TASK SCHEDULING AND RESOURCE ALLOCATION ON PARALLEL AND DISTRIBUTED MACHINES

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2022

Chong Ke

Department of Computer Science

Contents

Abstract **10**

Declaration **12**

Copyright **13**

Acknowledgements **14**

1 Introduction **15**

1.1 Motivation 17

1.2 Contribution 17

1.3 Thesis structure 18

2 Task graphs and scheduling **19**

2.1 Task graphs 19

2.2 Scheduling 23

2.3 Constraints 26

3 Scheduling models **28**

3.1 Classic model 28

3.1.1 A scheduling example in the classic model 30

3.2 Contention model 32

3.2.1 One-port model 33

3.2.2 A scheduling example in the one-port model 34

4 Existing scheduling approaches **37**
4.1 List scheduling . . . 37
4.1.1 Static list scheduling algorithms . . . 38
4.1.2 Dynamic list scheduling algorithms . . . 38
4.1.3 List scheduling in the classic model . . . 39
4.1.4 Some list scheduling approaches . . . 43
4.2 Clustering scheduling . . . 46
4.2.1 Generic clustering algorithms . . . 47
4.2.2 Clustering algorithms in the classic/one-port model . . . 49
4.3 Other scheduling algorithms . . . 51
4.4 Summary . . . 55

5 A look-forward algorithm for task scheduling **56**
5.1 Introduction . . . 56
5.2 Task scheduling (DAG, model, basic list scheduling) . . . 59
5.3 HEFT, look-ahead and look-forward algorithm . . . 61
5.3.1 HEFT . . . 61
5.3.2 Look-ahead . . . 63
5.3.3 Look-forward . . . 64
5.4 Experimental evaluation . . . 64
5.4.1 DAX and generator . . . 64
5.4.2 The simulator and settings . . . 66
5.4.3 Results . . . 67
5.4.4 Discussion . . . 86
5.4.5 Algorithm execution time . . . 87
5.5 Summary . . . 88

6 Look-forward scheduling in the contention model **93**
6.1 List scheduling in the one-port model . . . 93
6.2 Look-forward algorithms under the contention model . . . 98
6.3 Experimental setup . . . 98
6.3.1 Graphs . . . 98
6.3.2 DAX generator and simulator settings . . . 101

6.3.3 Results and evaluation 102
6.4 Summary 103

7 The identical data problem **107**
7.1 The identical data problem description 108
7.2 Identical data in list scheduling algorithms 111
7.3 Using the look-forward algorithm for identical data under the contention model 113
7.4 Experimental evaluation 113
7.4.1 HEFT scheduling versus look-forward 116
7.5 Summary 117

8 Conclusion **126**

Bibliography **128**

Word Count: 24271

List of Tables

5.1 Communication improvement of look-ahead and look-forward when the number of resources is 10 78
5.2 Communication improvement of look-ahead and look-forward when the number of resources is 2 79

List of Figures

2.1 Incoming and outgoing nodes and edges 20
2.2 Node level 22
2.3 Dominant sequence example 23

3.1 A fully connected network system 30
3.2 Scheduling example in classic model 31
3.3 Contention model and path finding 33
3.4 One-port model 34
3.5 Scheduling example in the one-port model 35

4.1 An example to illustrate how communication cost may affect the schedule length. 40

5.1 Average makespan with standard deviation using Montage workflow, CCR 0.5 and 2 resources 69
5.2 Average makespan with standard deviation using Montage workflow, CCR 0.5 and 10 resources 69
5.3 Average makespan with standard deviation using Montage workflow, CCR 1.0 and 2 resources 70
5.4 Average makespan with standard deviation using Montage workflow, CCR 1.0 and 10 resources 70
5.5 Average makespan with standard deviation using Montage workflow, CCR 2.0 and 2 resources 71
5.6 Average makespan with standard deviation using Montage workflow, CCR 2.0 and 10 resources 71

5.7 Average makespan with standard deviation using ligo workflow, CCR 0.5 and 2 resources 72
5.8 Average makespan with standard deviation using ligo workflow, CCR 0.5 and 10 resources 72
5.9 Average makespan with standard deviation using ligo workflow, CCR 1.0 and 2 resources 73
5.10 Average makespan with standard deviation using ligo workflow, CCR 1.0 and 10 resources 73
5.11 Average makespan with standard deviation using ligo workflow, CCR 2.0 and 2 resources 74
5.12 Average makespan with standard deviation using ligo workflow, CCR 2.0 and 10 resources 74
5.13 Average makespan with standard deviation using cybershake workflow, CCR 0.5 and 2 resources 75
5.14 Average makespan with standard deviation using cybershake workflow, CCR 0.5 and 10 resources 75
5.15 Average makespan with standard deviation using cybershake workflow, CCR 1.0 and 2 resources 76
5.16 Average makespan with standard deviation using cybershake workflow, CCR 1.0 and 10 resources 76
5.17 Average makespan with standard deviation using cybershake workflow, CCR 2.0 and 2 resources 77
5.18 Average makespan with standard deviation using cybershake workflow, CCR 2.0 and 10 resources 77
5.19 Average makespan with standard deviation using epigenomics workflow, CCR 0.5 and 2 resources 78
5.20 Average makespan with standard deviation using epigenomics workflow, CCR 0.5 and 10 resources 79
5.21 Average makespan with standard deviation using epigenomics workflow, CCR 1.0 and 2 resources 80
5.22 Average makespan with standard deviation using epigenomics workflow, CCR 1.0 and 10 resources 80

5.23 Average makespan with standard deviation using epigenomics workflow, CCR 2.0 and 2 resources 81
5.24 Average makespan with standard deviation using epigenomics workflow, CCR 2.0 and 10 resources 81
5.25 Communication amount of montage workflow under 2 resources 82
5.26 Communication amount of montage workflow under 10 resources . . . 82
5.27 Communication amount of cybershake workflow under 2 resources . . . 83
5.28 Communication amount of cybershake workflow under 10 resources . . 83
5.29 Communication amount of epigenomics workflow under 2 resources . . 84
5.30 Communication amount of epigenomics workflow under 10 resources . 84
5.31 Communication amount of ligo workflow under 2 resources 85
5.32 Communication amount of ligo workflow under 10 resources 85
5.33 Average algorithm execution time (in microseconds) of montage workflow under 2 resources 88
5.34 Average algorithm execution time (in microseconds) of montage workflow under 10 resources 89
5.35 Average algorithm execution time (in microseconds) of cybershake workflow under 2 resources 90
5.36 Average algorithm execution time (in microseconds) of cybershake workflow under 10 resources 90
5.37 Average algorithm execution time (in microseconds) of epigenomics workflow under 2 resources 91
5.38 Average algorithm execution time (in microseconds) of epigenomics workflow under 10 resources 91
5.39 Average algorithm execution time (in microseconds) of ligo workflow under 2 resources 92
5.40 Average algorithm execution time (in microseconds) of ligo under 10 Resources 92

6.1 Graph example for edge scheduling 96
6.2 Fork and join graph 98
6.3 Fork-join graph 100
6.4 Series-parallel graph 101

6.5 Out-tree graph 102
6.6 In-tree graph 103
6.7 Fork graph under the contention/one-port model 104
6.8 Fork-join graph under the contention/one-port model 104
6.9 SerieParallel graph under the contention/one-port model 105
6.10 In Tree graph under the contention/one-port model 105
6.11 Out Tree graph under the contention/one-port model 106

7.1 Identical data 108
7.2 Identical data in one-port model 109
7.3 Identical data in classic model 111
7.4 Schedule identical data 112
7.5 HEFT scheduling evaluation (CCR=0.1) 113
7.6 HEFT scheduling evaluation (CCR=1) 114
7.7 HEFT scheduling evaluation (CCR=10) 115
7.8 Look-forward algorithm(CCR=0.1) 116
7.9 Look-forward algorithm(CCR=1) 117
7.10 Look-forward algorithm(CCR=10) 118
7.11 Fork without identical data comparison 119
7.12 Fork with identical data comparison 119
7.13 Fork-join without identical data comparison 122
7.14 Fork-join with identical data comparison 122
7.15 SerieParallel without identical data comparison 123
7.16 SerieParallel with identical data comparison 123
7.17 Out Tree without identical data comparison 124
7.18 Out Tree with identical data comparison 124
7.19 In Tree without identical data comparison 125
7.20 In Tree with identical data comparison 125

Abstract

TASK SCHEDULING AND RESOURCE ALLOCATION
ON PARALLEL AND DISTRIBUTED MACHINES
Chong Ke
A thesis submitted to The University of Manchester
for the degree of Doctor of Philosophy, 2022

Task scheduling is a significant problem for parallel and distributed systems. As a mature and important topic, scheduling has attracted much attention among scholars. How to examine this topic in depth has become interesting and challenging. In this thesis, the author discusses it from three different aspects. Firstly, as a core question in high-performance computing, the scheduling algorithms are the main concern. The author proposes a novel algorithm called 'the look-forward algorithm'. Different from other algorithms, the look-forward algorithm provides a novel way of task allocation, which fully considers the DAG structure and incoming and outgoing communications. Experimental results show that the look-forward algorithm can get up to 40% improvement compared to the benchmark HEFT algorithm [THW02], especially when the number of tasks increases. Secondly, the author considers scheduling using a communication contention model [SS05, SS04, CCK12]. Research has shown the contention model has more stability and accuracy than the classic model in describing communication but it is also more complicated than the classic model because of the communication contention. The author extends the look-forward algorithm for contention model scheduling so that the algorithm can appropriately consider communication. Thirdly, after this algorithm

and scheduling model, the author focuses on communication data. A new problem, the identical data problem, is proposed. Identical data means that exactly the same data is sent from one task to other tasks, which can be used to reduce the communication among processors. Taking this into account, a new algorithm based on the look-forward algorithm is proposed. All three algorithms in this thesis are evaluated using simulation.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I would like to express my sincere gratitude to my supervisor Professor Rizos Sakellariou. Five years ago, I came to the United Kingdom alone and chose him as my supervisor. He taught me how to do good research and led me to the area of parallel/distributed computing. I am thankful for his useful academic advice and weekly discussions. I am also thankful for his suggestion for each of my ideas and for his patience to allow me making mistakes.

I also would like to thank my father, my mother, my beautiful sister and my lovely brother for their constant encouragement and support. They give me the strength to overcome every difficulty in my academic and personal life. I also would like to thank my friend Sarad Venugopalan for discussing my topic with me and giving me much inspirations.

A special thanks to my thesis checking team: Beibei Zhai, Simar Kalra, Sarad Venugopalan, Di Wang and Mark Tooley. Thanks for helping me to check every word of my thesis.

Finally, I would like to express my sincere thanks to the University of Manchester. It is a beautiful place and an amazing university.

Chapter 1

Introduction

Parallel computing has entered into the mainstream in the computing industry, as multi-core processors have now become widely available. This makes parallel programming more relevant than ever, but, regrettably, programming in a parallel system is still difficult and challenging. One of the crucial aspects is scheduling of the tasks of applications onto the processors of the parallel system. Such scheduling is computationally NP-hard [JSM91, KN84, TBW92]. Furthermore, the time needed to transfer data between processors can partially or completely eliminate the benefits of the parallel execution of tasks, which makes scheduling even more difficult.

With processors running at speeds close to their theoretical limit, single-processor systems will not meet the demand of computing. For some time, one proposed way has been the use of parallelism in hardware through multi-core or even dual-core processing systems [AIAS87]. This made scientists work towards the goal of parallelism. The good news today is that parallel and distributed systems [VR01] have entered into the mainstream, as multi-core processors are found everywhere, especially on-chip multiprocessor systems. Simply duplicating processors however does not automatically speed up application execution because the programs cannot automatically run concurrently. To execute an application consisting of several tasks on parallel/distributed machines, the tasks must be arranged in space and time on the multiple processors. In other words, the tasks must be mapped to the processors and ordered for execution. Besides this, the resources (jobs, tasks, communications) also need to be carefully assigned to the processors [Sin07]. These steps, which are referred to as *task scheduling* and *resource*

allocation, fundamentally determine the efficiency of the application's parallelization, that is, the speedup of its execution in comparison to a single-processor system.

Even before multi-core processor systems were readily available, scientists realized the importance of task scheduling and resource allocation in program parallelisation. Many parallel models and relevant distributed computing algorithms have been proposed [CTS97, Ert98, BD04]. However, these types of distributed approaches are constrained by their NP-hard complexity [TBW92, SKH95]. That is, the heavy inner cores' communication costs can partially or completely eliminate the benefit of the parallel execution of codes. Some fundamental heuristics, such as list scheduling, clustering, genetic approaches and others, have been studied to solve this problem, even though these heuristics can also have their shortcomings.

Except for the parallel model and distributed heuristics, scientists have been making many efforts towards studying parallel architectures. Flynn's taxonomy classifies the basic parallel architectures according to the multiplicity of the instruction and the data flows [Sny88]. SIMD (single instruction multiple data), MIMD (multiple instructions multiple data) are the two basic architectures within Flynn's taxonomy. Besides Flynn's taxonomy, it is generally agreed that memory architectures (including both centralized or shared memory and distributed memory) and the message passing model are also important for this topic. In order to represent the tasks and resources in a computer, relevant graph theory constructs are used, especially task graphs, flow graphs and dependence graphs [Sin07].

A program to be scheduled is represented by a directed acyclic graph (DAG) [Han94], where a node represents a task and an edge between two nodes represents the communication between two tasks. A contention model, the so-called one-port model [BMRR06], is introduced in Section 3.2.1 to represent a parallel/distributed system. When the communication costs and data transmission contention are considered, it is difficult to find an efficient schedule for task scheduling [SS05, RSS90]. One reason is that the conflict in the communication contention will make it hard for the scheduling algorithm to make a decision for every node. One approach to reduce the contention cost is to consider identical data under the contention/one-port model. Some new algorithms that can handle contention for identical data are proposed in this thesis.

List scheduling and clustering [SS01, CR92, GY92a, Gra99, PLW96, YG93, CJ01] are two important types of scheduling heuristics in the traditional task scheduling area,

especially under the classic model [SSS06, Sin07]. Considering identical data has only a little effect on the schedule length in the classic task scheduling model (see Section 3.1) but may be essential in a contention model (see Section 7.1). As only a few algorithms have been proposed under the contention model [SSS06], this work will focus on task scheduling under the contention model while also considering identical data. However, with respect to the contention model, neither list scheduling nor clustering is efficient in finding a good schedule length for task scheduling. As the typical list scheduling algorithms do not look forward and cannot consider the contention between two processors (even though the typical clustering scheduling algorithm may look forward in some ways), it is almost impossible to determine which nodes should be scheduled in the same cluster when communication contention is taken into consideration.

1.1 Motivation

High-performance computing areas are developing very fast these days. Cloud computing but also grid computing or transparent computing all rely on large-scale parallel or distributed systems. How to schedule parallel tasks in these parallel/distributed machines is one of the most important problems as this will affect the whole performance of an application. A lot of research work has been done in task scheduling or resource allocation in the parallel and distributed scheduling research area. DAG scheduling and the DAG model are the most significant methods used to schedule parallel tasks. Many problems are still challenging to researchers. For example, existing approaches cannot fully consider problems such as inaccuracy in task execution time (especially when the computation costs of tasks are unknown), intensive communication, different communication models, highly heterogeneous platforms and so on. If these problems could be addressed or better solutions could be found, the whole performance of the systems would be improved and the resources would be managed more efficiently.

1.2 Contribution

This thesis examines how it might be possible to improve DAG scheduling in parallel and distributed systems. There are three keywords in this work: algorithm, model

and data. These three aspects are studied in this work. Firstly, considering that the scheduling algorithm can reduce the schedule length and improve the efficiency of the whole system, after investigating the two fundamental heuristics for task scheduling – list scheduling and clustering – this thesis constructs a novel algorithm named look-forward to do DAG scheduling (see Chapter 5). The experiments show that this algorithm can significantly reduce scheduling length. Secondly, this thesis studies the contention model and provides a new algorithm that can be used under a contention model and reduce communication contention (see Chapter 6). Thirdly, a newly proposed algorithm can facilitate efficient scheduling in the contention model when identical data are considered. This algorithm introduces identical data into the task scheduling domain. When considering identical data, scheduling algorithms can reduce the schedule length by making good use of the properties of identical data (see Section 7.1). Since, when considering identical data, the classic model task scheduling model may not offer much improvement (see Section 3.1), in this part of the thesis, a contention scheduling model, the one-port model, is used.

1.3 Thesis structure

The rest of this thesis is structured as follows: Chapter 2 gives some background on task scheduling. An overview of the notions of task graph and scheduling will be introduced there. Chapter 3 reviews the task scheduling models. The properties of the classic model and contention/one-port model and how to define identical data will be discussed in detail. Chapter 4 provides an introduction to some existing scheduling approaches. Chapter 5 includes a detailed discussion of the look-forward algorithm and its advantages and drawbacks. Chapter 6 discusses the contention model and extends the look-forward algorithm to deal with a contention model. Chapter 7 takes into account identical data and considers how it could benefit from the look-forward algorithm. Finally, Chapter 8 looks at the whole thesis and makes suggestions for further research.

Chapter 2

Task graphs and scheduling

2.1 Task graphs

One of the foundational problems for parallel and distributed computing is task scheduling in parallel or distributed systems. When an application or a job has many tasks and runs on a multi-core or multiprocessor system, these tasks should be assigned to the cores or processors carefully to make them execute correctly and have a short execution time or low resource consumption. This assignment will be done via task scheduling. Task scheduling allocates the tasks to the processors and schedules these tasks in the allocated processors by giving each task a start time [Sin07]. The difference between allocation and scheduling is discussed in Section 2.2. In this work, the main job of a task scheduling algorithm is to find a close-to-optimal executing sequence for the application under different task scheduling models (see Chapter 3).

Before discussing task scheduling in parallel and distributed systems, it is necessary to introduce the basic concepts of a task graph, scheduling and related nomenclatures. In the task scheduling area, a task graph is used to represent the application or job that is of interest to the user. It is often represented by a directed acyclic graph (DAG). A directed acyclic graph is a graph whose vertices are connected by directed edges and has no directed cycles: that is, there is no path that starts from one vertex v and follows a sequence of vertices and directed edges and then goes back to v. In a DAG, the vertices are labelled nodes, the weight of the nodes represents the computing cost and the weight of a uni-directional edge corresponds to the communication cost between tasks.

For task scheduling, the program to be scheduled is represented by a directed acyclic graph (DAG) $G = (V, E, w, c)$, called task graph [XW01, KA99], where V denotes the finite set of tasks and E represents the finite set of edges. The non-negative weight $w(n)$ associated with node $n \in V$ gives the computing cost of a task, the non-negative weight $c(e_{ij})$ associated with edge $e_{ij} \in E$ is the communication cost between two tasks. As shown in Figure 2.1, a node n_i or $i \in V$ (in this figure n_i is labelled as i to make it simple) of a DAG represents a task and a directed edge $e_{ij} \in E$ from node n_i to n_j, n_i, $n_j \in V$ represents the communication edge of two tasks. When two tasks are allocated on the same processor, there is no communication cost between these two tasks.

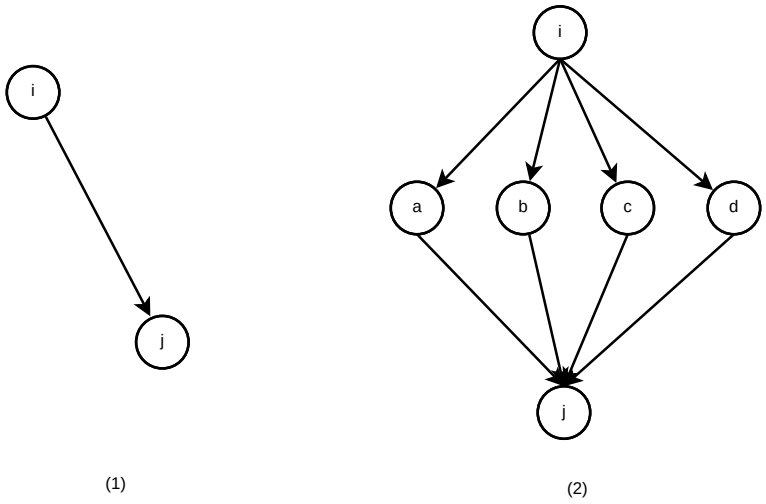

Figure 2.1: Incoming and outgoing nodes and edges

Some important notions relevant to the task graph are as follows:

Incoming and outgoing nodes. As shown in Figure 2.1(1), node n_i and n_j are connected by edge e_{ij}. Node n_i is the incoming node of e_{ij} and node n_j is the outgoing node via e_{ij}. An edge can only have one incoming node and one outgoing node.

Incoming and outgoing edges. As shown in Figure 2.1(1), node n_i and n_j are connected by edge e_{ij}. Edge e_{ij} is the outgoing edge of n_i and the incoming edge of n_j. One node can have more than one outgoing or incoming edges. For example, in Figure 2.1(2), edges to nodes a, b, c, d are the outgoing edges of node n_i and edges from nodes a, b, c, d are the incoming edges of node n_j.

Parent nodes and children nodes. Let $G = (V, E, w, c)$ be a task graph, $n_i \in V$ is a node in G. The parent nodes of n_i are all the incoming nodes of n_i. The children nodes of n_i are all the outgoing nodes of n_i.

Path. A path p in a task graph $G = (V, E, w, c)$ is a sequence of nodes $(v_0, v_1 .... v_n)$ such that from each node v_k, there is an edge to the next node v_{k+1} in the sequence. A path p in G must be finite. A finite path p always has a first node, n_{start}, called the start node, and a last node, n_{end}, called the end node. Both of them are called end or terminal nodes of the path. The other nodes in the path are internal nodes [ACKZ05].

Node level. Let $G = (V, E, w, c)$ be a task graph; the node level is the number of nodes in the shortest path from the start node (source node, entry node) to the end node (terminal node, exit node). The start node (source node, entry node) indicates a node without incoming communication and parent nodes. The end node (terminal node, exit node) indicates a node without outgoing communication and children nodes. As shown in Figure 2.2, all the nodes $n_i \in V$ are separated into different levels by the dotted lines. For example, node a is in level 0 and nodes e, g are in level 2. The number of the level is the node level of a node.

