments, but experiments use 20 task graphs which consist of 100 tasks.



Figure 14. Averages of normalized schedule lengths for task graphs with 100 tasks.

In Figure 15 and Figure 16, when *R* is very small, dual-mode algorithm got the same experimental results obtained by the PCS algorithm. A small *R* makes MODE1 more inclusive. Most tasks are chosen by MODE1, and dual-mode algorithm hardly shifts to MODE2. In this case, dual-mode algorithm behaves similarly to the PCS algorithm.

When a larger *R* is given, MODE1 only chooses tasks that can fully utilize idle cores. If MODE1 fail to get any task, algorithm shifts to MODE2, and a task with longest critical path will be chosen. Dual-mode algorithm behaved differently from PCS, tasks with shorter critical path and larger scale data-parallel is selected. We can get a conclusion from Figure 15 and Figure 16, the results of dual-mode improved up from a maximum of four percent than PCS especially with specific value of *R*.



Figure 15. Schedule lengths with 50 tasks (R=0~1)



Figure 16. Schedule lengths with 100 tasks (R=0~1)

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 or CS strategy. The experimental results show when R nearby 0.85 most task graphs will get good results.

### Results for Realistic Task Graphs

In addition to the random task graphs, we used three task graphs which are derived from realistic applications. The benchmarks are employed are same as section 4.3.2 (The task graphs of fpppp, robot and sparse contain 334 tasks, 88 tasks, and 96 tasks).

Figure 17 shows the average schedule lengths for the three realistic task graphs respectively. Same as Figure 13 and Figure 14, we normalized all the results by the result of PCS, and converted the data to bar charts as Figure 17 (a) to (c). We found the dual-mode algorithm yields 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.

1. robot
2. sparse
3. fpppp

Figure 17: Normalized schedule lengths for realistic task graphs.

# Genetic Algorithm

In chapter 5, dual-mode algorithm has greatly improved the list scheduling base method for task scheduling. But in essence, list scheduling algorithm use static rule based on experience or statistics. If the static rule is 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, it uses a more random strategy 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 scheduling problems, there are many task scheduling algorithms is 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 to 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, aiming 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 to speed up it.

## 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 6.

Table 6. 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 18: The flow chart of Genetic algorithm

Typically, a genetic algorithm can conclude as Figure 18, 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.

## 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.

### 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 19 shows an example of the proposed chromosome. In Figure 19, task1 (T1) will be scheduled first, the next one is task2 (T2), and so on.

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.



Figure 19. An example of chromosome

### 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  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | order[0] = 0; // dummy task S  **for** i = 1 to N // N is the number of tasks in task graph  min = MAX(order[Ti\_parent]) + 1;  order[i] = RANDOM\_BETWEEN(min, i); // random number between min and i  **for** j = 0 to (i-1)  **if**(order[j] >= order[i]) **then**  order[j] = order[j] + 1;  **endif**  **endfor**  **endfor**  **for** i = 1 to N  C[order[i]] = i;  **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 19, 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.

### 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 schedule 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. T*i* = the first gene in the chromosome.
2. Remove T*i* from the chromosome.
3. Calculate start time of T*i* as follows:
   1. *a* = MAX(finished time of T*i*’s parents).
   2. *b* = earliest time at which an enough number of cores for executing T*i* become free.
   3. Start time of T*i* = MAX(*a*, *b*).
4. Finish time of T*i* = start time of T*i* + execution time of T*i*.
5. Assign the cores which were selected at step 2.2 to T*i.*
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.

### 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:

(14)

|  |  |
| --- | --- |
|  |  |

denotes the minimum fitness value in population, and 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).

### 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.

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.



Figure 20. An example of crossover.

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

In Figure 20, 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.

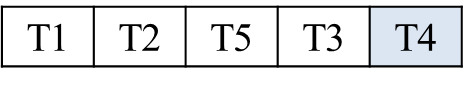
### Mutation

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.

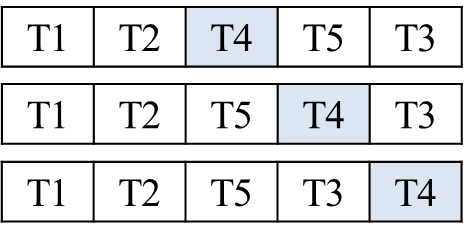
In the proposed chromosome, the value of i-th gene indicates the task whose execution order is i-th. Our mutation changes the order of execution of the task by the following algorithm.