Path length. Let $G = (V, E, w, c)$ be a task graph and $p = (v_0, v_1 .... v_n)$ be a path in $G = (V, E, w, c)$. The length of path p is the sum of all nodes'computation costs w and all edges'communication costs c:

$$len(p) = \sum_{v \in p} w(v) + \sum_{e_{ij} \in p} c(e_{ij})$$

Critical path. [RVB07] Let $G = (V, E, w, c)$ be a task graph; in task graph G, a node without an incoming edge is called a source node and a node without an outgoing edge is called a terminal node. If p is a path in $G = (V, E, w, c)$ and $len(p)$ is the length of the path p and the start node of p is a source node of G, the end node of p is a terminal node of G. A critical path cp in a task graph $G = (V, E, w, c)$ is one of the longest paths of G:

$$cp(G) = max_{p \in G}\{len(p)\}$$

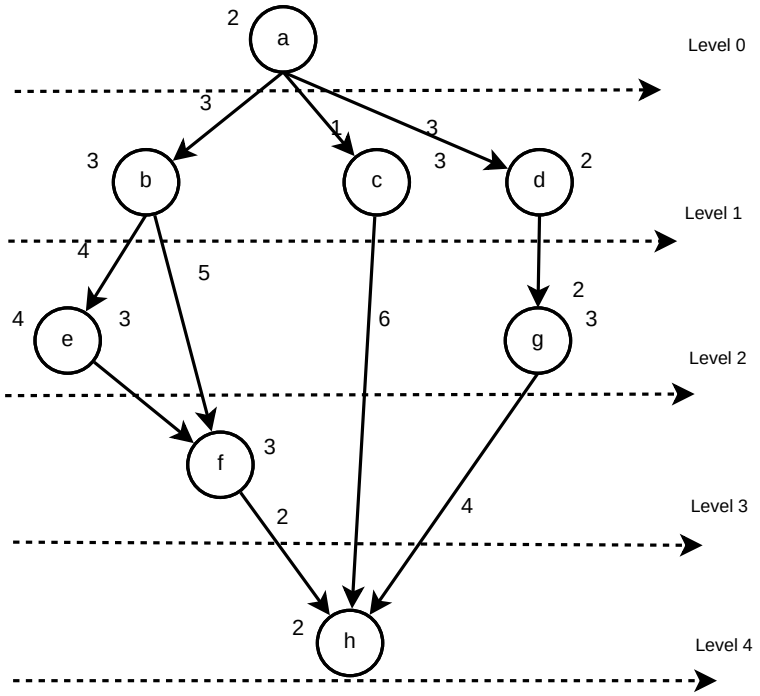

Figure 2.2: Node level

Dominant sequence. The critical path is the maximal length path of a task graph. The dominant sequence is another maximal length path of the graph. As shown in Figure 2.3, nodes a, d, c, g and f have already been scheduled or allocated to processors and nodes b, e and h have not yet been considered by the scheduling algorithm. Let $V_{scheduled}$ be the set of nodes that have already been scheduled or allocated and V_{others} be the set of nodes that have not been checked. Then $V_{scheduled}$, V_{others} and the edges e_{ij} between these nodes can form a new graph G_1 (if two nodes are on the same processor, the weight of the edge between them is zero). Sometimes the scheduling algorithms need to find the longest path of G_1, which is called the dominant sequence. The method of how to calculate the dominant sequence is introduced by Gerasoulis and Yang [GY92a, GY93, YG94, YG93]. The dominant sequence is often used in clustering algorithms for task scheduling (see Section 4.2).

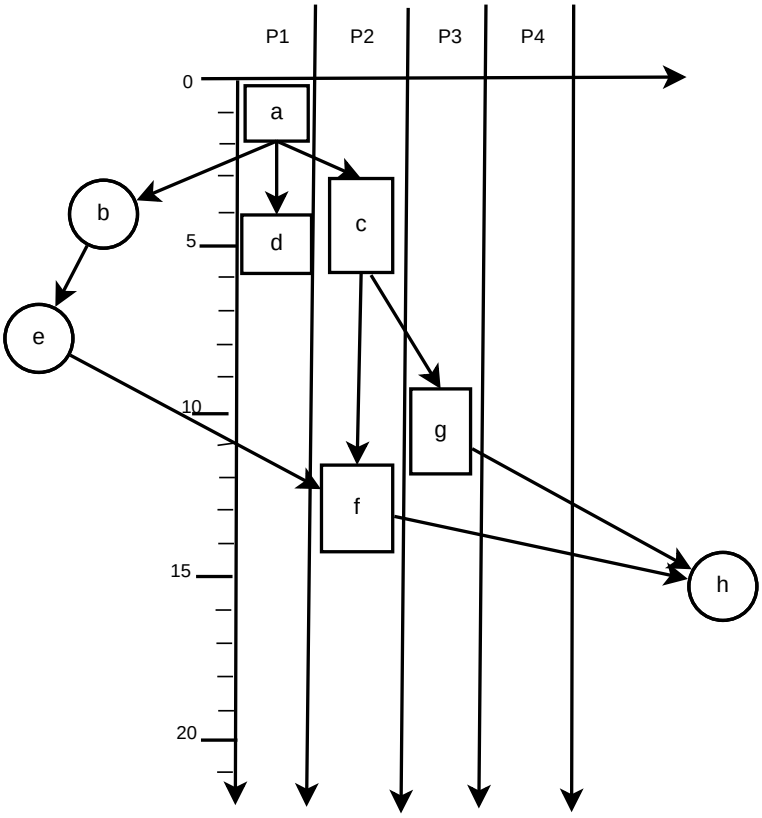

Figure 2.3: Dominant sequence example

2.2 Scheduling

Processor allocation [MVZ93, CS87] Let $G = (V, E, w, c)$ be a task graph and P be a parallel or distributed system. Processor allocation of $G = (V, E, w, c)$ means allocating all the nodes $n \in V$ to some or all processors $p \in P$. Note that each node $n \in V$ is allocated to only one processor.

Node start time and finish time Let $G = (V, E, w, c)$ be a task graph, P be a parallel or distributed system and S be a schedule of task graph $G = (V, E, w, c)$ on system P. For each scheduled node n, there is a start time $t_s(n)$ and a finish time $t_f(n)$ for n, where $t_s(n)$ is the start execution time of node n and $t_f(n)$ is the end execution time of n [STK09]. When node n is allocated to processor p, the start execution time of n can be denoted by $t_s(n, p)$. Correspondingly, the end execution time of n can be denoted by $t_f(n, p)$. The

finish time $t_f(n_i, p)$ of node n_i on $p \in P$ is given by:

$$t_f(n_i, p) = t_s(n_i, p) + w(n_i).$$

In other words, the finish time is the start time plus the task's computation weight.

Data ready time (DRT) [STK09, TS84] Let $G = (V, E, w, c)$ be a task graph and P be a parallel or distributed system. The data ready time of $n_i \in V$ on processor $p_i \in P$ is the time when all the incoming communications of n_i are finished. The term incoming communications refers to communications that come from other nodes to node n_i. Normally, the data ready time of n_i on processor p_i is:

$$t_{dr}(n_i) = \max_{n_j \in pred(n_i)} (t_f(e_{ji}, proc(n_j), p_i). \tag{2.1}$$

In Equation 2.1, $pred(n_i)$ means all the predecessor nodes of node n_i, $proc(n_j)$ means the processor in which the node n_j is allocated, e_{ji} means the edge between nodes n_j and n_i, $t_f(e_{ji}, proc(n_j), p_i)$ means the finish time of edge e_{ji}, which starts from processor $proc(n_j)$ and ends on processor p_i.

Node scheduling Let $G = (V, E, w, c)$ be a task graph and P be a parallel or distributed system. A scheduling of node $n_i \in V$ is a method to first allocate n_i to a processor $p \in P$ and then calculate the start time $t_s(n, p)$ and finish time $t_f(n, p)$ for n_i. Often, the allocation of node n_i is based on the earliest start time of n_i, which means n_i is allocated to the processor where it can start earliest. The earliest start time $t_s(n_i, p)$ of node n_i on $p \in P$ is given by:

$$t_s(n_i, p) = max(DRT, t_f(p)),$$

where DRT is an abbreviation for data ready time [SS04], also discussed above, and $t_f(p)$ is the finish time of processor p. Please note that node allocation will choose a processor for the node (or allocate the node to one processor). The node does not have a start time $t_s(n, p)$ and finish time after node allocation. This is also called processor allocation (see Section 2.2). Node scheduling does the node/processor allocation first and then gives a start time $t_s(n, p)$ and a finish time $t_f(n, p)$ to the node. Then, $t_f(p)$ is the finish time of processor p.

Free node Let $G = (V, E, w, c)$ be a task graph, P be a parallel or distributed system and S be a schedule of task graph $G = (V, E, w, c)$ on system P. Node $n_i \in V$ is a free node if n_i has no incoming node or all its incoming nodes are scheduled. In some literature, a free node is also called a ready node.

Edge scheduling [SSS06, WSF89, GDP08] Let $G = (V, E, w, c)$ be a task graph and P be a parallel or distributed system. Scheduling of edge $e_{ij} \in E$ is a method to find a schedule path (see Section 2.1) for e_{ij} and calculate the start time $t_s(e_{ij})$ and finish $t_f(e_{ij})$ of e_{ij}. Often, edge scheduling is used when the path cannot transfer communication concurrently and there is contention in the path.

Edge finish time Let $G = (V, E, w, c)$ be a task graph, P be a parallel or distributed system and S be a schedule of task graph $G = (V, E, w, c)$ on system P. The edge $e_{ij} \in E$ is the edge between nodes $n_{i,}n_j \in V$ and e_{ij} is communicated from processor p_{src} to p_{dst}. The edge finish time of e_{ij} is:

$$t_f(e_{ij}, p_{src,}p_{dst}) = t_f(n_{i,}p_{src}) + \begin{cases} 0 & if\ p_{src} = p_{dst} \\ c(e_{ij}) & otherwise \end{cases}$$

Processor finish time Let $G = (V, E, w, c)$ be a task graph, P be a parallel or distributed system and $p \in P$ be the processors in the parallel or distributed system. When a node n_i is scheduled in a processor p_i, the processor will have a finish time $t_f(p_i) = t_f(n_i, p_i)$. If n_i is the last node that executes on processor p, the processor finish time of p equals the finish time of node n_i. Whenever a new node is scheduled on p_i, the finish time of p_i will change.

Inter-processor communication and local communication [LCC+13, KDR13, JFKG13] Inter-processor communication means the communication between two different processors. When a node n_i on processor p_i needs to send or receive data from a node n_j on processor $p_j (p_i \neq p_j)$, this data transfer is called inter-processor communication. When two nodes, n_i and n_j, which are on the same processor p_i need to communicate with each other, this communication is called local communication. Normally, the local communication time is very small and can be ignored by the system

model, meaning that the local communication cost is assumed to be zero.

2.3 Constraints

In order to make the schedule feasible, two conditions, called exclusive processor allocation and precedence constraints [Sin07, STK09, CS87] must be fulfilled by all the nodes and edges in $G = (V, E, w, c)$. Exclusive processor allocation requires that the nodes allocated on the same processor must be executed one after the other (only one node can be executed in this processor at any time). A precedence constraint requires that a non-source node can only be executed when all its incoming edges have finished.

Exclusive processor allocation Let $G = (V, E, w, c)$ be a task graph, P be a parallel or distributed system and S be a schedule of task graph $G = (V, E, w, c)$ on system P. If two nodes, $n_i, n_j \in V$, are scheduled on the same processor p_i, their execution must be satisfied, which can be formally described as:

$t_s(n_i) < t_f(n_i) \leq t_s(n_j) < t_f(n_j)$ or $t_s(n_j) < t_f(n_j) \leq t_s(n_i) < t_f(n_i)$.

That is because n_i and n_j are on the same processor and one processor can only execute one node at a time. So the finish time of one node must be earlier or the same as the start time of the other node. Only nodes that can potentially run in parallel need to satisfy this constraint.

Precedence constraint Let $G = (V, E, w, c)$ be a task graph, P be a parallel or distributed system and S be a schedule of task graph $G = (V, E, w, c)$ on system P. For node $n_i \in V$, if e_{ji} is anyone of node n_i's incoming edges, then the following constraint specifies the precedence relation:

$$t_s(n_i) \geq t_f(e_{ij,} proc(n_j), p_i).$$

This means node n_i must not start earlier than the finish time of any of its incoming edges. A node must be started after all its communication data is ready, which means all precedence constraints are met. In scheduling algorithms, all nodes must satisfy the exclusive processor allocation and precedence constraint properties.

Finally, for any source node n_{source}, the start time of n_{source} will meet the following constraint:

$$t_s(n_{source}) \geq 0$$

This constraint implies that the whole application will always have a positive execution time.

Chapter 3

Scheduling models

3.1 Classic model

The parallel or distributed system in which the tasks will be executed will be discussed in this section. In order to discuss the target parallel/distributed system, a model which represents the processors and the connections between these processors must be defined. Most scheduling algorithms employ a strongly idealized model of the target parallel/distributed system. This model, which shall be referred to as the classic model [FAdM$^+$12, AFJ$^+$13], is defined as a parallel or distributed system P which consists of a finite number of processors and a fully connected communication network. The properties of the classic model are defined by O. Sinnen [Sin07] as follows :

1. Dedicated System. The target system can only execute one program or one task graph at a time. When the system P is executing a task graph G, nothing else can be executed on P until G is finished.

2. Dedicated Processor. When the processor $p \in P$ is executing one task or node n_i, p cannot execute anything until n_i is finished.

3. Local communication has zero cost. When two nodes n_i and n_j are executed by the same processor, the communication between them is defined as local communication. Compared to remote communication, where n_i and n_j are on different processors, the local communication cost is so small that it can be ignored by the scheduling algorithm. This property has an important effect in the task scheduling

area for the reason that many tasks may be scheduled on the same processor in order to keep the communication cost between them zero.

4. Communication is performed by a communication subsystem. This means that the processors are not involved in the communication. When tasks between two processors need to communicate with each other, which is called inter-processor communication, the processors just give the information to the communication subsystem and the subsystem will do the communication independently. Processors will not spend time in the communication.

5. Communication can be performed in parallel. Many communications that pass through the same links between processors can be done in parallel. For example processor p_1 can send data to processors p_2, p_3, p_4 at the same time without any resource contention.

6. The communication network is fully connected. This means every processor p in the parallel/distributed system can communicate directly with every other processor via a dedicated link between them. Figure 3.1 shows an example of a fully connected communication network system. In this system, every processor is connected to every other processor directly via a link. Note that these links can perform multiple communications in parallel in the classic model.

Based on this system model, the edge finish time only depends on the finish time of the origin node and the communication time. The edge finish time of e_{ij} is given by

$$t_f(e_{ij}, p_{src}, p_{dst}) = t_f(n_i, p_{src}) + 0,$$

if $p_{src} = p_{dst}$, and

$$t_f(e_{ij}, p_{src}, p_{dst}) = t_f(n_i, p_{src}) + c(e_{ij}),$$

otherwise. Thus, communication can overlap with the computation of other nodes, an unlimited number of communications can be performed at the same time, and communication has the same cost $c(e_{ij})$, regardless of the origin and the destination processor, unless the communication is local.

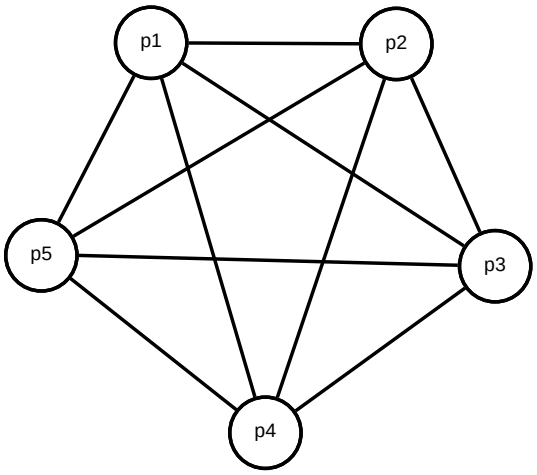

Figure 3.1: A fully connected network system

3.1.1 A scheduling example in the classic model

Based on the task graph and the general scheduling model discussed in Chapter 2, this subsection gives an example of scheduling under the classic model. Figure 3.2 depicts a sample task graph (see Figure 3.2(a)), and a Gantt chart of a schedule on four processors (see Figure 3.2(b)). A Gantt chart is an intuitive and common graphical representation of a schedule, in which each node is drawn as a rectangle. In Figure 3.2(b), the horizontal axis represents the processors in the parallel/distributed system and the vertical axis represents the execution time. The rectangles in the Gantt chart reflect the nodes in the task graph. Obviously, the start time of a node is aligned with the top side of the corresponding rectangle and the node finish time with the bottom side. The computation time is the length of the rectangle.

A simple scheduling procedure was used to create the schedule for Figure 3.2, where the node scheduling order is (a,b,c,d,e,f,g,h) and the algorithm allocates nodes to processors in a round robin fashion, starting with p_1. As seen in the Gantt chart, the start time of node a executed on processor p_1 is $t_s(a) = 0$ and the computation cost is $w(a) = 2$, the finish time of node a is $t_f(a) = t_s(a) + w(a) = 2$. Node b needs to receive communication data from node a and in order to make node b start from the earliest start time, b should start at $t_s(b) = t_f(e_{ab}, p_1, p_2) = t_f(a) + c(e_{ab}) = 2+3 = 5$. The computation cost of node b is $w(b) = 3$ and the finish time of node b is $t_f(b) = t_s(b) + w(b) = 8$. Node c also receives data from node a; the start time $t_s(c) = 3$ is the

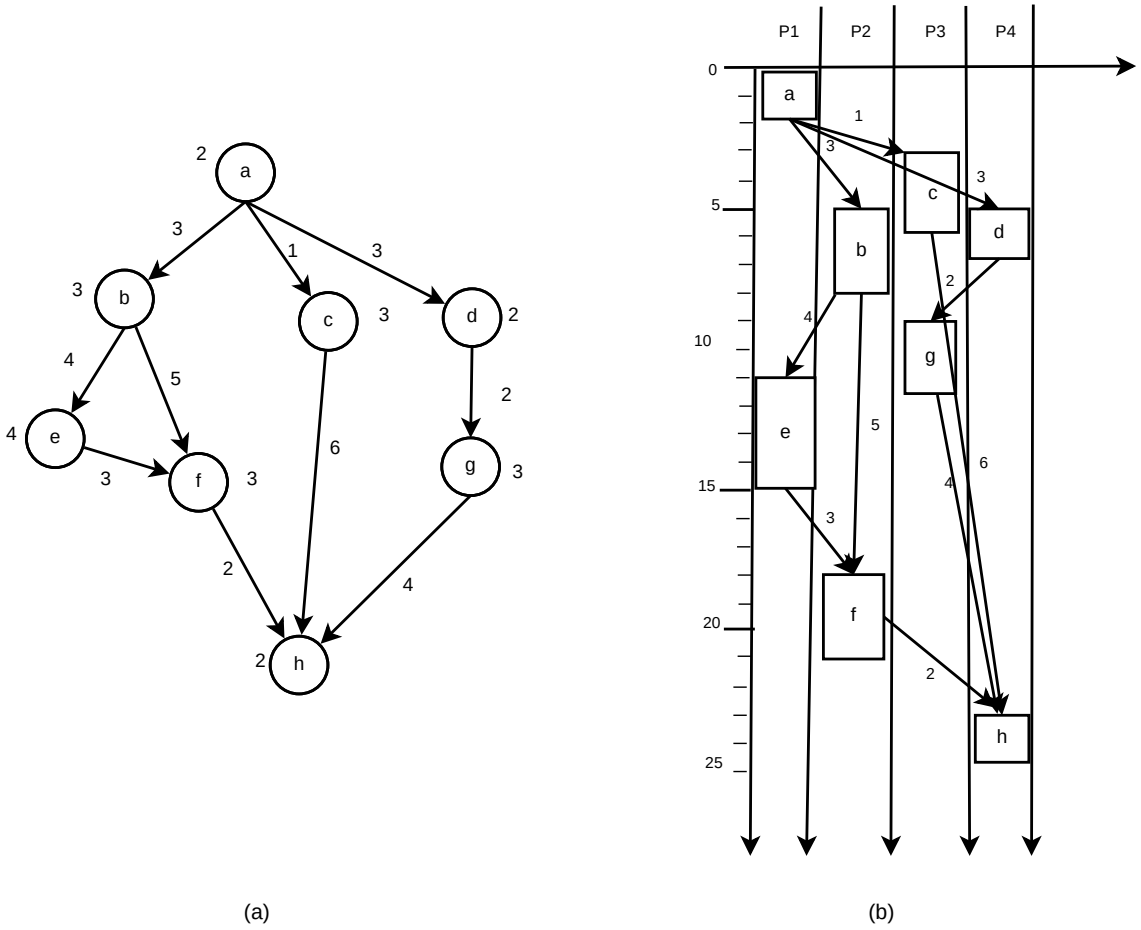

Figure 3.2: Scheduling example in classic model

earliest possible on p_3. So again, in this case, the communication between node a and c is remote and therefore there is a delay of 1 time unit corresponding to the weight of edge e_{ac}. The finish time is $t_f(c) = t_s(c) + w(c) = 6$. When coming to node d, d has the earliest start time $t_s(d, p_4) = 5$, for the reason that d is on a different processor than node a, which makes the communication remote. The finish time is $t_f(d) = t_s(d) + w(d) = 7$. For node e, the start time is $t_s(e) = t_f(b) + w(e_{be}) = 11$ and the finish time is $t_f(e) = t_s(e) + w(e) = 15$. Node f has two incoming edges, e_{ef} and e_{bf}. As node f on a remote processor relates to node e and on a local processor relates to node b, the data ready time of node f is $t_{dr}(f, p_2) = 18$, the start time is $t_s(f) = \max\{t_{dr}(f, p_2), t_f(p_2)\} = 18$ and the node finish time is $t_f(f) = t_s(f) + w(f) = 21$. The same logic applies to h, which has three incoming communication edges e_{fh}, e_{ch}, e_{gh}, so the data ready time is

$t_{dr}(h, p_3) = t_s(h) = 23$ and the node finish time is $t_f(h) = t_s(h) + w(h) = 25$. The final schedule length of this task graph is 25.

3.2 Contention model

Most scheduling algorithms are developed in the idealized classic model, which is not very realistic for modern parallel/distributed systems [SS05]. In a real parallel/distributed system, the inter-processor communication may have resource contention and the communication network may not be fully connected. In order to make task scheduling more than just a theoretical exercise, a more realistic model is necessary. The contention model includes the consideration of communication contention. It has the following properties (summarized from O. Sinnen's book [Sin07] in Chapter 4):

1. Dedicated System. No other program or task is executed on the system while the scheduled task graph is executed.

2. Dedicated Processor. The processor $p \in P$ can execute only one task at a time and the execution is not preemptive.

3. Local communication has zero costs. The communication cost between two tasks in the same processor can be ignored.

4. Communication is performed by a communication subsystem. This communication subsystem may not be fully connected and cannot perform multiple communications in parallel.

The contention model does not have properties 5 and 6 of the classic model, which makes scheduling in the contention model much more complicated than in the classic model. Without property 5 of the classic model, the communication cannot be performed in parallel. For example, when processor p_1 is sending data to processor p_2, a node n in processor p_3, which also needs data from p_1 at the same time, n needs to wait until processor p_1 is free, as there is contention for communication resources between p_1 and p_3. Without property 6 of the classic model, the scheduling algorithm needs to find a scheduling path for every communication action. Due to the fact that the communication processors are not fully connected, not every two processors may connect

directly, so a suitable path needs to be found first for the communication. Figure 3.3 gives an example of the contention model. In this model, the processors are not fully connected and the unidirectional arrows represent the links between the processors. The communication can only be transferred along the direction of the arrows. When any processor needs to communicate with any other processor, a connected route between them needs to be found. For example, when processor p_2 wants to send a data to processor p_4, the only path it can find is path $p = (L2, L3)$. When processor p_1 needs to send a data to processor p_4, there are two paths, $p_1 = (L1, L2, L3)$ and $p_2 = (L4)$, which can be chosen by processor p_1. The scheduling algorithm should choose the best path for every communication in this contention model.

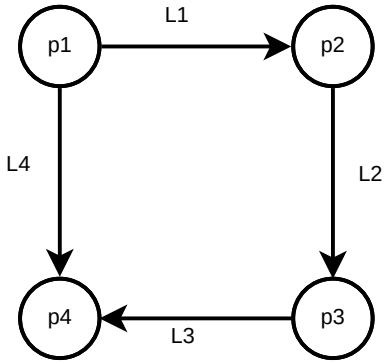

Figure 3.3: Contention model and path finding

3.2.1 One-port model

A contention task scheduling model named the one-port model [BMRR06] is introduced in this section. The task scheduling model assumes a specific parallel/distributed system, which consists of a set of processors connected by a communication network [SS05]. Every node $n_i \in V$ of a task graph $G = (V, E, w, c)$ is executed on a set of processors P and the communications $c(e_{ij})$ between the nodes are transferred between processors via the communication network. A task scheduling model represents a physical target parallel/distributed system. The proposed algorithms in this work are evaluated under the one-port model. The one-port model has the following properties:

1. Each processor has one in-port and one out-port.

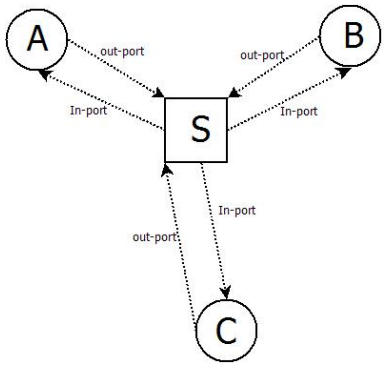

Figure 3.4: One-port model

2. All the processors are connected to a switch.

3. The cost of communication between tasks executed on the same processor, local communication, is negligible and considered to be zero.

4. The cost of communication between tasks executed on different processors, inter-processor communication, may be significant and needs to be considered.

5. Each processor can only execute one task at a time. Each port can only transfer exactly one inter-processor communication at a time too.

6. The switch is assumed to perform all of the communications in parallel without contention.

As illustrated in Figure 3.4, processors A, B and C are connected to a switch S. The directed edge from one processor to a switch represents an out-port and the directed edge from switch S to a processor represents an in-port. The communication between every two processors must pass through the switch S. An one-port model connecting an unlimited number of processors is called the unlimited model in this text. The unlimited model can simplify the complexity of the list scheduling algorithm.

3.2.2 A scheduling example in the one-port model

An example of task graph scheduling in the one-port model is given in Figure 3.5.

A simple procedure was used to create the schedule for Figure 3.5(a), where the node scheduling order is (a,b,c,d,e,f,g,h) and the algorithm allocates the nodes to

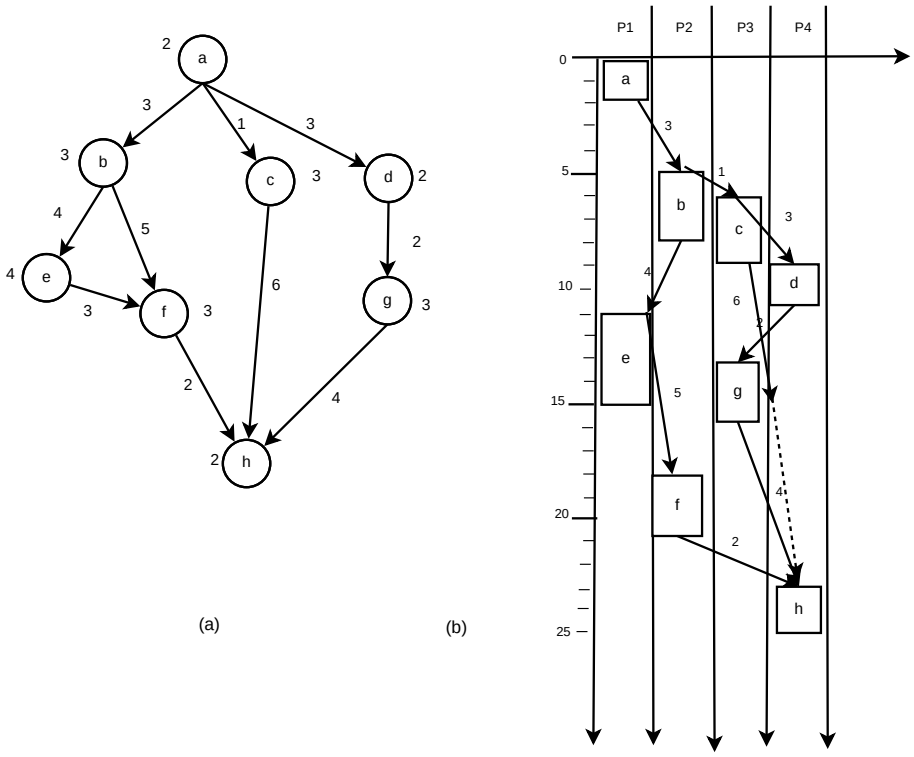

Figure 3.5: Scheduling example in the one-port model

processors in a round robin fashion, starting with p_1. The schedule in Figure 3.5(b) illustrates the execution of nodes in the one-port model and every node is scheduled to be processed by a specific processor by the scheduling algorithm. As seen in the Gantt chart, the start time of node a executed on processor p_1 is $t_s(a) = 0$ and the computation cost $w(a) = 2$, the finish time of node a is $t_f(a) = t_s(a) + w(a) = 2$. Node b needs to receive remote communication data from node a and in order to make node b have the earliest start time, b should be started at $t_s(b) = t_f(e_{ab}, p_1, p_2) = t_f(a) + c(e_{ab}) = 2 + 3 = 5$. The computation cost of node b is $w(b) = 3$ and the finish time of node b is $t_f(b) = t_s(b) + w(b) = 8$. Node c also receives data from node a, the start time $t_s(c) = t_f(e_{ac}, p_1, p_3) = t_f(a) + c(e_{ab}) + c(e_{ac}) = 2 + 3 + 1 = 6$. Note that the communication cost $c(e_{ab})$ must be added. That is because in the one-port model, according to properties 1 and 5, each processor has only one out-port and each port can only transfer one communication at a time, hence the communications e_{ab} and e_{ac} are transferred one by one. So, in this case, the communication between node a and c is

remote and therefore there is a delay of 4 time units corresponding to the weights of edge e_{ab} and e_{ac}. The finish time is $t_f(c) = t_s(c) + w(c) = 9$. Node d has the earliest start time is $t_s(d) = t_f(a) + c(e_{ab}) + c(e_{ac}) + c(e_{ad}) = 2+3+1+3 = 9$, as the three communications e_{ab}, e_{ac} and e_{ad} are serialized through the out-port of p_1. The finish time is $t_f(d) = t_s(d) + w(d) = 11$. For node e, the start time $t_s(e) = t_f(b) + w(e_{be}) = 11$ and the finish time is $t_f(e) = t_s(e) + w(e) = 15$. Node f has two incoming edges e_{ef}, e_{bf}. As node f is in a remote processor relative to node e and in a local processor relative to node b, the data ready time of node f is $t_{dr}(f, p_2) = 18$, the start time is $t_s(f) = \max\{t_{dr}(f, p_2), t_f(p_2)\} = 18$ and the node finish time is $t_f(f) = t_s(f) + w(f) = 21$. The same is true for node h, which has three incoming communications from edges e_{fh}, e_{ch}, e_{gh}, so the data ready time is $t_{dr}(h, p_3) = t_s(h) = 23$ and the node finish time is $t_f(h) = t_s(h) + w(h) = 25$. The final schedule length of this task graph is 25.

Compared with the example in Section 3.1.1, even though these two examples have the same schedule length 25, there are many differences between the classic model and the one-port model. In the classic model, node c has a start time of $t_s(c) = t_f(e_{ac}, p_1, p_3) = t_f(a) + c(e_{ac}) = 2 + 1 = 3$. In the one-port model, the start time of node c is $t_s(c) = t_f(e_{ac}, p_1, p_3) = t_f(a) + c(e_{ab}) + c(e_{ac}) = 2+3+1 = 6$. That is because in the one-port model, the out-port of processor p_1 can only send one communication at any one time. When p_1 wants to send data $c(e_{ac})$ to p_3, it needs to wait for the out-port to finish sending data $c(e_{ab})$; the same is true for the start time of node d. Generally speaking, communication contention will increase the start time of some nodes and make the schedule length longer than the schedule length obtained with the classic model, which ignores communication contention altogether.

Chapter 4

Existing scheduling approaches

4.1 List scheduling

List scheduling is one of the most popular scheduling heuristics for task scheduling. A good list scheduling algorithm can have low complexity and produce a short schedule length [THW02]. Generally speaking, a list scheduling algorithm will work iteratively in a loop. In each step of the loop, it will choose a node from the set of unscheduled nodes by the priority order; then, it will schedule this node according to a specific algorithm. The loop will not stop until all the nodes of the task graph are scheduled. The most important problems are how to determine the priorities of the nodes and how to schedule these nodes onto the processor in order to achieve a good scheduling result.

Normally, there are two kinds of list scheduling algorithms that are distinguished by the static and dynamic node priorities [SS04]. Static priorities mean all the node's priorities are determined by list scheduling at the same time and cannot be changed after the priorities are determined. The list scheduling algorithms which use the static priorities are called static list scheduling algorithms. Dynamic priorities mean the node's priorities are not fixed. In each step of the loop, the list scheduling algorithm will recalculate the priorities and make decisions by the new priorities. The list scheduling algorithms which use dynamic priorities are called dynamic list scheduling algorithms. Obviously, static priority algorithms have lower complexity and are much simpler than dynamic priority algorithms, but a good dynamic priority algorithm may result in a shorter scheduling length. Dynamic scheduling is often used when the computation or

communication costs are uncertain. This means the algorithm does not know exactly what the computation or communication cost is and cannot give accurate priorities to the nodes. Most of the dynamic scheduling algorithms change the priority list each time before a node is scheduled. Sometimes, dynamic scheduling may also be used when the system structure is unstable, for instance, because some virtual machines change.

4.1.1 Static list scheduling algorithms

A static list scheduling algorithm first assigns a priority to all the nodes $n_i \in V$. Then it sorts the nodes into a list L by their priorities. The static list scheduling algorithm will maintain a loop. In each step of the loop, the algorithm will choose a node $n_i \in L$ with the highest priority and schedule each node n_i on a processor p_i in the sorted order. The loop will be repeated until all the nodes $n_i \in L$ are scheduled. The process by which the algorithm chooses a processor p_i for node n_i, is called allocation of processor p_i for node n_i. The process by which a node n_i allocated to a processor p_i is given a start time $t_s(n_i, p_i)$ and a finish time $t_f(n_i, p_i)$ is called scheduling n_i on processor p_i. The priority order of the nodes is usually in non-ascending order and must satisfy the precedence constraints. The basic steps of static list scheduling are given below [SS04]:

1. Sort all nodes $n_i \in V$ into list L, according to a priority scheme and precedence constraints.
2. For each $n \in L$ do:
   (a) Choose a processor $p \in P$ for n.
   (b) Schedule n on P.

4.1.2 Dynamic list scheduling algorithms

A dynamic list scheduling algorithm is different from a static priority algorithm. In a dynamic list scheduling algorithm, the algorithm assigns each node $n_i \in V$ a priority in each step of the loop, chooses the node n with the highest priority, and then schedules this node n. In the next step of the loop, the algorithm will recalculate the priorities of all the unscheduled nodes n_{unsch}, then choose another node n' with the highest priority and schedule n'. This loop will repeat the same operation until all the nodes are scheduled.

In each loop iteration, one task will be scheduled and then the next loop iteration is processed. The priority scheme may change at the beginning of each loop iteration, and the reason why the priority will change is the dynamic algorithm chooses the best priority scheme according to the situation. The basic steps of dynamic list scheduling can be seen as follows:

1. If there are unscheduled nodes then do:

   (a) Sort all unscheduled nodes $n_i \in V$ into a list L, according to a priority scheme and precedence constraints.

   (b) Choose the node $n \in L$ with the highest priority.

   (c) Choose a processor $p \in P$ for n.

   (d) Schedule n on P and go back to the beginning of the loop.

4.1.3 List scheduling in the classic model

For the classic model, there are many different proposed list algorithms in the literature. As discussed in Section 3.1, the classic model has various properties. A good algorithm should make good use of these properties, to achieve low complexity and a short schedule length. The most important property is that the local communication has zero communication cost. When considering the factors that affect the whole length of the schedule, one of the most important factors is the communication cost between the nodes, especially in the one-port model. A task graph with a higher communication cost will often have a longer scheduling length, because the nodes need to spend more time waiting for the communication data transfer from another processor. An example of this is given in Figure 4.1.

In Figure 4.1, graph 1 and graph 2 have the same structure and are scheduled by the same algorithm. The schedule S of graph 1 and graph 2 indicates that nodes a, b are allocated to processor p_1 and node c is allocated to processor p_2. As shown in Figure 4.1 (b), even though graph 1 and 2 have the same structure and the same scheduling algorithm is used, they still have a different schedule length. Nodes a and b have the same start time and finish time, but node c in graph 2 needs to wait more time to get the data from node a, which makes the schedule length much longer than that of graph 1. So,

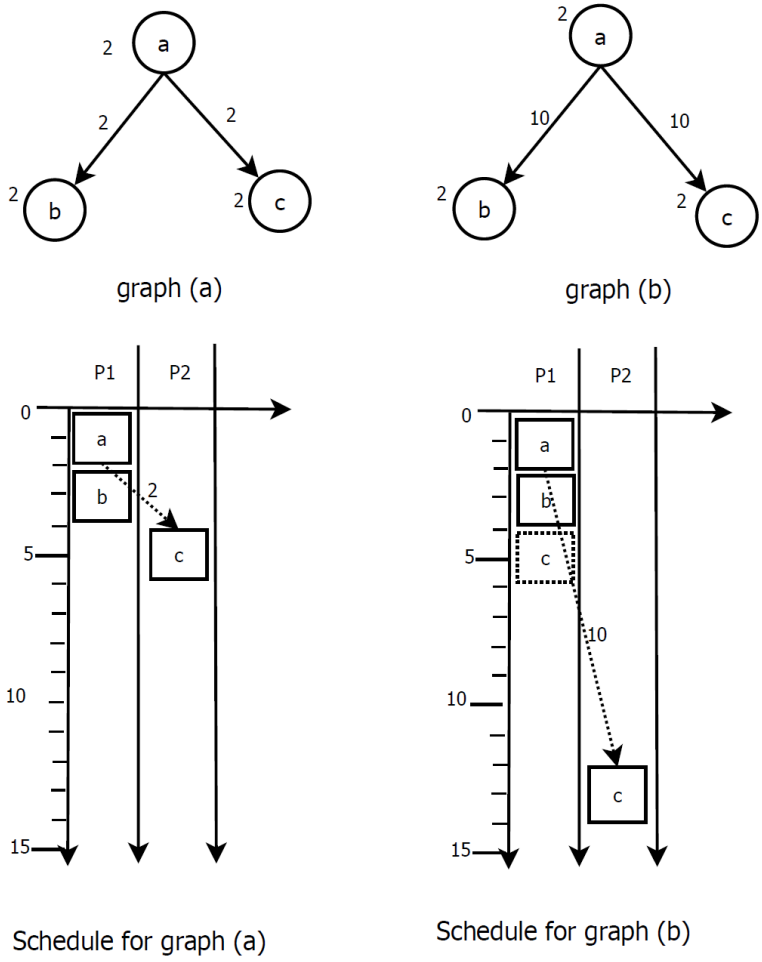

Figure 4.1: An example to illustrate how communication cost may affect the schedule length.

heavier communication cost can expand the schedule length. Studying how to reduce this waiting cost is a good strategy to reduce the whole schedule length.

In the given example, the scheduling algorithm should aim to allocate node c on processor p_1 and the schedule length of graph 2 will be reduced from 14 to 6. That is because if node c is allocated to processor p_1 (the same processor with its parent node a), node c does not need to wait for the data from node a, as the cost is assumed to be zero for local communication. If the proposed list scheduling algorithm can make full

use of this property in the classic model, the whole scheduling length can be reduced.

No matter whether someone makes use of dynamic list scheduling or static list scheduling, how to get the priorities of the nodes and in what way to choose a processor for a node are the most important questions that an algorithm needs to address. Many suggested ways of finding good priorities can be found in the literature, based on the critical path (see Section 2.1), node level (see Section 2.1), dominant sequence (see Section 2.1) and communication cost. Because a heavier communication cost can expand the schedule length, the nodes with a heavier communication cost are more likely to affect the whole schedule length. Usually, the nodes incurring a heavier incoming communication cost are given higher priority in order to boost their priority and schedule them early compared to other nodes. Based on this view, the priority of each node $n_i \in V$ is calculated by:

$$\frac{\sum_{j \in pred(i)} c(e_{ji})}{w_i},$$

where j is the number of n_i's predecessor nodes and the edges e_{ji} ($j \in pred(i)$) are the incoming edges of node n_i. Please note that this formula summarizes the relevant literature, it is not a new contribution of this thesis. The priority order of the nodes is in descending order; if two nodes have the same priority, they are ordered by random and the order is called non-ascending order. The priority order must satisfy the precedence constraints (see Section 2.3).

When it comes to the processor choice, the processor p_i that allows node n_i to have the earliest start time will be chosen. In order to find this processor p_i, the algorithm should calculate the start time $t_s(n_i)$ on every processor and return the specific processor p_i allowing the earliest start time of node n_i. Algorithm 4.2 shows how to choose a processor for a free node n_i. In Algorithm 4.2, $t_s(n_i)$ and $proc(n_i)$ mean the start time and the allocated processor of n_i. Every time the algorithm checks a processor, the start time and finish time of all the processors should go back to the state where Algorithm 4.2 begins. That is because Algorithm 4.2 is only used to choose a processor for node n_i and it will not change the state of any other nodes and processors. Algorithm 4.1 is used in Algorithm 4.2 to help Algorithm 4.2 get the data ready time. Please note that all the algorithms in Chapter 4, including Algorithm 4.1 to Algorithm 4.6, are summarized from the literature, they are not new contributions of this thesis.

Algorithm 4.1 Get $t_{dr}(n_i, p)$, the DRT of n_i on processor p,

```
1: Input: node n_i, processor p
2: Output: data ready time of n_i on p
3: Find all incoming edges Incom{n_i} of n_i
4: DRT ← 0
5: for all edge e_ji ∈ Incom{n_i} do
6:    schedule edge e_ji
7:    if edge e_ji's finish time t_f(e_ji, proc(n_j), p) > DRT then
8:       DRT=t_f(e_ji, proc(n_j), p)
9:    end if
10: end for
```

Based on the previous discussion, a generic static list scheduling algorithm is given to show how to schedule the nodes in classic model. The priority of each node $n_i \in V$ is calculated by the formula:

$$\frac{\sum_{j \in pred(i)} c(e_{ji})}{w(n_i)},$$

where w_i is the weight of the node n_i and edge e_{ji} is the incoming edge of node n_i and satisfies the precedence constraints. Once again, this formula results from the literature, it is not a new contribution of this thesis. The allocated processor is chosen by Algorithm 4.2 and the data ready time (DRT, see Section 2.2) is calculated by Algorithm 4.1.

As shown in Algorithm 4.3, the given algorithm first calculates the priorities by a specific formula and then sorts the nodes by the priority order. Then the algorithm will schedule each node using Algorithm 4.2 and Algorithm 4.1. The start time of node n_i on a processor p_i is the later one among the data ready time DRT and processor finish time $t_f(p_i)$. After scheduling n_i on processor p_i, n_i will be given a start time $t_s(n_i, p_i)$ and a finish time $t_f(n_i, p_i) = t_s(n_i, p_i) + w(n_i)$ and the finish time of $t_f(p_i)$ will be modified to $t_f(p_i) = t_f(n_i, p_i)$. This algorithm can make the nodes with heavier incoming communication costs have a higher priority. To do so, it considers the effect of communication cost on the whole scheduling and because it chooses the processor for each node by the earliest start time first, the property of local communication cost is zero (see Section 3.1 property 3). It is considered that this algorithm can reduce the

Algorithm 4.2 Choose a processor p_i for node n_i (classic model)

1: Input: node n_i
2: Output: processor p_i for node n_i
3: $t_{temp} \leftarrow \infty$, $p_{temp} \leftarrow null$.
4: **for all** processors $p_i \in P$ **do**
5: **if** $t_{temp} > \max(t_{dr}(n_i, p_i), t_f(p_i))$ **then**
6: use Algorithm 4.1 to get $t_{dr}(n_i, p_i)$
7: $t_{temp} \leftarrow \max(t_{dr}(n_i, p_i), t_f(p_i)$
8: $p_{temp} \leftarrow p_i$
9: **end if**
10: **end for**
11: $t_s(n_i) \leftarrow t_{temp}$, $proc(n_i) \leftarrow p_{temp}$

whole schedule length.

4.1.4 Some list scheduling approaches

This section examines some list scheduling approaches from the literature. This thesis will later propose some new list scheduling algorithms. The detailed differences and contributions of the proposed algorithms compared to the following approaches will be discussed in Chapter 5, Chapter 6, and Chapter 7.

HEFT and CPOP [THW02]: HEFT is a typical list scheduling algorithm. HEFT can have both low-complexity and good performance. As HEFT is targeting heterogeneous environments, the computing cost for one node is different in different machines. Thus, one problem is how to select an appropriate weight for each node. HEFT uses the average value and the upward ranking to calculate the priorities of nodes. There are two phases in HEFT, which correspond to the two steps of list scheduling heuristics. The first is the task prioritizing phase. Every task n_i is given a priority by the upward rank value, $rank_u(n_i)$. Tasks are sorted in a task list in decreasing order of priority. When two tasks have the same upward value, tie-breaking is done randomly. The second phase is the processor selection phase. Processor selection is the most important part of HEFT. Different from some other list algorithms which select the processor by the earliest start time,

Algorithm 4.3 List scheduling (classic model)

1: Input: a task graph $G = (V, E, w, c)$
2: Output: a schedule for all nodes $n_i \in V$
3: Calculate the priority of each node n_i as $\sum_{j \in pred(i)} c(e_{j,i})/w_i$
4: Order all nodes $n_i \in V$ into list L by priority and precedence constraints (non ascending order)
5: **for all** node $n_i \in L$ **do**
6: Choose a processor p_i for node n_i (Algorithm 4.2)
7: Schedule n_i on processor p_i
8: Finish time $t_f(p_i) \leftarrow t_s(n_i, p_i)+w(n_i)$
9: **end for**

HEFT chooses the processor by the earliest finish time. This makes HEFT more effective in heterogeneous environments. There is also an insertion-based policy to help HEFT selecting processors. As HEFT is based on the list scheduling, it is as low-complexity as any list scheduling heuristic. The original HEFT paper also proposes a critical-path-on-a-processor (CPOP) algorithm. CPOP finds all the tasks in a critical path and schedules these tasks in one processor. CPOP tries to use the critical path to reduce the scheduling length. HEFT is used in a heterogeneous environment (distributed computing) which is different from a homogeneous environment (parallel computing).

HBMCT [SZ04a] : HBMCT uses a hybrid heuristic also targeting heterogeneous environments. Three phases are included: ranking, group creation and scheduling independent tasks. In the first phase, each node is given a rank value by the upward ranking. Nodes are sorted in descending order of priority. In the second phase, nodes are divided into groups. The nodes in one group are independent of each other. In the third phase, any heuristic for scheduling the independent tasks can be used to schedule the nodes of the groups in the second phase. The original paper proposes a balanced minimum completion time (BMCT) heuristic to schedule the independent nodes. The basic ideas of BMCT are: schedule a group, then find the maximum finish time machine m; if moving any task n_i in machine m to machine n can reduce the makespan, then reallocate this task to machine n; do

this until no better machine is found then schedule the next group. HBMCT uses a level-based idea to split the nodes into groups, then uses the earliest start time to schedule each node, finally, it uses a balance-swap idea to reduce the makespan.