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:
   1. *upper* = the current location of the task.
   2. *lower* = MAX(locations of its parents) + 1.
   3. New location of the task = RANDOM\_BETWEEN(*lower, upper*).
4. Move the task to the new location and slide other tasks accordingly.

Figure 21 shows an example of mutation for the task graph in Figure 1. Assume that T4 in the chromosome in Figure 21(a) is selected for mutation in steps 1 and 2. According to Figure 1, T2 is a parent of T4. Therefore, T4 cannot be moved before T2, and there exist three possibilities for mutation of T4 as shown in Figure 21 (b). Our mutation algorithm chooses one of the three mutations randomly.



1. A chromosome



1. Possible mutations

Figure 21. An example of mutation.

### 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 22.

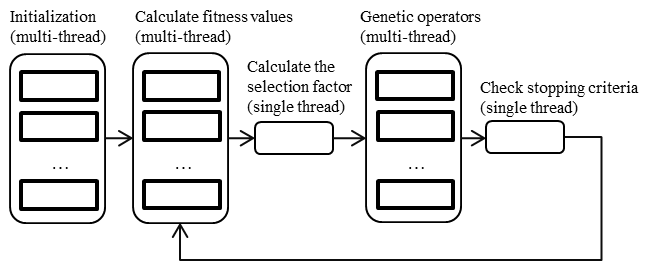


Figure 22. The parallelization framework.

## 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 7.

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

Table 7. The list of parameters.

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

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 7-b. Schedule 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 | |



Figure 23. Results of the four scheduling algorithms on four cores.

The results of scheduling for task graphs with 50 tasks are shown in Figure 7-a and Figure 7-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 9-a. Schedule 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 8-b. Schedule 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 |

For ease of comparison between the other algorithms, we normalize all results to PCS in Figure 23 and Figure 24. Figure 23 shows that for task graphs with 50 tasks, our genetic algorithm achieves 2.5%, 5.2%, 6.8% and 6.8% reduction in the schedule length on 2, 4, 8 and 16 cores, respectively, compared with the PCS algorithm. And Figure 24 shows that for task graphs with 100 tasks, our genetic algorithm achieves 2.5%, 5.2%, 6.8% and 6.8% reduction in the schedule 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 10. 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 multicore platforms. The parallelized implementation achieved approximately seven times speed-up.



Figure 24 Results of the four scheduling algorithms on eight cores.

Table 10. Runtimes of the 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 |

# Branch-and-Bound Algorithm

In chapter 4, 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 achieve optimal solutions. Finding optimal solutions is indispensable to evaluate the quality of the algorithms. Also, finding 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.

## Depth-First Search

Our algorithm uses a branching tree to enumerate all possible schedules systematically. For example, Figure 25 shows a branching tree for the task graph in Figure 1. In the tree, each node represents a task, and a branch between two nodes denotes that the parent task is scheduled no later than the child task. A path from the root to a leaf denotes a schedule. For example, the path in Figure 26 denotes the schedule shown in Figure 25. 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 (Figure 26 (a), (b) and (c)) 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 1(b). in Figure 2 also result in the same schedule as shown in Figure 1(b). 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.

Figure 25: The tree enumerates all possible solutions.



(a)



(b)



(c)

Figure 26: Valid scheduling results for Task S →Task 1→Task 3→Task 2→Task 4→Task 5

## 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.

### Related Pattern Rule

Let us consider the branching tree in Figure 27, 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 28(a) and Figure 28(b) show time charts of partial schedules (1 → 2) and (2 → 1), respectively. In Figure 28(a), one of the four cores is 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.

Similarly, tasks 3, 4 and 5 are schedulable at time 30 in Figure 28(a). In Figure 28(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.

Figure 27: Related patterns

### Exclusive Task Branch Rule

Let us consider the task graph in Figure 9. Initially, either task 1 or 2 is schedulable at time 0. In this case, scheduling task 1 first leads to an optimal schedule in the following reason.

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

There are two types of *exclusive* task.

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

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

### Reducing Meaningless Idle Time

Let us consider partial schedule in the branching tree shown in Figure 27. There are three branches from task 2, going to tasks 3, 4 and 5. If we look at the time chart in Figure 28 (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 travelling 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 28. Partial schedules with same tasks

### 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 schedule length. If the lower bound is longer than the length of the temporarily-optimal schedule, the branch is pruned.