DLS [SL93]: Dynamic level scheduling is a compile-time scheduling heuristic. List scheduling does two things at each step: choose a node n_i then choose a processor p_i for n_i. DLS chooses one node n_i and a processor p_i for n_i that make $DL(n_i, p_i)$ have a max value. $DL(n_i, p_i)$ is defined by $DL(n_i, p_i) = SL(n_i) - EST(n_i, p_i)$, where $SL(n_i)$ means the static level of node n_i (the upward rank of n_i) and $EST(n_i, p_i)$ is the earliest start time of n_i on p_i. $DL(n_i, p_i)$ shows how well node n_i matches p_i. Note that this is a dynamic priority list scheduling and it is different from level-base algorithms.

FCP [RvG99]: FCP stands for fast critical path. FCP is also a kind of a list scheduling algorithm. The difference is in the first phase of FCP, it only sorts a constant size of ready tasks in a priority-list, other ready nodes are added in a FIFO-queue. In the second phase of FCP, instead of considering all processors as possible targets for a given task, FCP only chooses between two processors: the one from which the last messages to the given task arrive and the one that becomes idle at the earliest.

LMT [IÖF95]: Levelized min-time is a two-phase algorithm. The first phase groups the tasks that can be executed in parallel using the level attribute (the upward value or bottom level). The second phase assigns each task to the fastest available processor (the processor that minimizes the sum of the node's computation cost and the total communication costs with nodes in the previous levels). The nodes at lower level have higher priority than nodes at higher level. Within the same level, the nodes are sorted by the computation cost.

Rescheduling [SZ04b]: This is a run-time scheduling policy. It suggests a rescheduling policy to reduce the actual makespan. This policy tries to reduce the time of rescheduling while optimising the makespan. The interesting parts are the definitions of spare time and slack time, which may be used by other algorithms. A $delay = RST - EST$ is defined where RST is the real start time and EST is the

static earliest start time that any algorithms statically estimated before rescheduling. If one node has $delay > slack$ or $spare$, the remaining non-executed nodes can be rescheduled.

Weight [ZS03]: An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. It investigates the effects on the makespan of 6 different ways to calculate the weight of a node and 2 different ways to calculate the priorities (upward ranking and downward ranking). The interesting part is the 6 different ways to calculate the weight: (i) Mean value (M), the average over all the input arguments including average communication cost and average communication cost; (ii) Median value (ME), median value over all the input arguments; (iii) Worst value (W), the worst value (i.e., maximum computation cost and the communication cost between the two machines on which each of the two communicating tasks has its highest computation cost); (iv) Best value (B), the best value (i.e., minimum computation cost and the communication cost as determined by the procedure described previously); (v) Simple worst value (SW), the worst value for both computation cost and communication cost; (vi) Simple best value (SB), the minimum value for both computation cost and communication cost.

Multiple DAGs [ZS06]: the method focuses on scheduling more than one DAG at the same time and trying to make sure that every DAG has a fair execution time. The idea is based on merging all the DAGs into a single bigger composite DAG. Then, this DAG will be scheduled by a conventional DAG scheduling algorithm. In order to keep fairness, two policies are investigated. One fairness policy is based on finish time and the other is based on current time. Different definitions of fairness are proposed, such as: slowdown, unfairness, average slowdown.

4.2 Clustering scheduling

Clustering is another important heuristic for task scheduling [GY92a, YG94, GY93]. With list scheduling, the tasks are always allocated and scheduled one by one, based on the priorities and the given scheduling algorithm. Determining the order of scheduling each task in list scheduling may have some good or bad effects on other tasks, something

that will affect the final schedule length. When a list algorithm schedules a task or node, it needs to consider all the effects on the other nodes and make a good decision to get the shortest schedule length or a schedule of lower complexity. Unfortunately, as scheduling is an NP-hard problem, no existing list scheduling algorithm can do this perfectly. Clustering gives a different way of finding a solution to the task scheduling problem. Task scheduling allocates every node of a processor and schedules these nodes to the processors. When analyzing these allocations and schedules, it is easy to find that some nodes are allocated and scheduled on the same processor. This is because the algorithm thinks it can get a better result (shorter schedule length, lower complexity or something else) by placing them together on the same processor. Clustering is developed based on this idea: the nodes can be divided into different groups and all the nodes in one group should be scheduled on the same processor. So, how to determine these groups is the main challenge that clustering needs to accomplish.

4.2.1 Generic clustering algorithms

Clustering Let $G = (V, E, w, c)$ be a task graph. A clustering $C = (p_0, p_1, p_2, p_3, ...p_{|V|})$ is a schedule of G in a parallel/distributed system P. This schedule divides all the nodes into different groups. Each group belongs to one processor p. The number of groups $|C|$ is the same as the number of processors $|V|$.

Clusters Let $G = (V, E, w, c)$ be a task graph. A clustering $C = (p_0, p_1, p_2, p_3, ...p_{|V|})$ is a schedule of G in a parallel/distributed system P. Each element p of $C = (p_0, p_1, p_2, p_3, ...p_{|V|})$ is called a cluster c. All the processors $p \in C$ are called clusters of C.

Clustering determines which nodes should always be scheduled into the same cluster. In order to obtain this information, clustering scheduling uses an edge zeroing technique, which means when two nodes n_i and n_j are determined to be in the same cluster p, then the edge weight $c(e_{ij})$ between them is zeroed. The basic principle of the edge zeroing technique is that if zeroing this edge e_{ij} will not increase the whole scheduling length sl, then this operation is accepted by the clustering algorithm and the two nodes n_i and n_j connected by this edge e_{ij} are placed in the same cluster p. If n_i and n_j are in two different clusters p_i and p_j, p_i and p_j will be combined into one cluster p_i by putting

all the nodes of cluster p_j into cluster p_i. From this basic principle, it is easy to see that the core problem of the technique is how to find the scheduling length *sl* before and after the algorithm and to ascertain whether an edge should be zeroed. Unfortunately, it is hard for a clustering scheduling algorithm to get the scheduling length *sl* directly in a contention model. In a clustering algorithm, if an edge is determined to be zeroed, the process of how to zero an edge is implemented by a function $f_{zeroing}(e_{ij})$. Function $f_{zeroing}(e_{ij})$ tries to zero the edge e_{ij} and returns a new clustering for the task graph $G = (V, E, w, c)$.

The three generic steps of a clustering algorithm [Sin07] are as follows:

1. Find a clustering C of G (using the basic principle, described in the previous paragraph, and specific algorithm. This step is done in the unlimited processors virtual system, that is $|C| = |V|$).

2. Mapping clusters on physical processors (each cluster assigned to an independent processor. This step is done in the limited reality processors system P, that is $|C| \geq |P|$). The limited processors system means the number of processors P is limited or finite.

   (a) Sorting clusters C in non-descending order of the total computation weight $w(C)$;

   (b) for *i*=1 to $|C|$ do

   (c) Mapping clusters C_i to processor $P_{imod|P|}$; where$|P|$ means the number of processors (limited reality processors system), $imod|P|$ means the modulus (%) operation of taking the modulo of i against $|P|$.

3. Schedule nodes on each processor (determine the scheduling order of the nodes and decide the start and finish times).

In Algorithm 4.5, firstly, the algorithm will create an initial clustering C_0 for the task graph G. Each node n_i belongs to a separated cluster p_i in C_0. Secondly, the algorithm will try to check all the edges (the order of the edges can be decided by any specific algorithm, for example, ordering the edges by the communication cost). For each edge e_{ij}, Algorithm 4.5 will try to zero it, then get the new clustering C_{i+1}. If the schedule

Algorithm 4.4 Function $f_{zeroing}(e_{ij})$

1: Input: an edge $e_{(ij)}$
2: Output: new clustering C_{i+1} and the edges that should be zeroed.
3: Let $e_{(ij)}$ be an edge to be checked
4: **if** $proc(n_i) = proc(n_j)$ **then**
5: C_{i+1}=C_i
6: **else**
7: Create new clustering C_{i+1}: merge clusters $proc(n_i)$ and $proc(n_j)$ into one cluster and zero all the edges between cluster $proc(n_i)$ and $proc(n_j)$
8: **end if**

length $sl(C_{i+1}) \leq sl(C_i)$, then this zeroing operation for e_{ij} is accepted and C_{i+1} will become the new clustering. If not, edge e_{ij} will remain the same, nothing will be done by the algorithm and the clustering should go back to C_i. The algorithm will go on until all the edges are checked and the final clustering will be the result of step one. In step two (see the above three generic steps), a simple load balancing scheme is used to map the clusters to the physical processors. The total computation weight $w(C) = \sum_{n \in C} w(n)$ of each cluster is calculated and mapped to the processors in non-descending order, which can make every processor have a balanced computing load. In step two, every node already has an allocated processor. The algorithm will schedule these nodes on the allocated processors by the node's priorities. The priorities of the nodes can be calculated by the formula:

$$\frac{\sum_{j \in pred(i)} c(e_{ji})}{w_i},$$

where w_i is the weight of and e_{ji} is the incoming edge of node n_i. Please note that this formula results from the literature, it is not a new contribution of this thesis. Algorithm 4.6 shows the process in step 3.

4.2.2 Clustering algorithms in the classic/one-port model

As most parts of the clustering algorithms for both the classic model and the one-port model are the same except for the scheduling of the edges, clustering algorithms in the one-port model are introduced in what follows. One of the most important problems is

Algorithm 4.5 Generic clustering algorithm (step 1)

1: Input: a task graph $G = (V, E, w, c)$
2: Output: new clustering C
3: Create initial clustering C_0, each node belongs to one separated and distinct cluster $p \in C_0$
4: $i \leftarrow 0$
5: **for all** edge $e_{ij} \in E$ **do**
6: call $f_{zeroing}(e_{ij})$
7: calculate new clustering C_{i+1}
8: **if** $sl(C_{i+1}) > sl(C_i)$ **then**
9: $C_{i+1} \leftarrow C_i$
10: **end if**
11: $i \leftarrow i + 1$
12: **end for**

how to get the schedule length $sl(C_i)$ and $sl(C_{i+1})$, especially under the contention/one-port model, because the algorithm cannot know the whole schedule length until all the nodes are scheduled. In reality, the schedule length of a clustering is often replaced by the critical path or dominant sequence. This is due to the fact that, once the algorithm zeroes an edge e_{ij}, it indicates putting the outgoing node n_j on the same processor $proc(n_i)$ as its parent node n_i, so that the local communication $w(e_{ij})$ can be zeroed. When the algorithm has a clustering, it has already known which nodes are allocated to the local processor. In this situation, the algorithm needs only to find the critical path or dominant sequence, which can represent the approximate schedule length in the classic model. Indeed, the critical path or dominant sequence is approximate and makes sense only in the classic model. The critical path and dominant sequence cannot represent the schedule length in the contention/one-port model. In order to get the schedule length $sl(C_i)$ and $sl(C_{i+1})$, the critical path is chosen as the approximate schedule length. The critical path can be found by a depth first search (DFS) algorithm [Tar72].

Algorithm 4.6 Clustering algorithm (step 3)

1: Input: a task graph $G = (V, E, w, c)$
2: Output: a schedule of all nodes $n_i \in V$
3: Calculate the priority of each node n_i as $\sum_{j \in pred(i)} c(e_{ji})/w_i$
4: Order all nodes $n_i \in V$ into list L by priority and precedence constraints (non-ascending order)
5: **for all** node $n_i \in L$ **do**
6: Get n_i's allocated processor p_i
7: Get data ready time (DRT) of n_i on p_i (Algorithm 4.5)
8: Get finish time $t_f(p_i)$ of p_i
9: Get start time $t_s(n_i, p_i)$=max(DRT,$t_f(p_i)$)
10: Schedule n_i on processor p_i
11: **end for**

4.3 Other scheduling algorithms

Other than the above, there are also some additional approaches for parallel task scheduling. These approaches use some ideas that differ from standard list scheduling and clustering scheduling and offer some new perspectives to carry out scheduling, which can be useful in appreciating the objectives of this thesis. Some of the approaches are presented below:

Duplication-based scheduling [AK98, CC91, DA97, LLLW06, PC01, RA00, STK09, ZB97]. Duplication-based scheduling algorithms copy some tasks into some processors when the processors sit idle. By copying a task, this task's children nodes do not need to receive data from it if the children nodes are allocated on the same processor with it. Copying can avoid the communication transfer between the processors, which may reduce the start time of the children nodes. Note that some systems do not support the duplication mechanism and, at the same time, duplicating the nodes may make the load of systems heavier. If the focus is to reduce the scheduling length and the duplication mechanism is supported by the systems, duplication-based scheduling can be a good strategy.

AVL-tree-based cloud computing [CHJC14]: task scheduling based on load approximation in real-time average cloud computing environment. The tasks provided by the users are ranked by a reciprocal pairwise comparison matrix and the analytic hierarchy process (AHP). An AVL tree is a height-balanced tree to make the binary trees as flat as possible. Each device is logically connected based on the rule of the binary tree and the comparison factor of the binary tree is the available load of devices. The device's left devices' load is smaller than the device's load and the device's right devices' load is larger than the device's load. By using the AVL binary tree approach, the tree operation's time complexity is $O(logn)$ in terms of the add operation, delete operation and resource allocation of the cloud device. Concerning task priority, the basic concept is to give tasks for which the requested load is closest to the average load of the cloud environment a high execution priority. If no cloud device can provide the requested load for the selected task, the next task with the closest current cloud load is selected to execute. In terms of device selection, the device whose average system load with the new task is closest to the average system load before assigning the new task will be selected.

The analytic hierarchy process [EKP+13]: task scheduling and resource allocation in a cloud computing environment. This is task-oriented resource allocation in a cloud computing environment. A task-oriented resource allocation model using the analytic hierarchy process is proposed. An induced bias matrix (IBM) only based on the original inconsistent comparison matrix is proposed to identify the inconsistent elements and improve the consistency ratio (CR) when the CR is more than 0.1. In the proposed framework, computing tasks are collected in the task pool. Tasks are ranked and submitted to computing resources distributed in cloud computing nodes. The computing resources are allocated according to the weights of tasks. The proposed framework will be further illustrated in the following section. The tasks are scheduled by resources such as network bandwidth, completion time, task costs, reliability, and so on. When the cloud computing service providers receive the tasks from users, the tasks can be pairwise compared using the comparison matrix technique. The cloud computing providers negotiate with the users on the requirements of tasks including network bandwidth,

completion time, task costs, and reliability of task.

Bayesian optimization algorithm [YXP+11]: task scheduling using Bayesian optimization algorithm for heterogeneous computing environments. This research presents a novel scheduling algorithm based on the Bayesian optimization algorithm (BOA) for heterogeneous computing environments. This algorithm is divided into two phases. First, a Bayesian algorithm is used to get population of assignments; second, list scheduling is used to get the makespan. After the first phase, each node has a processor assigned and then list scheduling is used to order the nodes and schedule each node on its assigned processor. The Bayesian algorithm works as follows:

- Generate initial population of assignments, $P(t = 0)$, randomly.
- Choose candidate solutions $S(t)$ from $P(t)$. (step 2)
- Learn the Bayesian network and get the conditional probability tables (CPTs). For a DAG, the DAG is an initialization Bayesian network. Learning the Bayesian network will add some dependencies between sibling nodes. As a result, all the sibling pairs will be checked for whether there should be an extra edge between them. The BD (bayesian-dirichlet) metric is chosen to determine the sibling edges. In this way, the Bayesian network can be obtained.
- Sampling of Bayesian network. Firstly, the algorithm computes an ancestral ordering of the nodes, where each node is preceded by its parents. Then, the values of all variables in a new candidate solution are generated based on the corresponding CPTs in the network under the ordering computed.
- Incorporate a new population into $P(t)$.
- Repeat step 2, until the average makespan of population is not improved.
- Select the best solution from $P(t)$.

Instruction Set extensions [ANF12]: instruction set architecture extensions for a dynamic task scheduling unit. This is heterogeneous multiprocessor system-on-chip task scheduling. It extends the instruction set architecture to improve performance

for dynamic data dependency checking, task scheduling, processing element (PE) allocation and data transfer management. The CellSs programming model can get task level parallelism. The dynamic task scheduler can be implemented in hardware as an accelerator or in software, running on a general purpose core. Checking the data dependencies at run time is the most time consuming part of the software approach (for a dynamic task scheduler). This paper extends the instruction set of CoreManager. The added instructions belong to the large instruction word (VLIW), single instruction multiple data (SIMD), data dependency checking. The results, after adding the SIMD and VLIW, show a reduction of up to 97% is achieved.

Load balanced [KA11]: load balanced min-min algorithm for static meta-task scheduling in grid computing. This approach proposes a load balanced min-min (LBMM) algorithm. Compared to the traditional min-min algorithm, LBMM can reduce the makespan, achieve load balance and increase resource utilization. First, the algorithm schedules the tasks using the min-min algorithm (makespan is ms_1). Then it reschedules some of the tasks to reduce the makespan and achieve the load balance. For rescheduling, firstly, the algorithm chooses the heaviest load resource R_i, and finds the minimum execution time task t_i. Secondly, it finds another resource R_i which caused time t_i to have the maximum finish time $f_{max}(t_i)$; if $f_{max}(t_i) < ms_1$, it reschedules t_i in resource, else it finds the next maximum finish time resource for t_i. Thirdly, it repeats these steps until the shortest makespan is found.

Chemical reaction optimization [XLL11]: chemical reaction optimization for task scheduling in grid computing. Several versions of the Chemical Reaction Optimization (CRO) algorithm have been proposed for the grid scheduling problem. Simulation results show that the CRO methods generally perform better than other existing methods and performance improvement is especially significant in large-scale applications. A grid usually consists of five parts: clients, the global and local grid resource brokers (GGRB and LGRB), grid information server (GIS), and resource nodes. Clients register their requests for processing their computational tasks at GGRB. Resource nodes register their donated resources at LGRB and process clients' tasks according to the instructions from LGRB. In practice, a

client and a resource node can be on the same computer. GIS collects the resource information from all LGRBs, and transfers it to GGRB. GGRB is responsible for scheduling. A solution to task scheduling is looked as a molecule. Many solutions are a set of molecules. These solutions are represented in two ways (permutation-based and vector-based). These molecules will have collisions. During the collision different operators will be done, which corresponds to on-wall ineffective collision, intermolecular collision, synthesis, molecular decomposition; then a new molecule is obtained, which means a new solution is obtained. These reactions last until there is a satisfactory solution.

Priority-Based scheduling [ZTS15]: priority-based scheduling heuristic to maximize parallelism of ready tasks for DAG applications. This approach proposes a priority-based scheduling heuristic (PB) for just-in-time scheduling. It aims at maximizing the parallelism of ready tasks during the execution of DAG application so as to minimize the makespan. The main objective is to get as many ready tasks as possible. In order to get this, the PB algorithm gives every node a priority by the principle of DQ>LQ>EQ. DQ: Direct Quotient, the number of tasks which become ready immediately after the completion of the current node. LQ: Level Quotient, the maximum length from a node to the exit node. EQ: Export Quotient, to distinguish between nodes with the same DQ and LQ. The node with the highest DQ has the highest priority; if two nodes have the same DQ, the one which has the highest LQ will have higher priority; when the LQ is the same, use the EQ.

4.4 Summary

This chapter has reviewed list scheduling and a number of task scheduling algorithms from the literature. The next chapter will propose a new task scheduling algorithm with good performance characteristics.

Chapter 5

A look-forward algorithm for task scheduling

The task scheduling problem for heterogeneous systems is focused on achieving a minimal scheduling length with a reasonable algorithm execution time. Even though thousands of algorithms have been proposed to study this NP-complete problem [Ull75], there is still much room for improvement in an innovative way. In this chapter, a novel predictive, decision-making algorithm called the look-forward algorithm is proposed to further consider the structure of DAGs and communications, information that can be used to get a good schedule length result. Simulation results show that a significant improvement in scheduling length can be achieved by the look-forward algorithm described in this chapter with an appropriate execution time.

5.1 Introduction

The key problem of task scheduling is mapping the tasks onto different resources, which means the algorithm should decide what tasks should be executed using which resources, with a start time and a finish time for each task. These resources include multi-processors, multi-cores, different servers and distributed platforms. In order to achieve a correct and efficient schedule, the decision-making strategy must follow the task precedence constraint [BHR09] (such as task i must be executed and completed

before task j can start if there is a communication $c(e_{ij})$ from task i to j) and the processor constraint [SSS06] (e.g. if task i and j are executed by the same resource r, their execution times cannot overlap). At the same time, the competition for resources, overall scheduling length, use of memory, communication intensity or algorithm execution time should also be considered for different purposes of scheduling. In this thesis, the core concern is to achieve the minimal scheduling length with a reasonable algorithm execution time.

No matter what methodology the algorithm uses to do the scheduling, when it comes to mapping a specific task i onto a processor p (or resource r), how to choose the processor p becomes particularly important. Different solutions have been proposed during the development of task scheduling. At the beginning, the algorithm chooses p by the priorities of the processors, which means the highest priority processor p will be chosen for task i. A simple way is to choose the processors with a round-robin sequence, each processor is given a sequence number and chosen by the circular order of the numbers [CB76]. This simple way can give every processor an equal chance to execute tasks, but shortages are also common, as the processor may be too busy to execute the previous tasks and the new task has to wait, even though other processors are free to execute it at the same time, or the chosen processor may spend time being idle when the incoming task is not ready, because the precedence constraint conditions are not satisfied.

Another popular way to prioritise the processor is to use the first free processor [ERLA94]. This is understandable; the first available processor is chosen to execute the next ready task. As the algorithms have developed, some other ways [BSB+01] of handing the processor priority have been devised, such as: opportunistic load balancing [FGA+98, AHK98], minimum execution time [FGA+98, AHK98], minimum completion time [AHK98], min-min [FGA+98, AHK98, IK77], max-min [LMXZ14, MXL14, FGA+98, AHK98]. Generally speaking, relying on processor priority is a way of balancing workload and is often used for independent tasks. The most common way to consider about dependent tasks is to choose the processor with the earliest start/finish time [CB76, Ull75]. When combined with a good task priority algorithm (for example, Max-min [MXL14, LMXZ14]), choosing the earliest start/finish processor will create an algorithm with a good workload for dependency tasks without communication delays.

When communication delays are integrated into the scheduling problem [YG93,

CC91], things become much more difficult. Communication delays start to play an important role in affecting the scheduling. Not only the workload balance of the processors, but also the overall schedule length must be considered. However, a good workload balance algorithm does not necessarily imply a minimal schedule length solution. Compared to task scheduling without communication delays, some additional issues, such as communication or the structure of tasks, need to be considered. Focusing has moved from the priority decision of processors to the priority of tasks when mapping a specific task i onto a processor p (or resource r). In this thesis, the words *'processor'* and *'resource'* are used interchangeably, so are the words *'task'* and *'node'*.

Even though the task scheduling with communication delays problem has been proved to be NP-hard [SS01, STK09], many heuristic-based algorithms, such as: heterogeneous earliest finish time (HEFT) [THW02], dynamic level scheduling (DLS) [SL93], critical path on a processor (CPOP) [SS10], fastest critical path (FCP) [RvG99], levelized-min time (LMT) [IÖF95] have been proposed to give some good performance solutions. These algorithms use different ways to improve the performance, and all of them accept one idea of mapping one task i by finding a resource r in which the task has the earliest finish time (EFT), but not by the priority of the processors. EFT causes every task to be finished at the earliest finish time, so that the whole application can be finished in a minimal time. Generally speaking, it has good performance and low complexity, so it has been used as a mainstream task scheduling algorithm since it was proposed.

However, the EFT idea has also some shortcomings: first, it only considers the finish time of the current scheduling task, but not the whole structure of the task graph. Since the whole structure of the task graph is already known previously, algorithms should make full use of this information and not only develop a scheduling idea as if the structure is unknown. Some algorithms, like genetic algorithms [Gra99], evolutionary algorithms [ASA16], electromagnetism-like algorithms [ASK$^+$13], are good, but are often used when the structure and information are unknown. In this thesis, the structure of a task graph is supposed to be known; some structures with join nodes (the nodes which have more than one incoming parent nodes) cannot fully be considered by the EFT time based algorithm.

Second, the basic EFT idea cannot consider resource competition between one task

and the lower priority nodes (not only the children tasks). Because of resource competition between the tasks, a task needs to compete for a resource (local resource, fastest resource or earliest finished resource) with other tasks. One task to be finished in the earliest finished resource may force the lower priority task to choose a later resource. Current studies use a priority queue to consider the resource competition problem. Each task is given a priority by the importance and precedence constraint but none of these priority-based algorithms can determine an optimal priority order due to the NP-hard property of task scheduling. Using only a priority queue is far from enough to solve the resource competition problem.

The shortcomings of basic EFT have already been realized by many researchers; lots of new efforts have been made to overcome these problems, such as clustering heuristic [GY92b, PLW96], insertion technique [THW02], task duplication technique [RA00, STK11], task swapping [SZ04a]. The clustering heuristic has evolved from the idea of using basic EFT to map a task onto a processor. It tries to put the heavy communication tasks in one cluster and put them on the same processor. It is a successful idea which makes it essentially the second mainstream task scheduling heuristic besides the list scheduling heuristic. Clustering heuristics started to use the whole structure of DAG in a general but not in a detail-aware way. They are also developed under unbound processors; if the actual processors are less than what is needed to process the clusters, problems may arise [GY92a, GY93]. Recent studies also show that clustering scheduling is an adapted form of list scheduling if certain conditions are met [Sin07]. Insertion, duplication and swapping are some techniques used to remedy the shortcomings of basic EFT. Another solution that needs to be mentioned is the look-ahead algorithm [BSM10], which uses a look-ahead variant combined with the HEFT algorithm to solve the problem (see the detailed discussion in Section 5.3).

5.2 Task scheduling (DAG, model, basic list scheduling)

An application can be decomposed into many dependent tasks before it can be executed on the distributed machines. These dependent tasks and the dependencies among them are represented by a directed acyclic graph (DAG) [SZ04a], where a vertex represents a task and an edge between two vertices represents the communication between these two tasks. A directed acyclic graph is a graph whose vertices are connected by directed

edges and which has no directed cycles. That is, it contains a set of vertices and directed edges; two vertices are connected by only one edge and there should be no path that starts from one vertex v and follows a sequence of vertices and edges and then goes back to v. In a DAG, the vertices are named nodes or tasks; the weight of the nodes represents the computing cost and the weight of the edge corresponds to the communication cost between tasks. Each task has a computation time and the computation time is different for different resources, depending on the speed of the resources, and each edge has a communication time. Task scheduling is also called DAG scheduling when the application is represented by a DAG. A definition of a DAG is given below.

For task scheduling, the application to be scheduled is represented by a directed acyclic graph (DAG) $G = (V;E;w;c)$, also called task graph [GY93, SKS07], where V denotes a finite set of tasks and E represents a finite set of edges. The non-negative weight $w(n)$ associated with node $n \in V$ gives the computing cost of a task; the non-negative weight $c(e_{ij})$ associated with edge $e_{ij} \in E$ is the communication cost between two tasks.

A parallel/distributed system is the target computing environment in which the tasks will be executed. To discuss the target parallel/distributed system, a model which can represent the processors and the connection between these processors must be defined. Most scheduling algorithms employ a strongly idealized model of the target parallel/distributed system. This model, which shall be referred to as the classic model, has been defined in Section 3.1. The classic model represents a parallel/distributed system P which consists of a finite number of processors and a fully connected communication network. The target system can only execute one program or one task graph at a time. If two tasks are allocated to the same processor, the communication between them is called local communication. Local communication is very small and can be ignored by the scheduling algorithm. This property has an important effect on task scheduling for the reason that many tasks may be scheduled on the same processor in order to zero the communication between them.

The scheduling algorithm is the algorithm approach that is used to allocate and schedule tasks into the target parallel/distributed system. If these tasks are executed in a random or disorganized order, the target system may need a long time to finish the application and a lot of computing resources will be wasted. A good task scheduling algorithm can help the system to finish its tasks as quickly as possible, reduce the

time that resources are idle and finally improve the whole performance of the system. There are two mainstream scheduling heuristics for task scheduling, list scheduling and clustering scheduling heuristics. The algorithms included in this thesis all fall under the category of list scheduling heuristics. List scheduling is one of the most popular scheduling heuristics for task scheduling. A good list scheduling algorithm can have low complexity and a short schedule length. The basic steps of static list scheduling are given below [SS04]:

1. Sort all nodes $n_i \in V$ into list L, according to a priority scheme and precedence constraints.
2. For each $n \in L$ do:
   (a) Choose a processor $p \in P$ for n.
   (b) Schedule n on P.

5.3 HEFT, look-ahead and look-forward algorithm

This thesis presents a look-forward algorithm. This algorithm will be compared to a benchmark algorithm called HEFT [THW02] and another algorithm called look-ahead [BSM10]. In these three algorithms, the priority of each node $n_i \in V$ is calculated by the upward rank [THW02]. The upward rank of a task n_i is recursively defined by:

$$rank_u(n_i) = w_i + max\{n_j \in succ(n_i)(c_{i,j} + rank_u(n_j))\},$$

where w_i is the weight of task n_i, $c_{i,j}$ is the communication cost of edge e_{ij}, $succ(n_i)$ is the set of all the immediate successors of task n_i.

5.3.1 HEFT

HEFT is a typical list scheduling algorithm. There are two phases in HEFT, which correspond to the two key phases of list scheduling heuristics. The first is the task prioritizing phase. Every task n_i is given a priority by the upward rank value, $rank_u(n_i)$. Tasks are sorted in a task list in decreasing order of priority. When two tasks have the

same upward value, tie-breaking is done randomly. The second phase is the processor selection phase. Processor selection is the most important part of HEFT. The algorithm will choose a processor for task n_i. Different from some other list scheduling algorithms which select the processor by the earliest start time, HEFT chooses the processor by the earliest finish time of n_i. This makes HEFT particularly effective for the heterogeneous environments. There is also an insertion-based policy to help HEFT select processors. The algorithm will choose an earlier slot for a task and insert it into the slot if it is available. As HEFT is based on list scheduling, it has as low complexity as list scheduling heuristics. The HEFT algorithm can be seen in Algorithm 5.1 where $rank_u$ means the upward rank of the tasks, EFT means the earliest finish time of the task t and $FT(t, r_i)$ means the finish time of task t on resource r_i.

Algorithm 5.1 HEFT planning algorithm

1: Input: a task graph $G = (V, E, w, c)$
2: Output: a schedule for all tasks $t_i \in V$
3: rank tasks $t_i \in V$ using the $rank_u$
4: **while** there are unscheduled tasks **do**
5: $t \leftarrow$ unscheduled task with highest $rank_u$
6: $EFT_t \leftarrow MAXVALUE$
7: $MAXVALUE$ is a maximum number
8: $r \leftarrow NULL$
9: **for all** resource $r_i \in$ the resources set R **do**
10: schedule t on r_i
11: get $FT(t, r_i)$ finish time of t on resource r_i using insertion-based scheduling policy
12: **if** $EFT_t > FT(t, r_i)$ **then**
13: $EFT_t \leftarrow FT(t, r_i)$
14: $r \leftarrow r_i$
15: **end if**
16: **end for**
17: schedule t on r
18: **end while**

5.3.2 Look-ahead

In order to address the shortcomings of the basic EFT idea, previous research has proposed a look-ahead idea [BSM10]. It proposes an improvement of HEFT, where the locally optimal decisions made by the heuristic do not rely on the earliest finish time of one single task, but instead look ahead in the schedule and take into account the children tasks. Different from other algorithms, the look-ahead algorithm will schedule a task by the earliest finish time of this task's children tasks. The heuristic makes the decision by choosing the resource that makes all the children of the task have the earliest finish time. Look-ahead checks the finish time of the children nodes; if one resource can make the children nodes have an earlier finish time, this resource will be chosen. In this way, the look-ahead algorithm can consider the join node structures of a DAG and because the parent node will consider the children nodes, it can also consider the resource competition between parent nodes and children nodes. The experiments show that in comparison to HEFT, look-ahead can improve the schedule in most of the cases, especially in cases where the communication cost is higher with respect to computation. The details of the look-ahead algorithm can be seen in Algorithm 5.2.

Algorithm 5.2 Look-ahead planning algorithm

1: Input: a task graph $G = (V, E, w, c)$
2: Output: a schedule for all tasks $t_i \in V$
3: rank tasks $t_i \in V$ using the $rank_u$
4: **while** there are unscheduled tasks **do**
5: $t \leftarrow$ unscheduled task with highest $rank_u$
6: $L \leftarrow$ children of t
7: **for all** resource $r_i \in$ the resources set R **do**
8: schedule t on r_i
9: schedule all tasks $\in L$ on using $HEFT$
10: $EFT_{r_i} \leftarrow$ maximum EFT for tasks $\in L$
11: return to the schedule state at the beginning of this loop
12: **end for**
13: schedule t on r_i such that $EFT_{r_i} \leq EFT_{r_k}$
14: **end while**

5.3.3 Look-forward

Most of the existing approaches accept the heuristic of scheduling one task by finding a resource by which the task has the earliest finish time (EFT). EFT causes every task to be completed as soon as possible, so that the whole application can be completed in a short time. Even though look-ahead provides a step in the right direction to cope with the shortcomings of basic EFT idea, its decision making method, the earliest finish time of all the children nodes, may still make some poor decisions. That is because it cannot consider the bad effects from some unscheduled but higher priority nodes (higher priority than children nodes), which means the benefit from look-ahead will be reduced or eliminated by these unscheduled but higher priority nodes. Meanwhile, it does not consider the resource competition between higher priority tasks and the children nodes. Based on these observations, a look-forward idea is proposed to solve this problem.

This look-forward algorithm uses a new heuristic; the higher priority node should also take into account lower priority nodes. Look-forward is a further improvement of the look-ahead algorithm. It inherits the heuristic of look-ahead where parent nodes should look ahead to the earliest finish time of all the children nodes. But it also considers the bad effects from the unscheduled but higher priority nodes (higher than the children nodes) which are being looked forward. As to the question of what higher priority tasks the look-forward algorithm should consider: the look-forward algorithm keeps a node set L; every time when it schedules a node t_i, it puts t_i's children nodes into L and then chooses the resource that makes all the nodes in L have the earliest finish time. If t_i is inside L, then L is emptied first and the loop for t_i is repeated. The look-forward algorithm can be seen in Algorithm 5.3.

5.4 Experimental evaluation

5.4.1 DAX and generator

A workflow generator is used to generate the input workflows. This generator uses the information gathered from actual executions of scientific workflows as well as the understanding of the processes behind these workflows to generate realistic, synthetic workflows resembling those used by real-world scientific applications. The format of

Algorithm 5.3 Look-forward planning algorithm

1: Input: a task graph $G = (V, E, w, c)$
2: Output: a schedule for all tasks $t_i \in V$
3: rank tasks $t_i \in V$ using the $rank_u$
4: $L \leftarrow NULL$
5: **while** there are unscheduled tasks **do**
6: $t \leftarrow$ unscheduled task with highest $rank_u$
7: **if** $t \notin L$ and children of $t \notin L$ **then**
8: $L \leftarrow$ children of t
9: **end if**
10: **if** $t \in L$ **then**
11: $L \leftarrow NULL$
12: $L \leftarrow$ children of t
13: **end if**
14: **for all** resource $r_i \in$ the resources set R **do**
15: schedule t on r_i
16: schedule all tasks $\in L$ using $HEFT$
17: $EFT_{r_i} \leftarrow$ maximum EFT for tasks $\in L$
18: return to the schedule state at the beginning of this loop
19: **end for**
20: schedule t on r_i such that $EFT_{r_i} \leq EFT_{r_k}$
21: **end while**

the workflow is called 'DAX'. A DAX consists of information about tasks, input and output files and the relationship between tasks and files. A task has a runtime which can be used to calculate the computation time of this task and a file has a data length which can be used to calculate the communication cost during different target systems. Four kinds of DAX are generated: Montage [BCD+08], LiGo [BCD+08], Epigenomics [BCD+08], CyberShake [BCD+08]. The computation cost and communication cost of each task were generated from the interval (500, 4000). In order to generate different CCR values (CCR is the rate of communication cost divided by computation cost) for each DAX, the communication costs will be changed according to the CCR before scheduling. For each type of DAX, three kinds of CCR (0.5, 1.0, 2.0) are chosen to do

the experiments. All the workflows are run by HEFT, MHEFT, look-ahead and look-forward algorithms. Each algorithm runs 200 randomly generated workflows under 2 resources and 10 resources for each type of workflow. Except for the four kinds of workflow, a test workflow which has the same structure as the one in the HEFT paper [THW02] is also produced to test the accuracy of the algorithms.

5.4.2 The simulator and settings

WorkflowSim [Che13] is used to simulate the running of the workflows. WorkflowSim is an open source workflow simulator that extends CloudSim by providing workflow level support for simulation. It models a workflow with a DAG model and supports an elaborate model of node failures, a model of delays occurring in different layers and components, and includes implementations of several of the most popular dynamic and static workflow scheduling algorithms (e.g., HEFT [THW02], look-forward (see Section 5.3), min-min [KA11]) and task clustering algorithms (e.g., runtime-based algorithms, data-oriented algorithms and fault-tolerant clustering algorithms). There are two layers in WorkflowSim: workflow planning layer (HEFT, DLS [SL93], FCP [RvG99], LMT [IÖF95]) and workflow scheduling layer (static scheduling, FCFS, MAXMIN). The HEFT, look-ahead and look-forward algorithms all work on the workflow planning layer. In order to keep the result of planning layer algorithms, a static scheduling algorithm will be used in the workflow scheduling layer. DAXs are used as the input file of WorkflowSim, the output is a schedule for each DAX. WorkfowSim reads the DAXs files, then runs the planning algorithms (e.g., HEFT, look-ahead, look-forward); after finishing the planning algorithms, the results will be sent to a simulator and the simulator will run the tasks by scheduling them with our algorithms then get the scheduling time of DAX. For each workflow, the simulation runs for different algorithms and under different target machines. The output data for all the workflows will be collected to analyze the results. The average scheduling length for each kind of DAX and for each algorithm will be obtained. The four algorithms are implemented in WorkflowSim and the workflows are also tested under WorkflowSim.

A HEFT planning algorithm based on the paper “Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing” [THW02]) has been implemented and used to test the workflows. A HEFT planning algorithm called MHEFT

is implemented and compared to a standard HEFT with different workflows. The result shows that MHEFT and HEFT have the same performance.

5.4.3 Results

The experiments show the results of the HEFT, MHEFT, look-ahead and look-forward algorithms on 2 or 10 resources and using CCR 0.5, 1.0 and 2.0. The main purpose is to compare the look-forward with the HEFT and look-ahead algorithms. The horizontal axis shows the number of tasks in DAGs and the vertical axis shows the average makespan.

Figure 5.1 to Figure 5.6: these six figures present the results of Montage workflow with different CCR values (0.5, 1.0 and 2.0). The results allow us to make the following observations: HEFT and MHEFT have the same performance as expected. Both look-ahead and look-forward work better than HEFT in most cases. When the number of tasks is small, the look-ahead and the look-forward algorithms obtain similar results. But, in some cases, look-ahead works slightly better than the look-forward algorithm when the number of tasks is small (for example, when the number of tasks is 24, CCR 2.0 and using 10 resources, look-ahead achieved a 21.8% improvement but look-forward achieved a 19.14% improvement). With an increasing number of tasks, look-forward starts to work better than look-ahead (see Figure 5.1 to Figure 5.6 when the number of tasks is 50, 100, 200, 500, 700, 1000). Look-ahead obtains less and less improvement with an increasing number of tasks: for example look-ahead obtains a benefit from 10.91% to -0.78% when the number of tasks increases from 24 to 700, CCR 1.0 and 10 resources (see Figure 5.4). Look-forward has a trend where the higher the number of tasks the better improvement it can make: for example look-forward obtained an improvement from 5.11% to 48.27% when the number of tasks increases from 24 to 1000, CCR 0.5 and 10 resources (see Figure 5.2). When the number of tasks is more than 500, look-forward obtains more than a 35% improvement in all the cases, more than 40% improvements when the number of tasks is more than 700, while look-ahead obtains little improvement sometimes even negative improvement.

Figure 5.7 to Figure 5.12: these six figures present the results of the LiGo workflow with different CCR. The results of the LiGo workflow are similar to the Montage workflow in Figure 5.1 to Figure 5.6. As the number of tasks increases, look-forward

works better and better than both HEFT and look-ahead; look-ahead obtains less and less improvement compared to HEFT. Some new observations can be seen: look-ahead becomes unstable when the CCR reduces. When the CCR is reduced, look-ahead cannot always obtain a stable benefit and in many cases it even obtains worse results when the CCR becomes smaller (see Figure 5.12, 5.10 and 5.8 look-ahead parts). When CCR reduces from 2.0 to 0.5, look-ahead obtains improvements from 5.46% to -1.83%). By contrast, look-forward kept obtaining a stable benefit when the CCR is changed. The improvements of look-forward are around 2.45% to 9.11%. For all the cases, the results show that look-forward works better than look-ahead.

Figure 5.13 to Figure 5.24: Figures 5.13 to 5.24 present the results of CyberShake and Epigenomics workflow with different CCR. The results of CyberShake and Epigenomics also prove that look-forward works better than look-ahead when the number of tasks increases and works more stably than look-ahead when the CCR is changed. It also shows that look-forward works better than look-ahead and HEFT in all the tested cases. For the CyberShake workflow, look-ahead obtains around 0.33% to 1.85% improvement when the number of resources is 2 and obtains around -4.5% to 3.16% improvement when the number of resources is 10. At the same time, look-ahead will obtain 1.08% to 8.57% improvement when the number of resources is 2 and will obtain 2.15% to 46.6% improvement when the number of resources is 10. For the Epigenomics workflow in general, look-ahead obtains a benefit of -2.5% to 2.13%, on average, it will obtain around 1% improvement. Look-forward obtains a benefit of 1.31% with CCR 0.5 and a number of resources 2 to 23.25% with CCR 2.0 and a number of resources 10; the range is different for different CCR values, number of tasks and resources (see Figures 5.19 to 5.24).

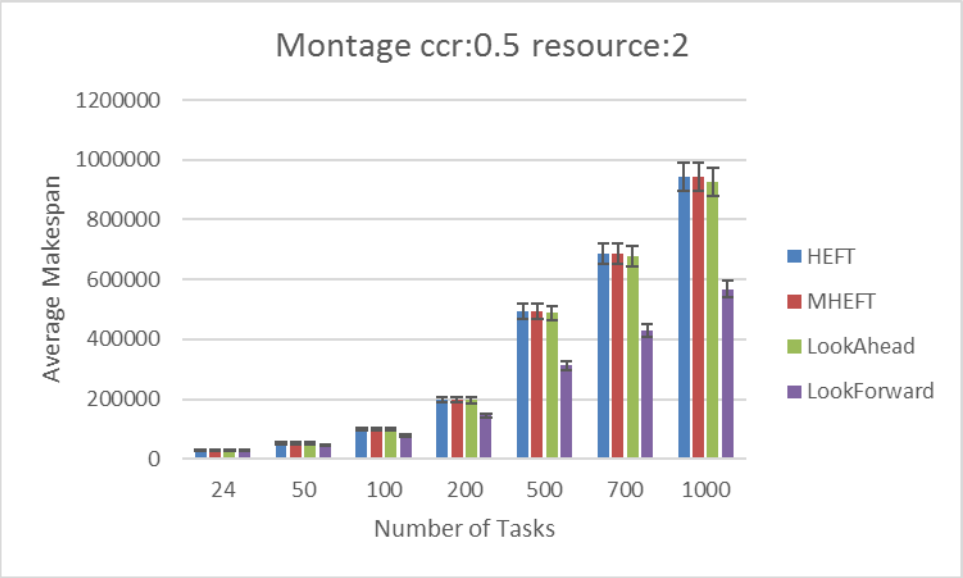

Figure 5.1: Average makespan with standard deviation using Montage workflow, CCR 0.5 and 2 resources

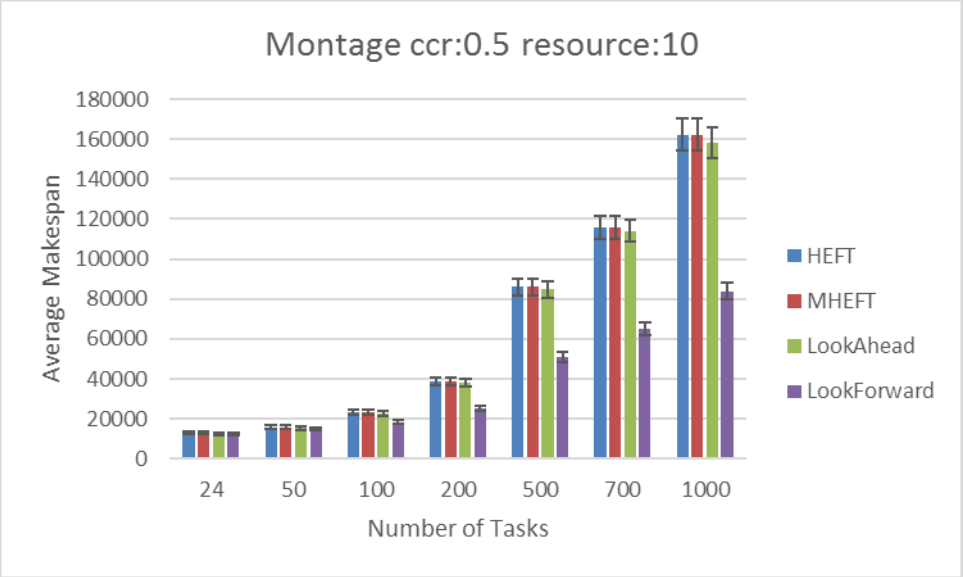

Figure 5.2: Average makespan with standard deviation using Montage workflow, CCR 0.5 and 10 resources