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

(15)

(16)

In the formulas, denotes the available time of core j. For example, in Figure 28 (a), *ATj* is 30 for 0 ≤ *j* ≤ 2, and *AT3* = 10. *φ* is a set of tasks which are not yet scheduled. *Pi* and *Ti* 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 schedule 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.

(17)

## 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 11. Optimal results for graphs with 10 tasks on 4 cores

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Task graph ID | Schedule 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 |

## 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 11 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 schedule 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 any cases of 10 tasks, our branch-and-bound algorithm found optimal schedules within a second. On the other hand, the runtime of CPLEX significantly varied depending on the task graph. In the worst case, it took more than 60 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 runtime 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 12-a. Schedule lengths for task graphs with 50 tasks on 2 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task graph IDs | Schedule Length | | | | | B&B  Runtime  12 hour  (sec) |
|
|
| PCS | Dual- mode | GA | B&B  12 hour | 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 12-b. Schedule lengths for task graphs with 50 tasks on 4 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task graph IDs | Schedule Length | | | | | B&B  Runtime  12 hour  (sec) |
|
|
| PCS | Dual- mode | GA | B&B  12 hour | 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 12-c. Schedule lengths for task graphs with 50 tasks on 8 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task graph IDs | Schedule Length | | | | | B&B  Runtime  12 hour  (sec) |
|
|
| PCS | Dual- mode | GA | B&B  12 hour | 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 12-d. Schedule lengths for task graphs with 50 tasks on 8 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task graph IDs | Schedule Length | | | | | B&B  Runtime  12 hour  (sec) |
|
|
| PCS | Dual- mode | GA | B&B  12 hour | 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 12. The tables show not only the schedule 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 29. Average schedule length normalized by dual-mode for

task sets with 50 tasks

Table 13-a. Schedule lengths for task graphs with 100 tasks on 2 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task graph IDs | Schedule Length | | | | | B&B  Runtime  12 hour  (sec) |
|
|
| PCS | Dual- mode | GA | B&B  12 hour | 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 13-b. Schedule lengths for task graphs with 100 tasks on 4 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task graph IDs | Schedule Length | | | | | B&B  Runtime  12 hour  (sec) |
|
|
| PCS | Dual- mode | GA | B&B  12 hour | 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 13-c. Schedule lengths for task graphs with 100 tasks on 8 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task graph IDs | Schedule Length | | | | | B&B  Runtime  12 hour  (sec) |
|
|
| PCS | Dual- mode | GA | B&B  12 hour | 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 13-d. Schedule lengths for task graphs with 100 tasks on 16 cores

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Task graph IDs | Schedule Length | | | | | B&B |
|
|
| PCS | Dual- mode | GA | B&B  12 hour | 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 13. 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 29 and Figure 30, each bar indicates the average schedule length of 20 task graphs which is normalized to the B&B (limited in 12 hour). 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 30. Average schedule length normalized by dual-mode for  
task sets with 100 tasks

# 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 present a series of different algorithms for task scheduling for data-parallel tasks on multicore architectures.

In Section 4, we propose six algorithms base on list scheduling. The experimental results show that, among the six algorithms, the PCS algorithm yields the best schedule 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 with common list scheduling algorithm, the proposed algorithm has two priority types, and changes its behaviour under the available cores conditions of system. This algorithm has achieved 2%, reduction in the schedule 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 schedule lengths over the PCS or dual-mode algorithm.

In order to deeper understand the scheduling problem and better evaluate the effectiveness of proposed algorithms. The study about finding optimal solutions for task scheduling is also indispensable. We propose 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).

# 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 PhD 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 deeply 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 Lecturer 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 PhD. I would also like to thank Lecturer 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 the vice-president of my PhD Committee.

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, September1994.
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](https://www.sciencedirect.com/science/journal/03032647/43/2), 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](https://www.sciencedirect.com/science/article/pii/S1568494612001901)," *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)

# Achievements

## Journal

* 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.
* 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

* Y. Liu, I. Taniguchi, H. Tomiyama, and L. Meng, "List Scheduling Strategies for Task Graphs with Data Parallelism," *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," *Proc. of Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 371-374, Ishigaki, Japan, November 2014.
* 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.