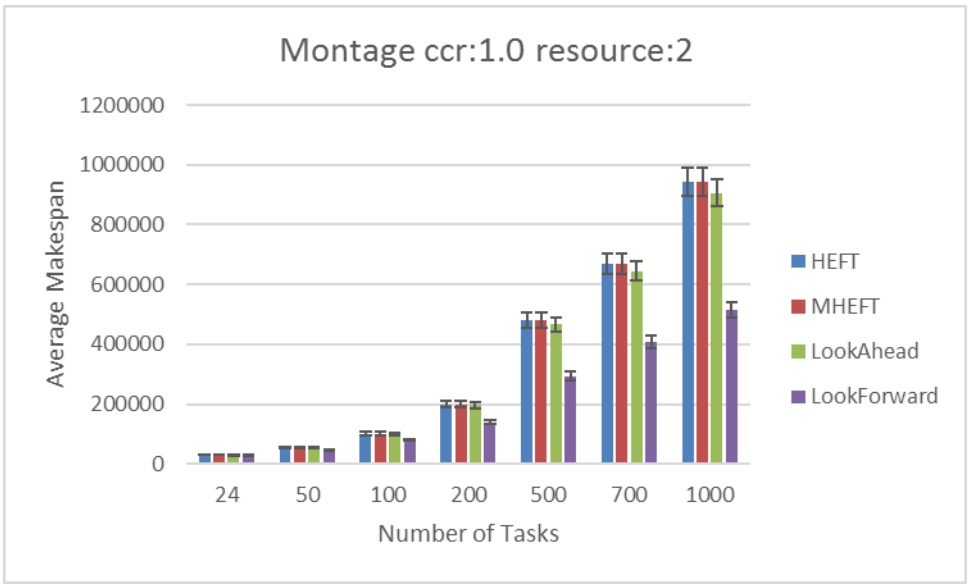

Figure 5.3: Average makespan with standard deviation using Montage workflow, CCR 1.0 and 2 resources

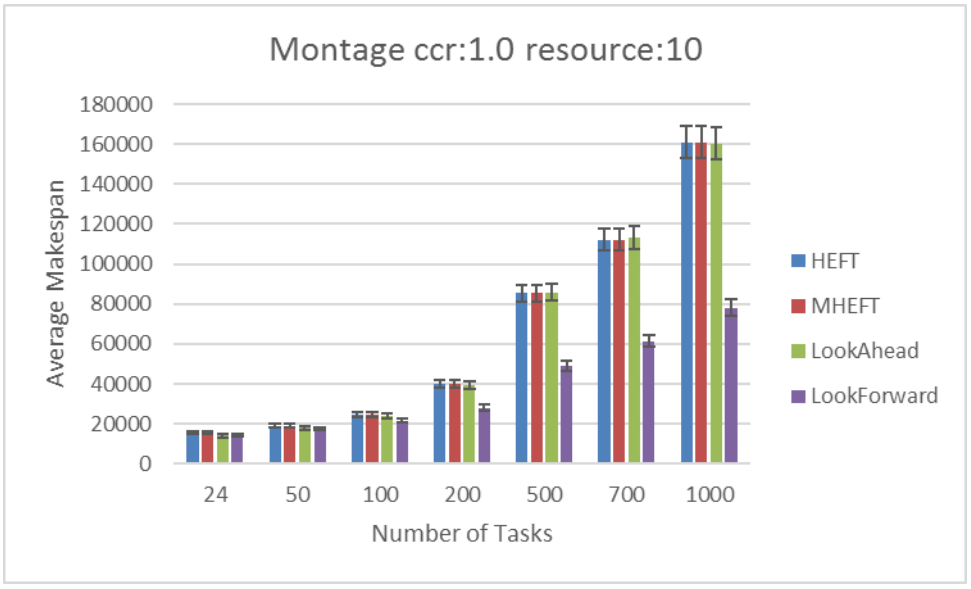

Figure 5.4: Average makespan with standard deviation using Montage workflow, CCR 1.0 and 10 resources

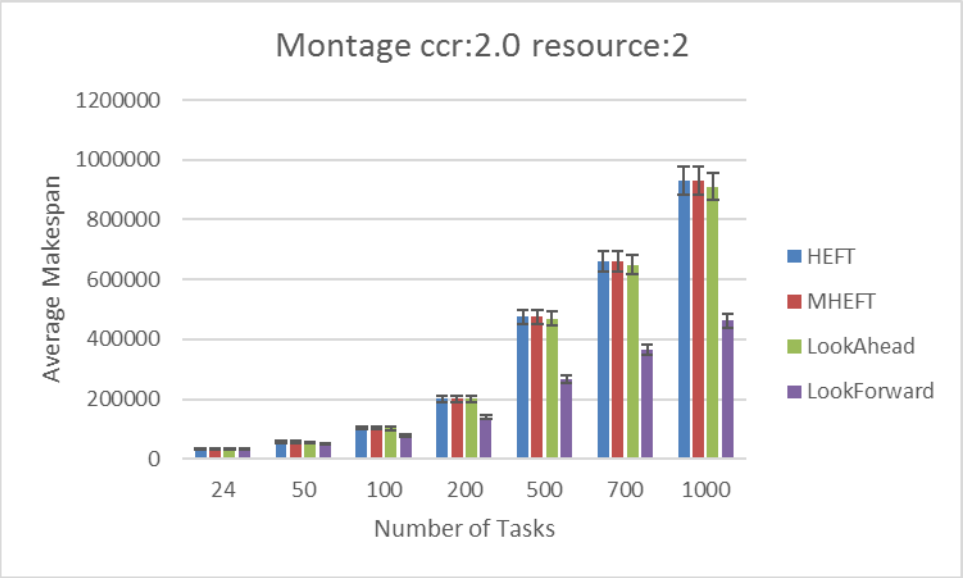

Figure 5.5: Average makespan with standard deviation using Montage workflow, CCR 2.0 and 2 resources

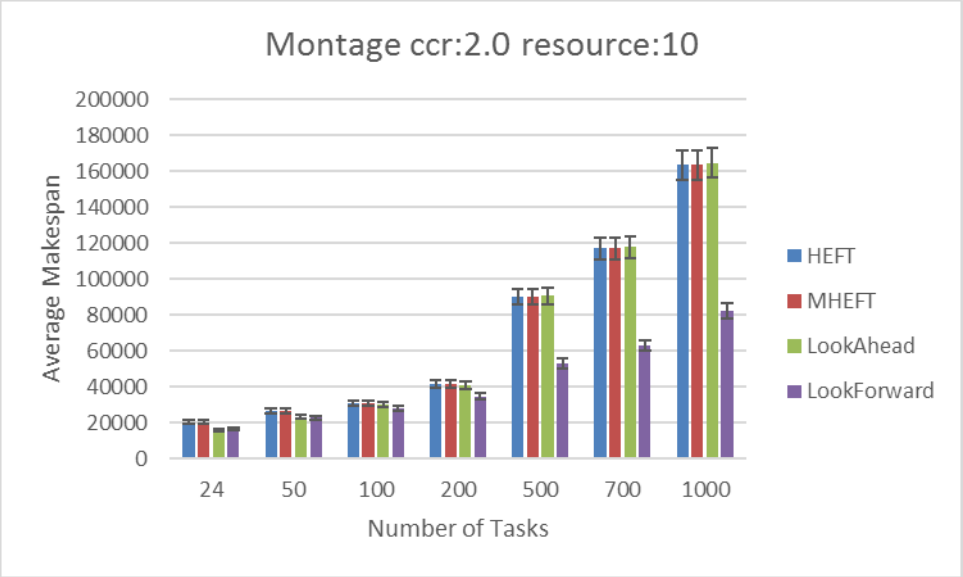

Figure 5.6: Average makespan with standard deviation using Montage workflow, CCR 2.0 and 10 resources

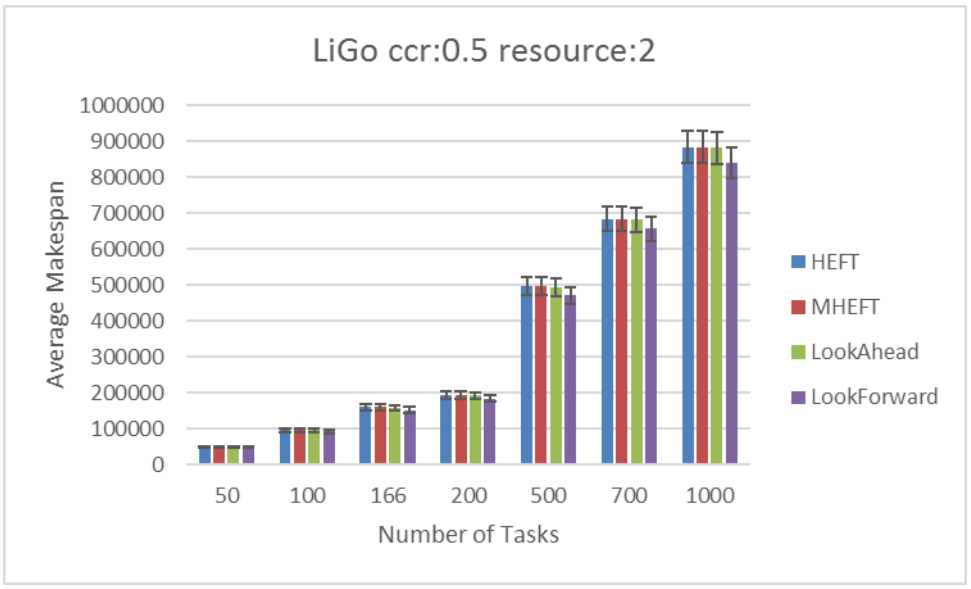

Figure 5.7: Average makespan with standard deviation using ligo workflow, CCR 0.5 and 2 resources

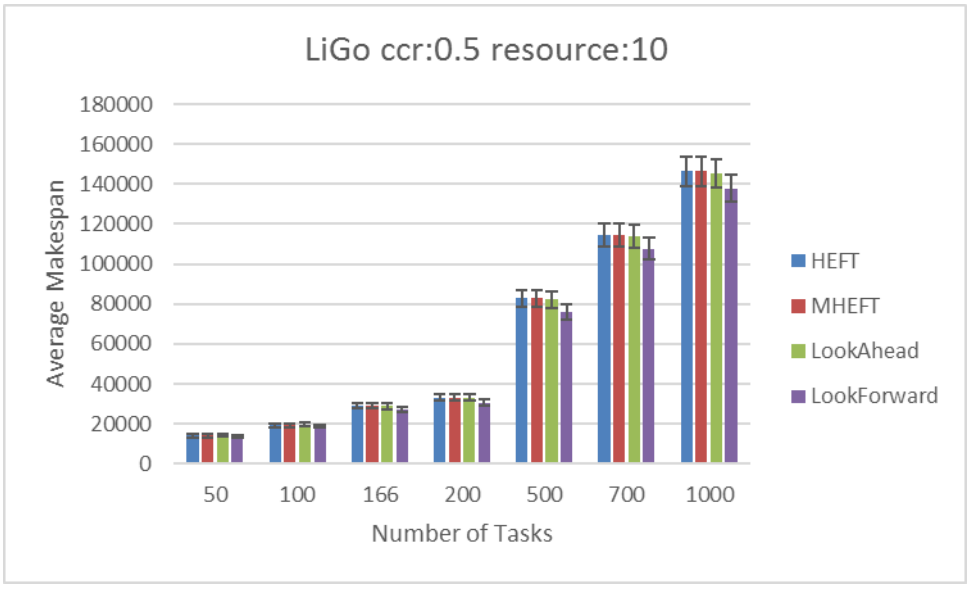

Figure 5.8: Average makespan with standard deviation using ligo workflow, CCR 0.5 and 10 resources

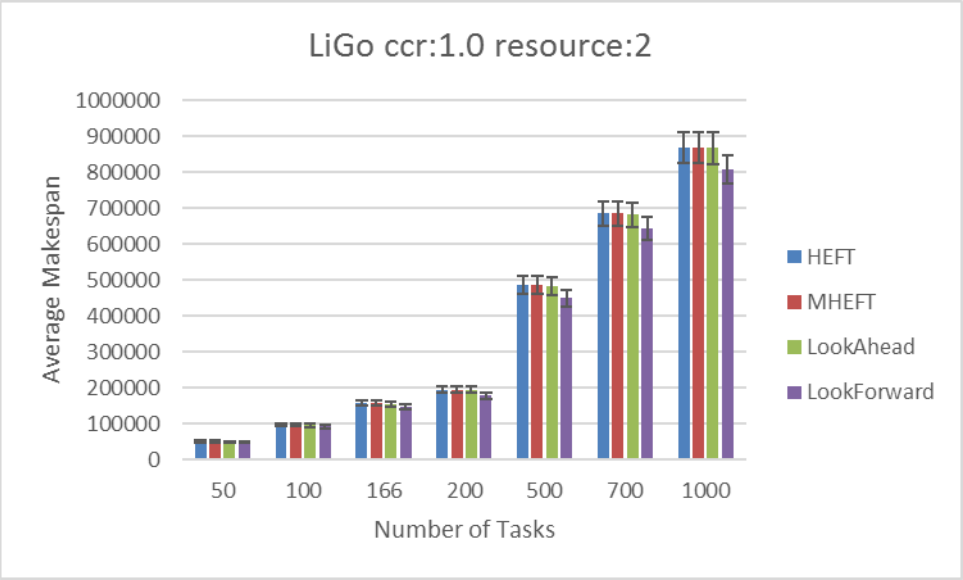

Figure 5.9: Average makespan with standard deviation using ligo workflow, CCR 1.0 and 2 resources

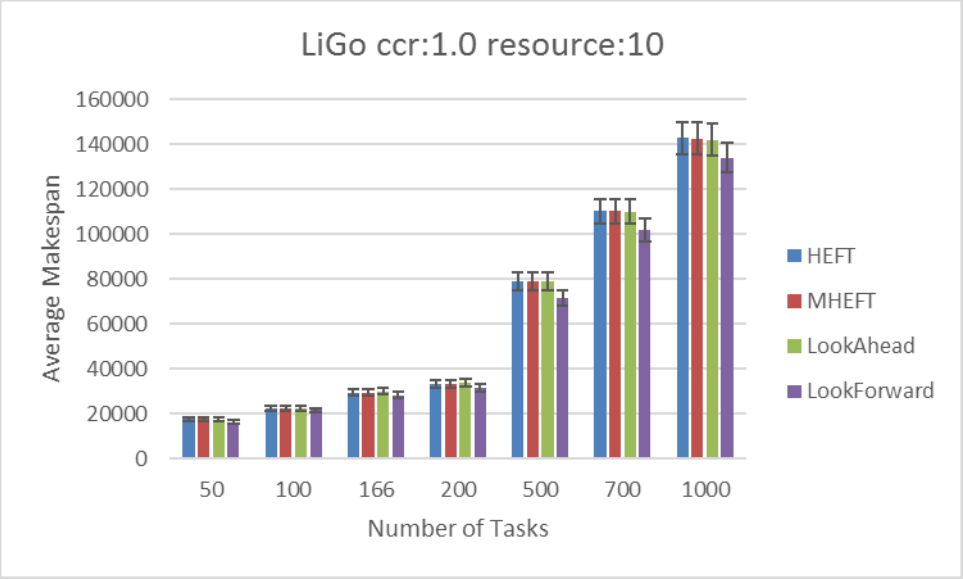

Figure 5.10: Average makespan with standard deviation using ligo workflow, CCR 1.0 and 10 resources

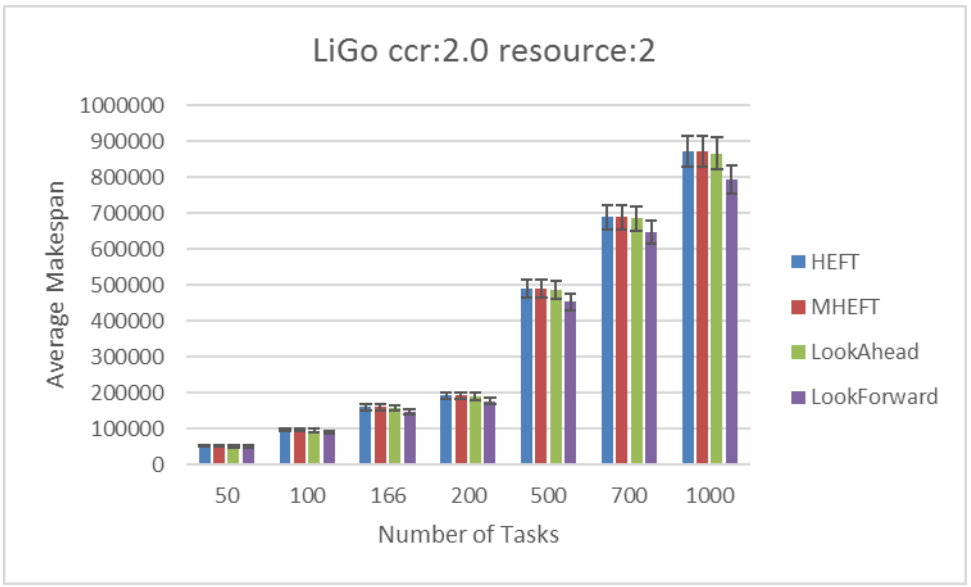

Figure 5.11: Average makespan with standard deviation using ligo workflow, CCR 2.0 and 2 resources

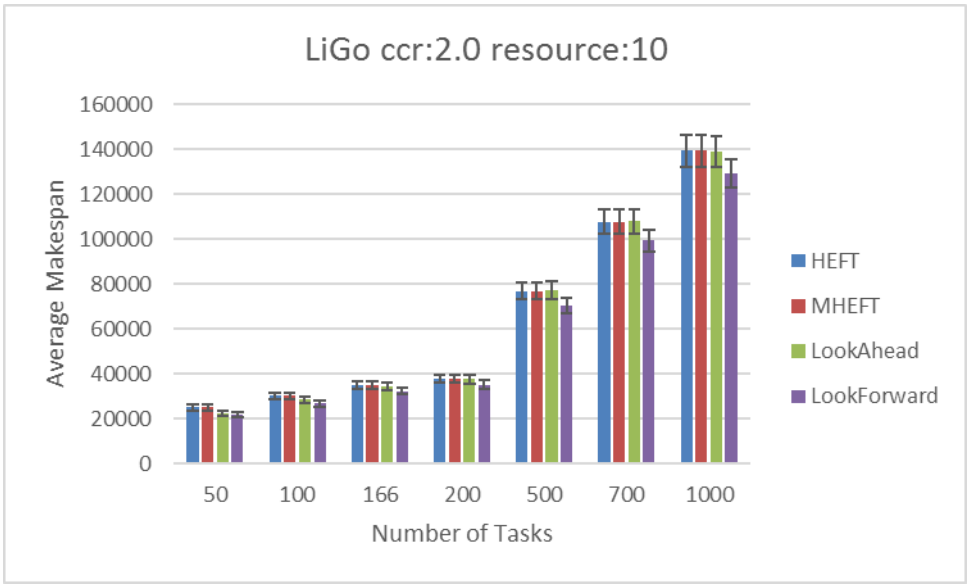

Figure 5.12: Average makespan with standard deviation using ligo workflow, CCR 2.0 and 10 resources

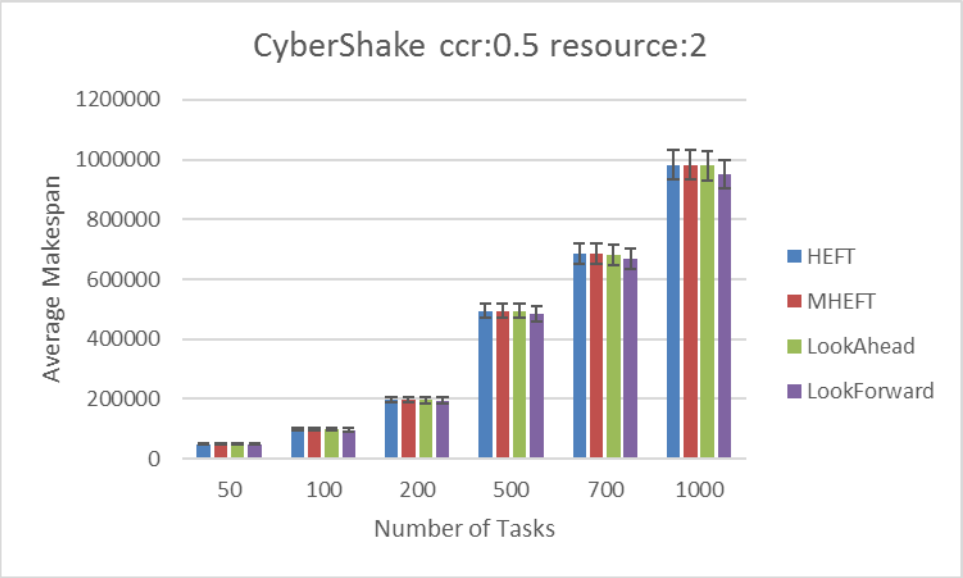

Figure 5.13: Average makespan with standard deviation using cybershake workflow, CCR 0.5 and 2 resources

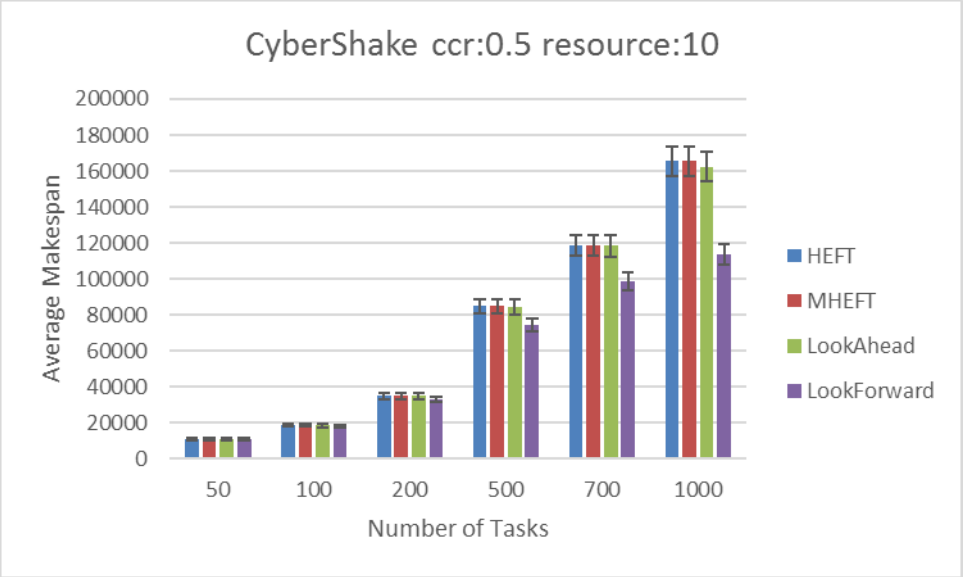

Figure 5.14: Average makespan with standard deviation using cybershake workflow, CCR 0.5 and 10 resources

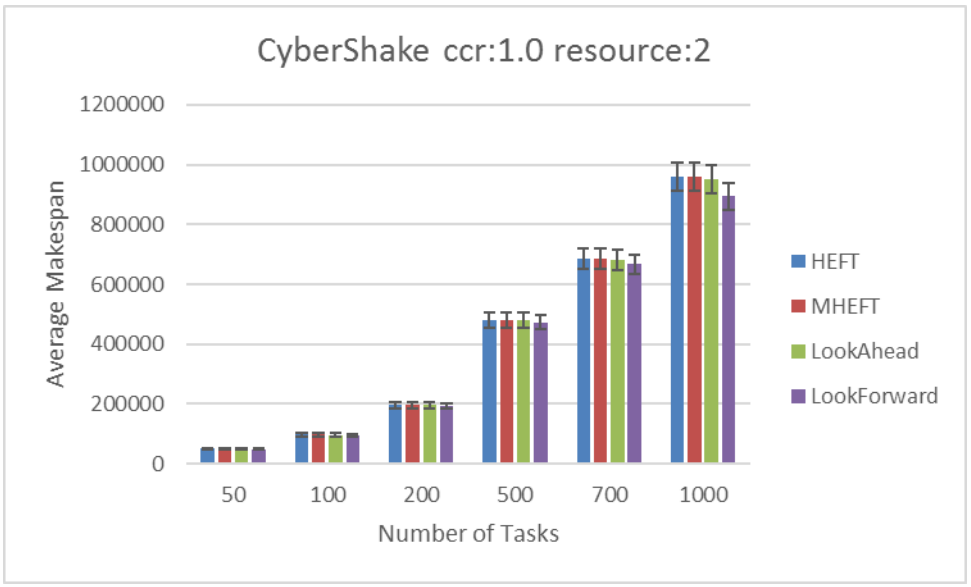

Figure 5.15: Average makespan with standard deviation using cybershake workflow, CCR 1.0 and 2 resources

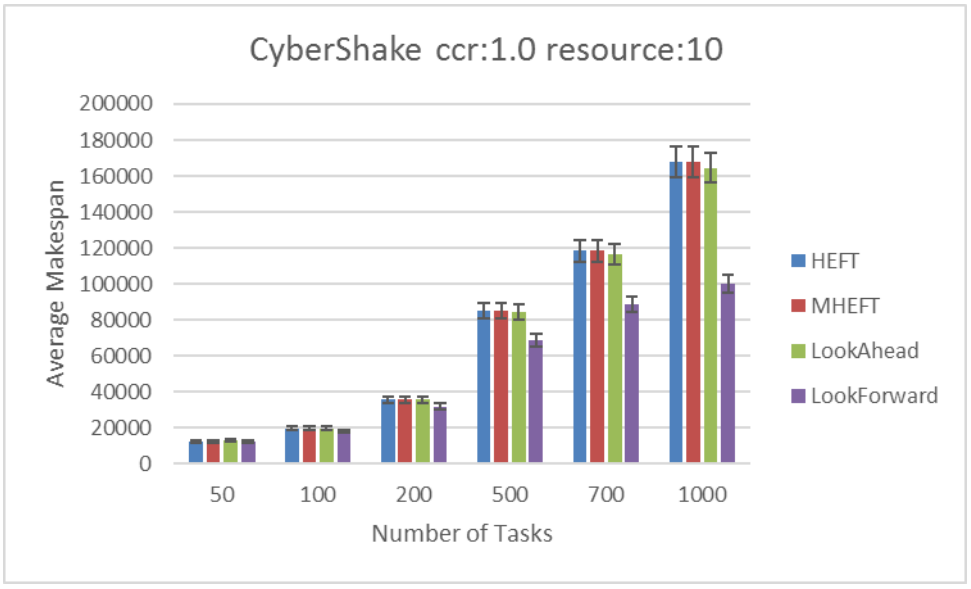

Figure 5.16: Average makespan with standard deviation using cybershake workflow, CCR 1.0 and 10 resources

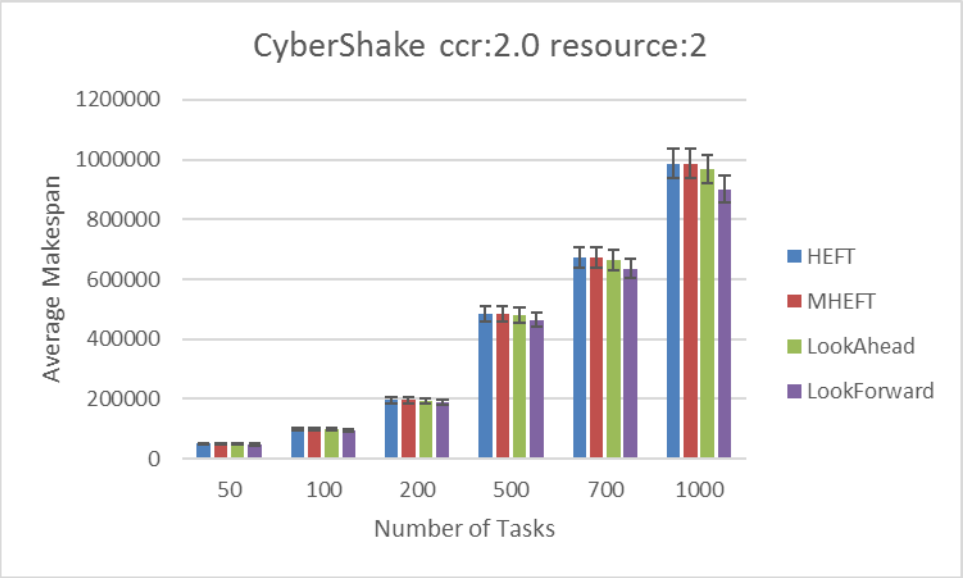

Figure 5.17: Average makespan with standard deviation using cybershake workflow, CCR 2.0 and 2 resources

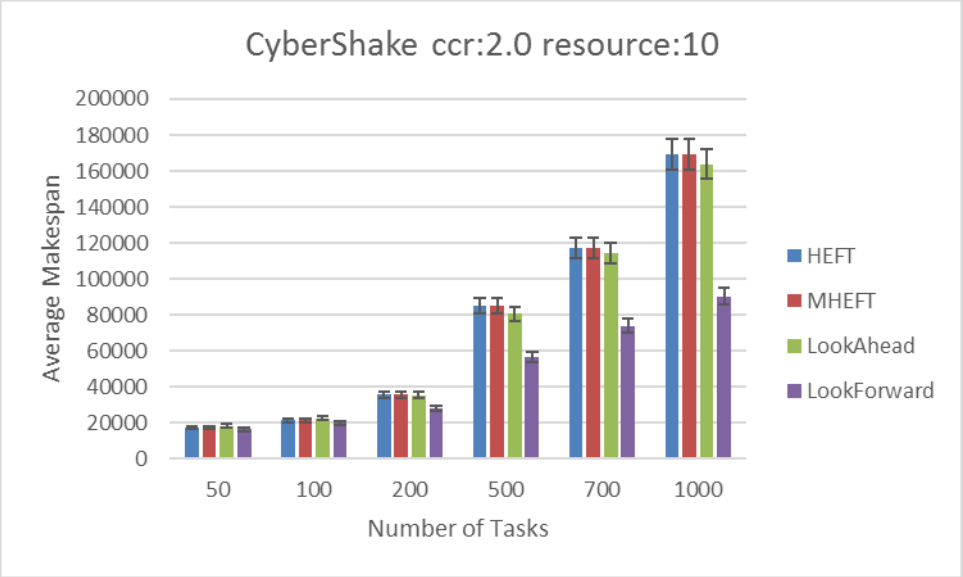

Figure 5.18: Average makespan with standard deviation using cybershake workflow, CCR 2.0 and 10 resources

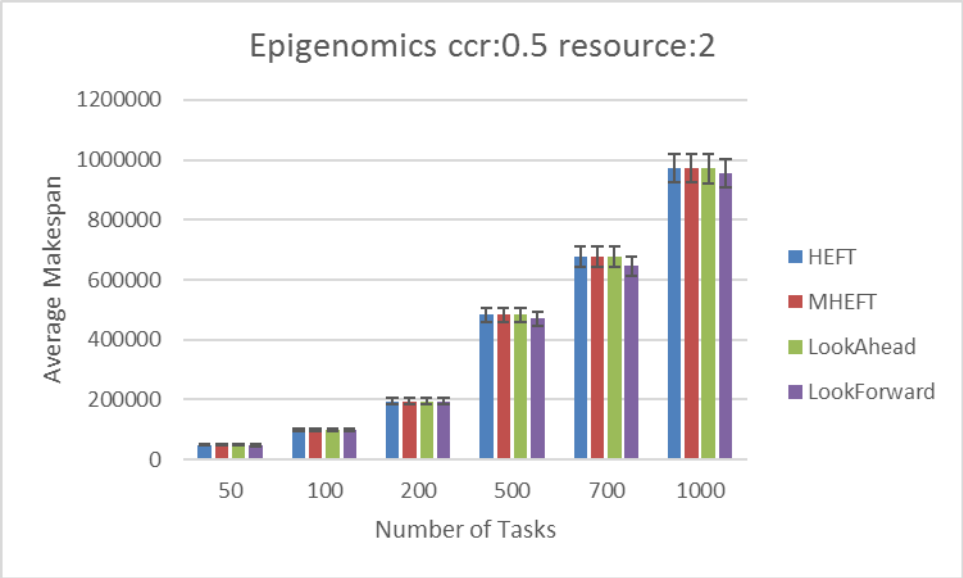

Figure 5.19: Average makespan with standard deviation using epigenomics workflow, CCR 0.5 and 2 resources

Figure 5.25 to Figure 5.32: Figures 5.25 to 5.32 present the amount of communication of the four algorithms. Comparing to HEFT, look-forward reduces the amount of communication in all the cases and look-ahead can reduce the amount of communication in most of the cases. In some cases, look-ahead increases the communication a little (see Figure 5.31 look-ahead parts, number of tasks 500, 700, 1000). Compared to look-ahead, look-forward can reduce communication more especially when the number of tasks increases. When the number of tasks is less than 200, both look-ahead and look-forward can obtain improvements upon HEFT and look-forward works slightly better than look-ahead. When the number of tasks is more than 200, the improvement levels can be seen in Table 5.1 and Table 5.2.

(resource10)	Improvements (%)	Montage	CyberShake	Epigenomics	LiGo
500 tasks	Look-ahead/Look-forward	3.45/22.55	6.03/7.57	8.41/10.2	9.37/11.95
700	Look-ahead/Look-forward	3.14/22.28	5.74/9.11	6.93/9.44	6.59/8.75
1000	Look-ahead/Look-forward	3.11/20.91	5.23/12.56	5.99/10.98	5.54/7.86

Table 5.1: Communication improvement of look-ahead and look-forward when the number of resources is 10

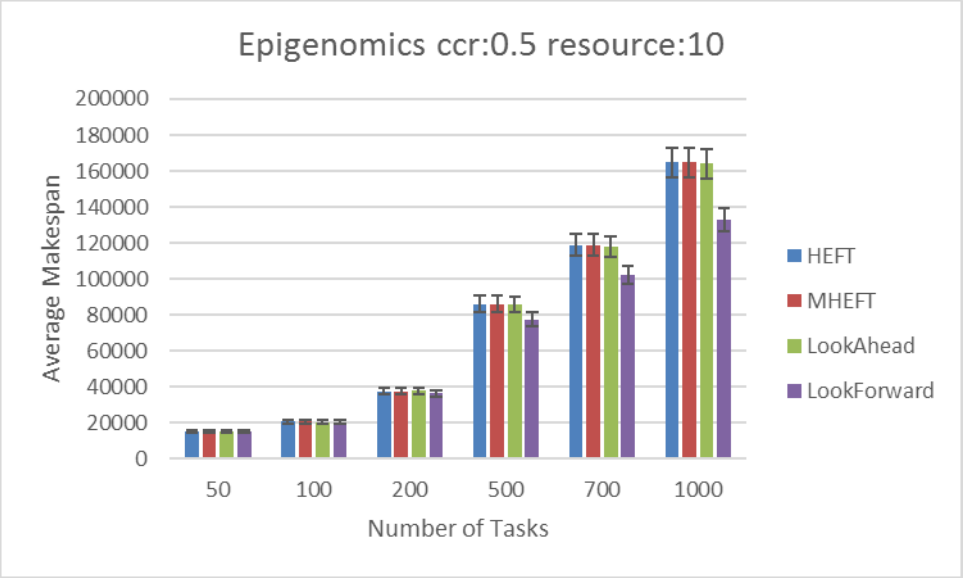

Figure 5.20: Average makespan with standard deviation using epigenomics workflow, CCR 0.5 and 10 resources

(resource 2)	Improvements (%)	Montage	CyberShake	Epigenomics	LiGo
500 tasks	Look-ahead/Look-forward	6.54/8.51	3.99/3.39	6.82/7.68	-1.85/3.05
700	Look-ahead/Look-forward	5.68/10.58	2.92/2.91	5.8/8.31	-2.71/3.06
1000	Look-ahead/Look-forward	6.13/13.12	2.74/2.39	3.91/5.71	-1.56/3.88

Table 5.2: Communication improvement of look-ahead and look-forward when the number of resources is 2

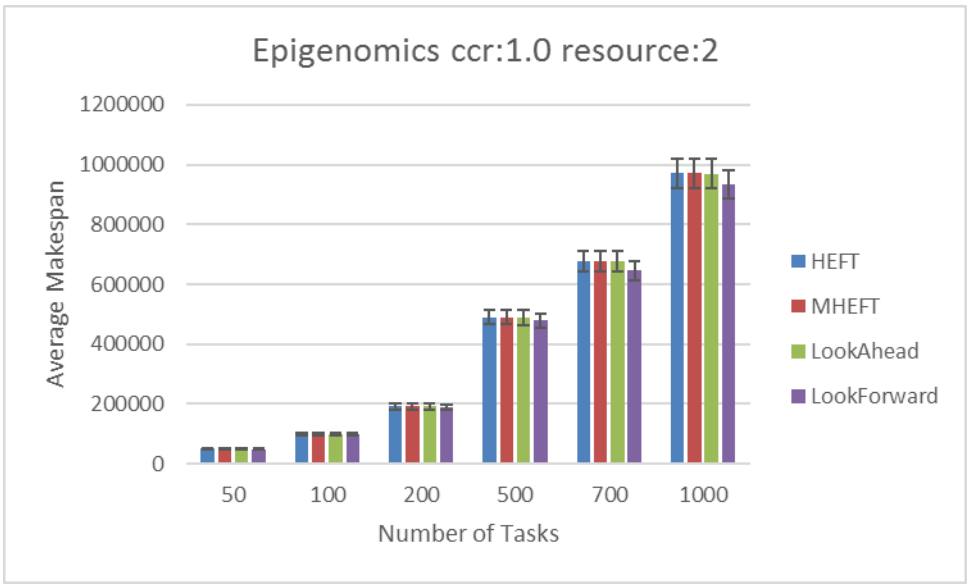

Figure 5.21: Average makespan with standard deviation using epigenomics workflow, CCR 1.0 and 2 resources

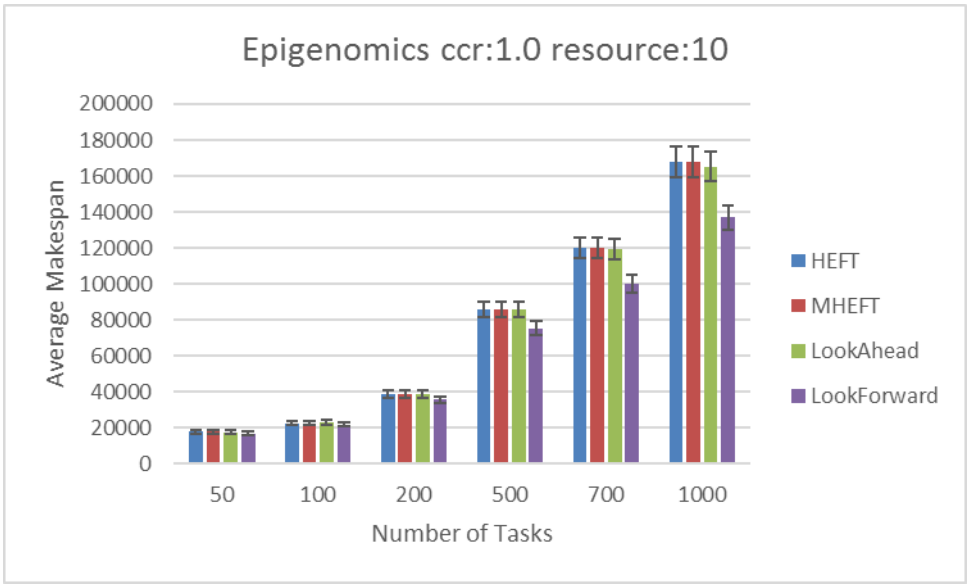

Figure 5.22: Average makespan with standard deviation using epigenomics workflow, CCR 1.0 and 10 resources

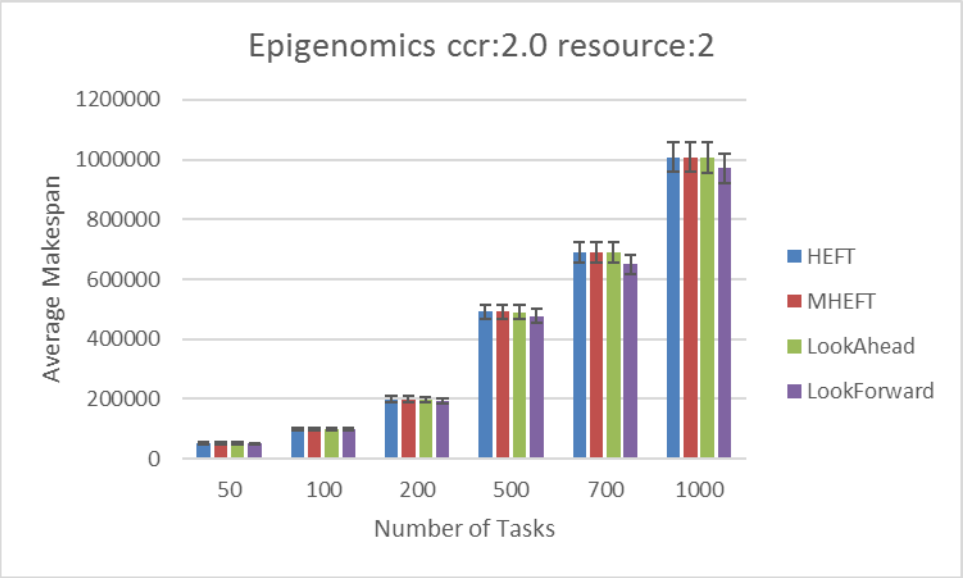

Figure 5.23: Average makespan with standard deviation using epigenomics workflow, CCR 2.0 and 2 resources

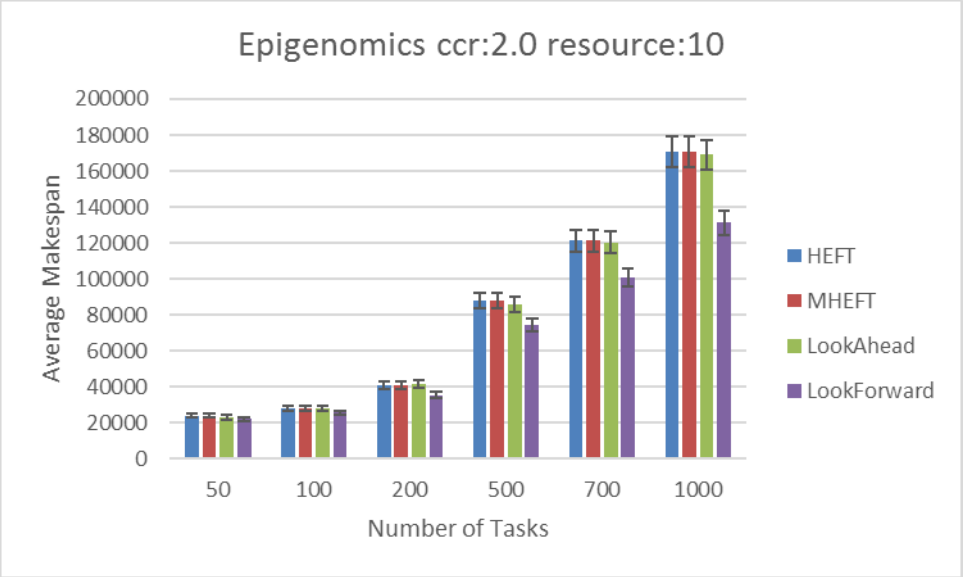

Figure 5.24: Average makespan with standard deviation using epigenomics workflow, CCR 2.0 and 10 resources

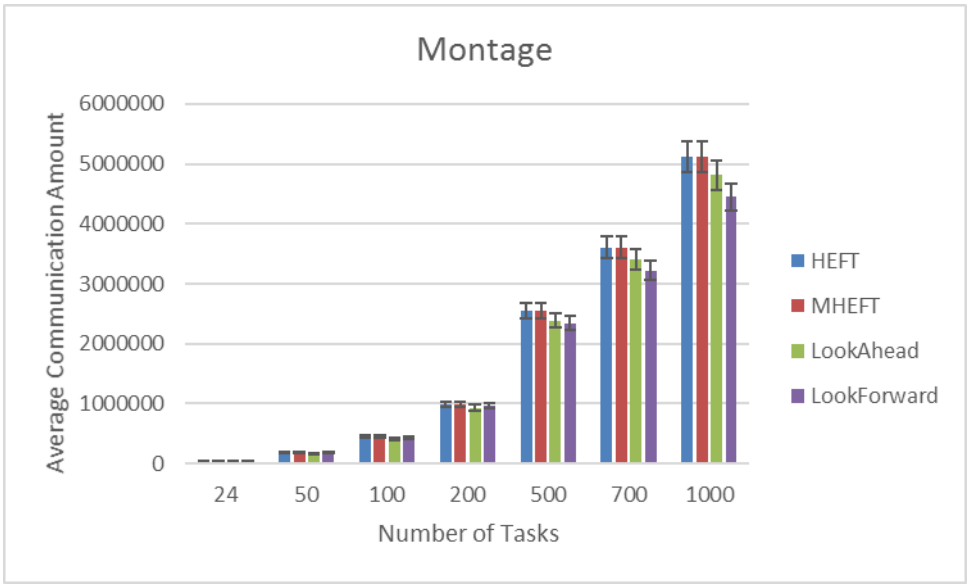

Figure 5.25: Communication amount of montage workflow under 2 resources

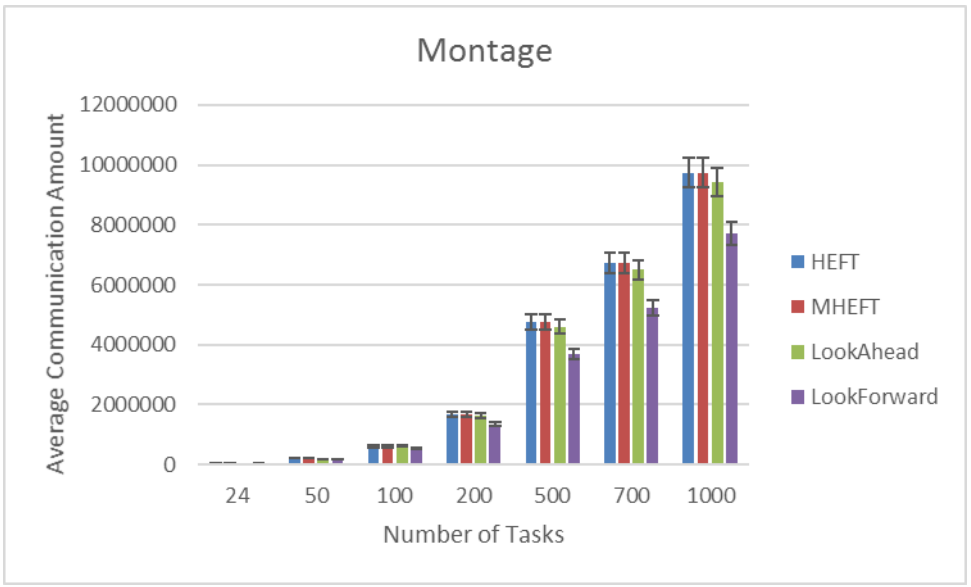

Figure 5.26: Communication amount of montage workflow under 10 resources

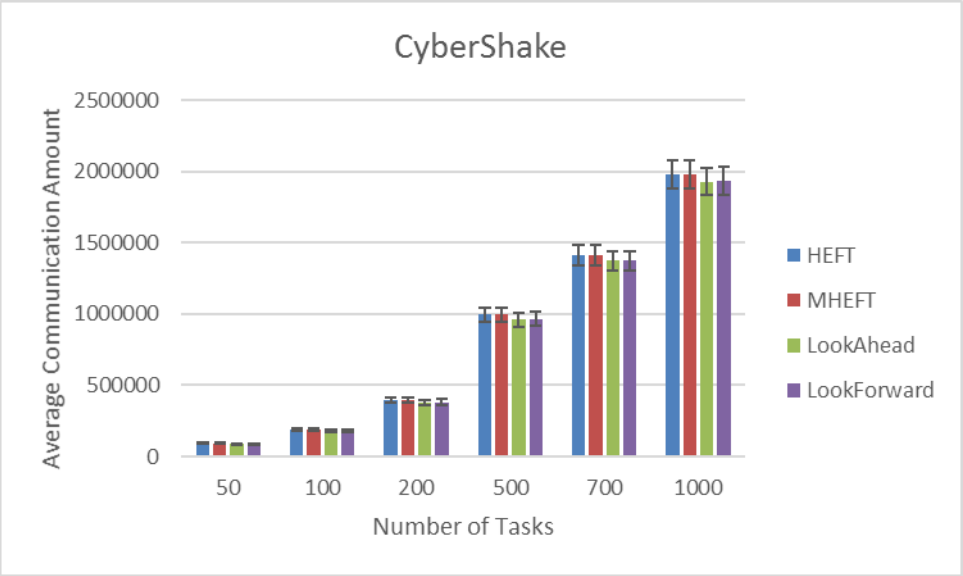

Figure 5.27: Communication amount of cybershake workflow under 2 resources

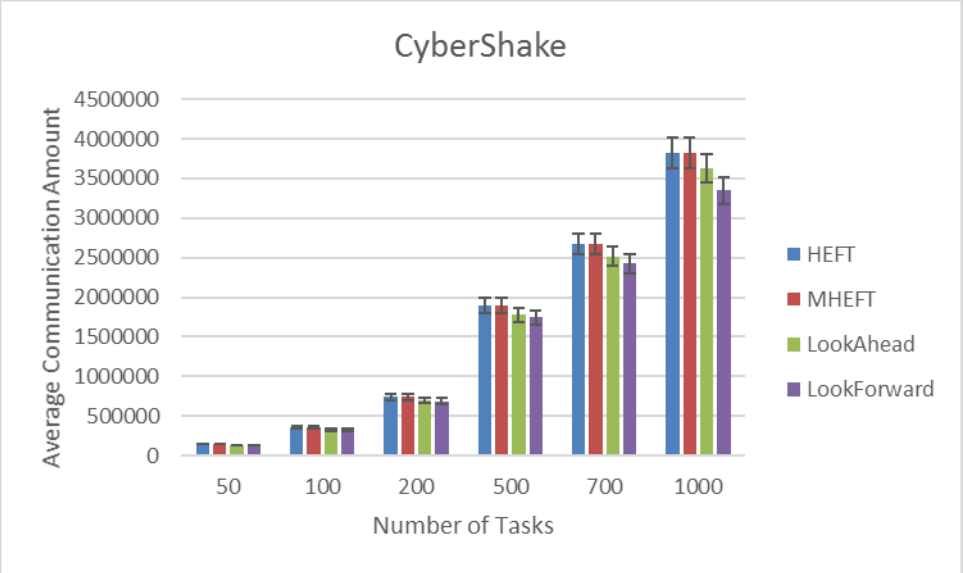

Figure 5.28: Communication amount of cybershake workflow under 10 resources

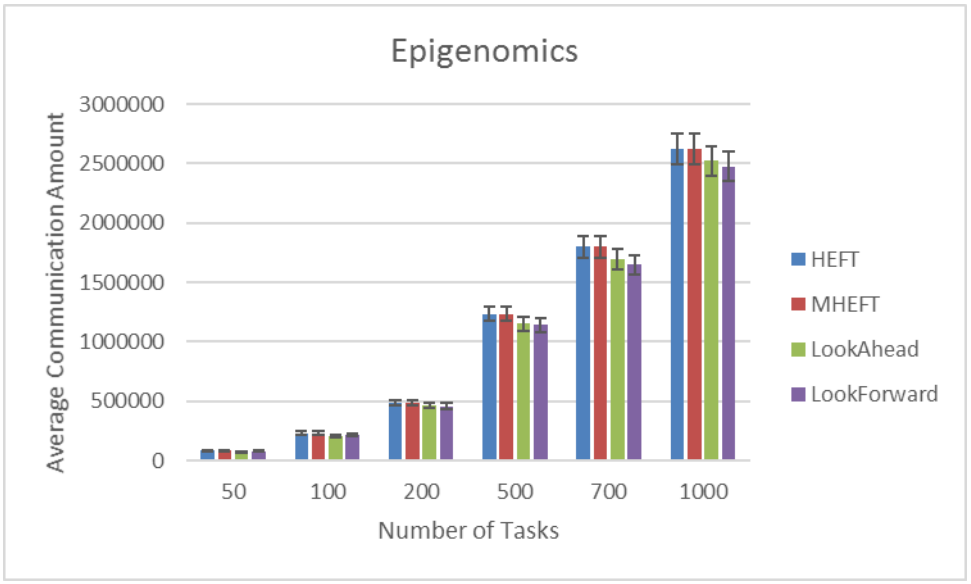

Figure 5.29: Communication amount of epigenomics workflow under 2 resources

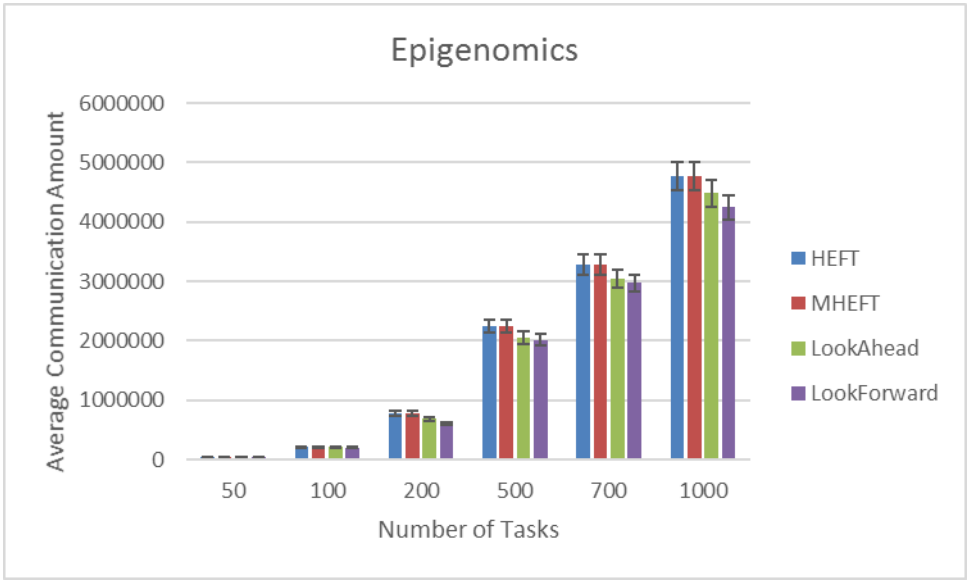

Figure 5.30: Communication amount of epigenomics workflow under 10 resources

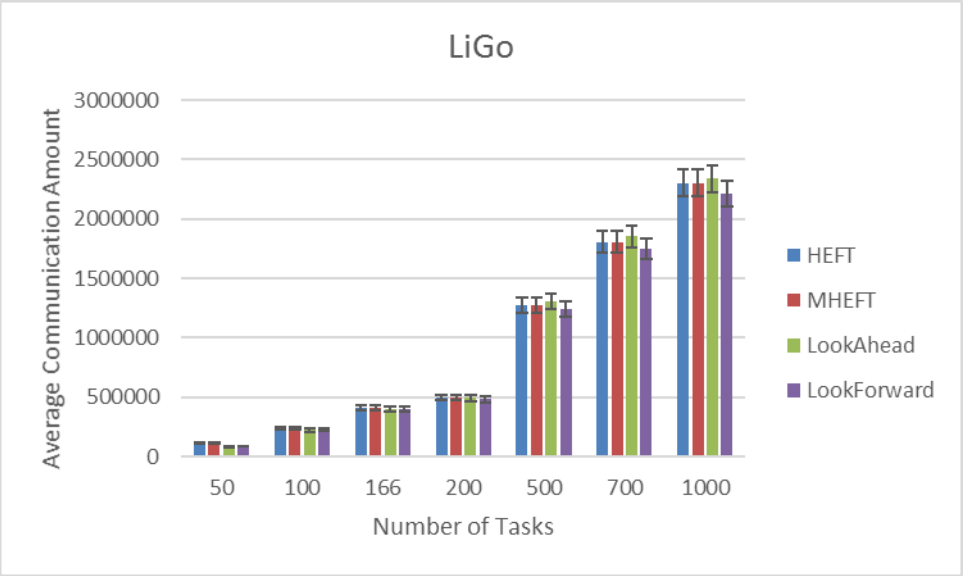

Figure 5.31: Communication amount of ligo workflow under 2 resources

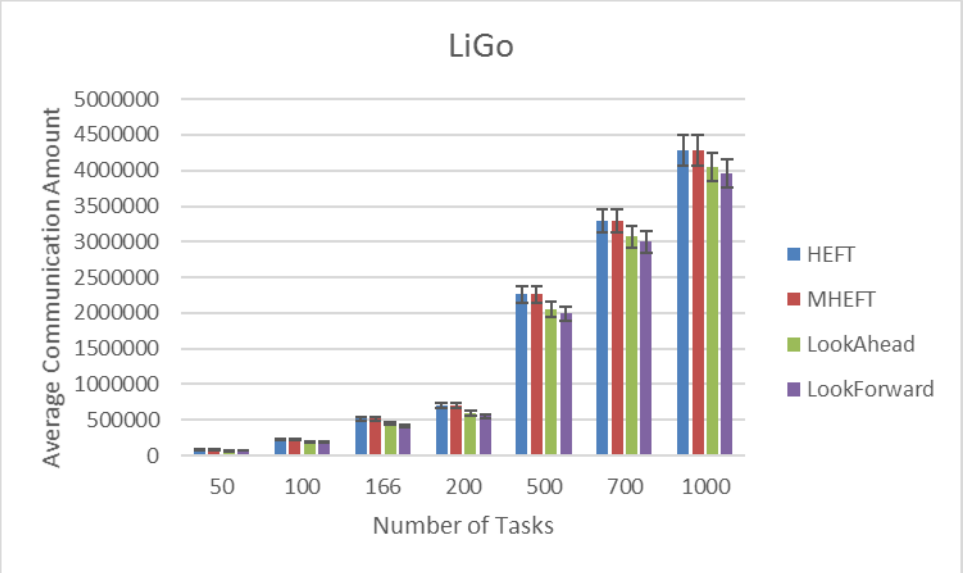

Figure 5.32: Communication amount of ligo workflow under 10 resources

5.4.4 Discussion

From the above results, some observations can be made. First, both look-ahead and look-forward algorithms can obtain improvement upon HEFT. There are two reasons to explain this: communication and DAG structure. When communication comes into task scheduling, it obviously affects the whole scheduling length. Both look-ahead and look-forward can 'foresee' some key tasks and their communications; if an algorithm finds that some communication will affect the scheduling, it will choose the best destination resource to avoid or reduce the negative effect of this communication. By applying this way step by step, the whole scheduling length can be reduced. The other reason is the DAG structure. One typical structure is the 'join structure'. When a task t has more than one incoming communication, this task is called a join task. The parent tasks of t, join task t and communications E between them constitute a join structure. Most of the list scheduling algorithms using traditional EFT, such as HEFT, cannot fully consider the join structure. However, look-ahead and look-forward consider the resources available, which makes the join task t have the earliest finish time at the stage of scheduling the parent tasks. This naturally considers the join structure and communications between them. So, both look-ahead and look-forward can reduce the scheduling length compared to basic EFT-only algorithms.

Secondly, with an increasing number of tasks, look-ahead obtains less and less benefit but look-forward obtains more and more benefit. That is because both look-ahead and look-forward use a prediction tactic to make decisions. The benefits of this prediction tactic can only be obtained when the planned future happens without interruption. Any unexpected interference can make the planned future change and waste a lot of resources. Look-ahead looks into the earliest finish time of the children tasks which involves considering the communications and structure of DAG; it satisfies the finish time of some parent tasks and makes a plan for the children tasks. The problem is that there are many tasks which have a higher priority than the planned children tasks among the parents and children tasks. These higher priority tasks will preemptively seize the resources of the planned children tasks, and then an unpredictable interruption happens. The decision-making tactic starts to make poor decisions. The larger size a DAG has, the poorer the decisions that look-ahead may potentially make. So look-ahead will get less and less benefit with the number of tasks increasing; in some cases, look-ahead can

even lead to a result worse than HEFT (see Figure 5.6). Based on these considerations, the look-forward algorithm considers not only the children tasks but also the higher priority tasks. It carefully chooses some higher priority tasks and the children tasks, using a complete way of reasoning to add these tasks to a scheduling list or to release them from it in an appropriate time frame and trying to avoid or reduce the interruptions from the higher priority tasks. Then, it uses a new prediction strategy which can consider the communications and the structure of the DAG, with generally good decision-making. That is why look-forward can keep obtaining benefits, even though the number of tasks of a DAG increases.

Thirdly, compared to HEFT, both look-ahead and look-forward can reduce the communications; look-forward reduces communication more than look-ahead. As discussed in point one of Section 5.4.4, the two algorithms 'foresee' some future tasks and avoid the costly communications among the current and future tasks; the amount of communication is reduced in this way. Both the algorithms consider the join DAG structure, which can also reduce the communications. The reasons why look-forward reduces communication further is that look-forward fixes the poor decision-making problem which has been discussed in point two of Section 5.4.4. One more reason is that look-forward considers more tasks than look-ahead which makes it more stable and potentially reduces communication more than look-ahead.

Some other observations, like the fact that look-forward remains stable even when the CCR changes, were also made.

5.4.5 Algorithm execution time

Figures 5.33 to 5.40 present the average algorithm execution time of four algorithms. Both look-ahead and look-forward need to consider more tasks than HEFT when scheduling one task; as expected, look-ahead and look-forward require more time to execute than HEFT. Because look-ahead considers only the children tasks but look-forward considers both the children tasks and some higher priority tasks, the execution time of look-forward is 1 or 2 times higher than that of look-ahead. Nevertheless, this higher execution time may be worthy for a static algorithm, especially in situations where a shorter makespan is important.

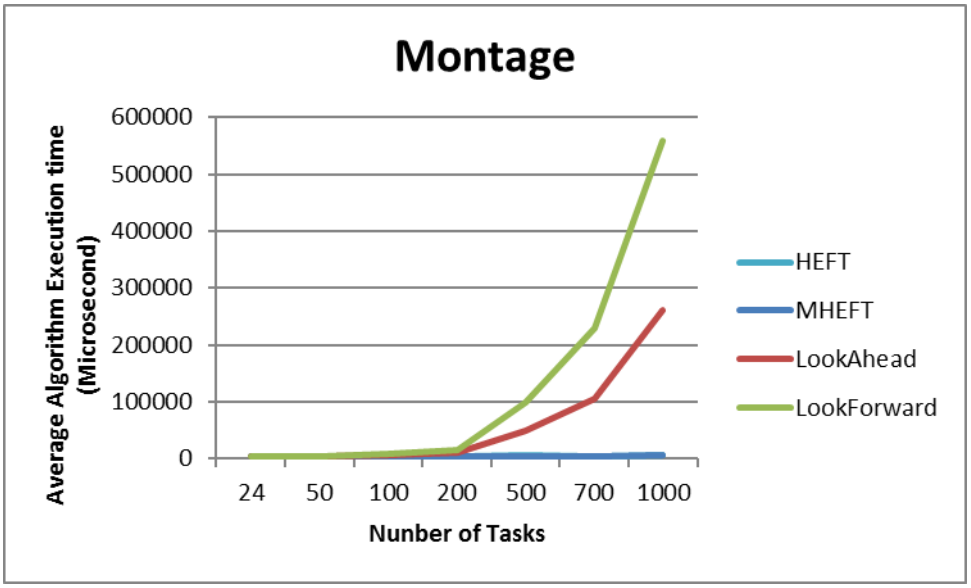

Figure 5.33: Average algorithm execution time (in microseconds) of montage workflow under 2 resources

5.5 Summary

This chapter presented a look-forward approach to schedule DAGs in heterogeneous systems. The look-forward algorithm is also a forecast algorithm but considers the higher priority and unscheduled tasks. The simulation experiments show that look-forward can significantly improve the schedule made by HEFT and look-ahead, especially in cases where there is a large number of tasks. As the number of tasks increases, look-forward shows more stable and effective performance than look-ahead, which proves that the scheduling strategy we propose is effective. The proposed algorithm can shorten the makespan in some cases by up to 40%, which is a significant improvement. Despite the increased time complexity, this time consumption can be controlled to be lower than many other higher time complexity algorithms; at the same time, it can make more reduction in schedule length than these algorithms. As this is a static algorithm, the time complexity is acceptable when considering the improvement it can make.

Further work can study how look-forward works when communication becomes

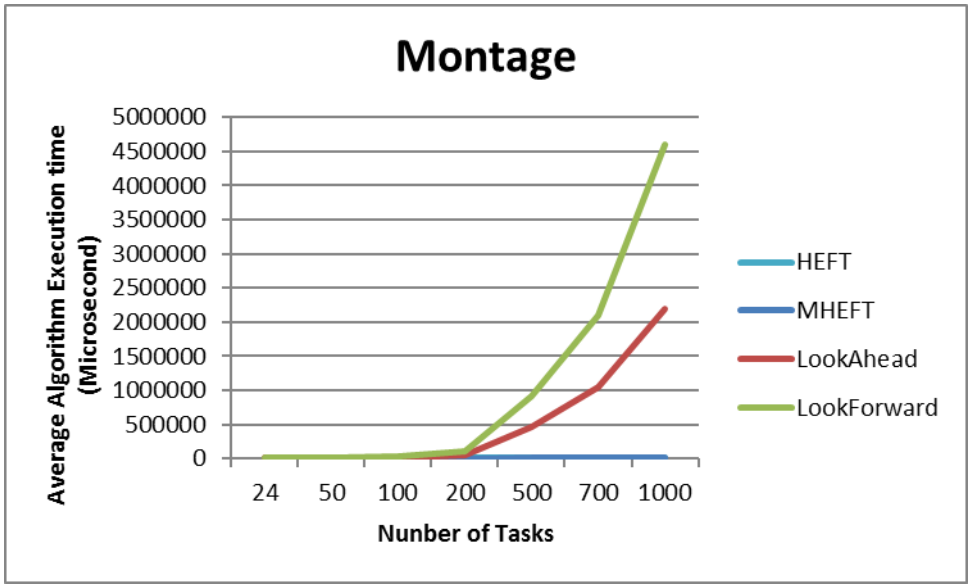

Figure 5.34: Average algorithm execution time (in microseconds) of montage workflow under 10 resources

important. One example is the contention model problem. In the contention model, algorithms need to consider communication contention during scheduling. Look-forward can 'foresee' the communications, and may solve the communication contention problem. The other problem is the identical data problem. Identical data should be considered during the scheduling, in order to find a suitable way to schedule identical data and obtain the benefit. Look-forward may be a good algorithm to do it. These problems will be considered in the following chapters.

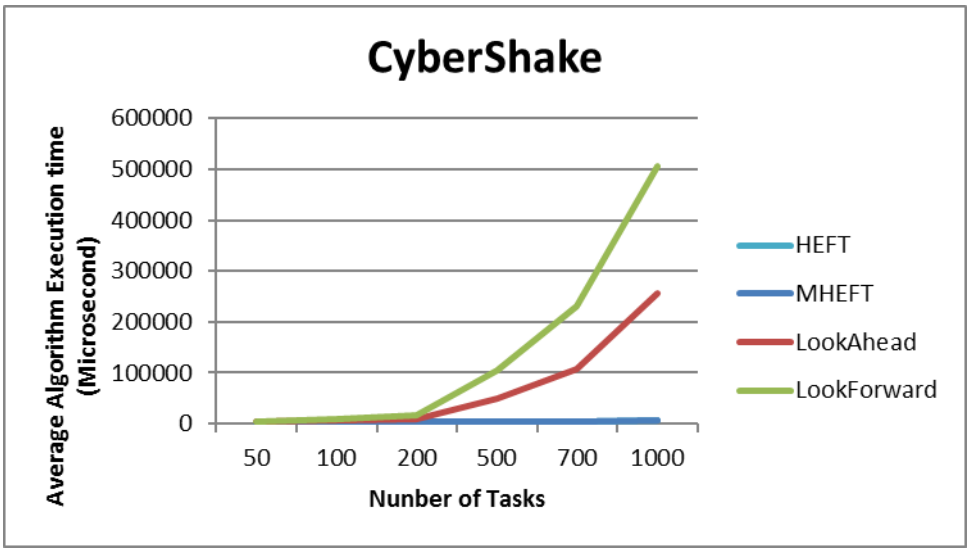

Figure 5.35: Average algorithm execution time (in microseconds) of cybershake workflow under 2 resources

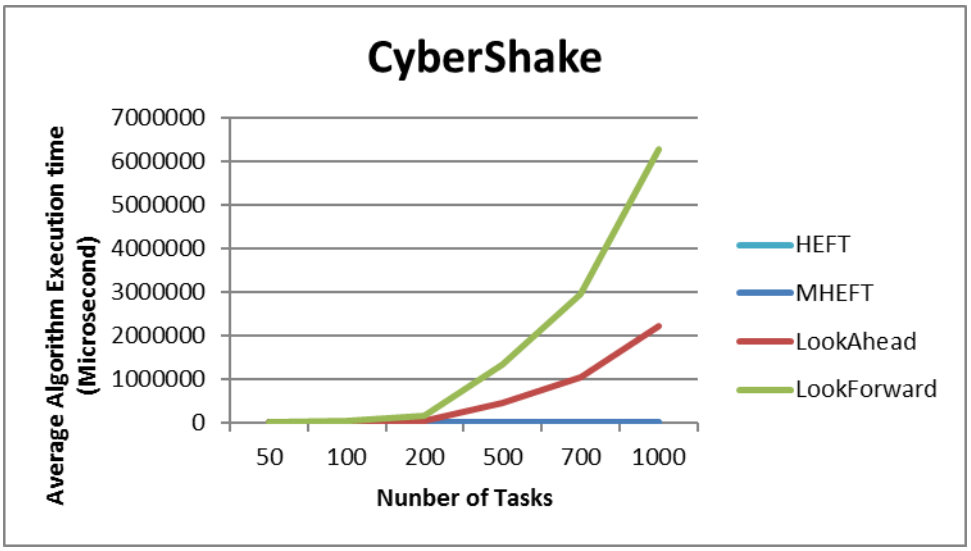

Figure 5.36: Average algorithm execution time (in microseconds) of cybershake workflow under 10 resources

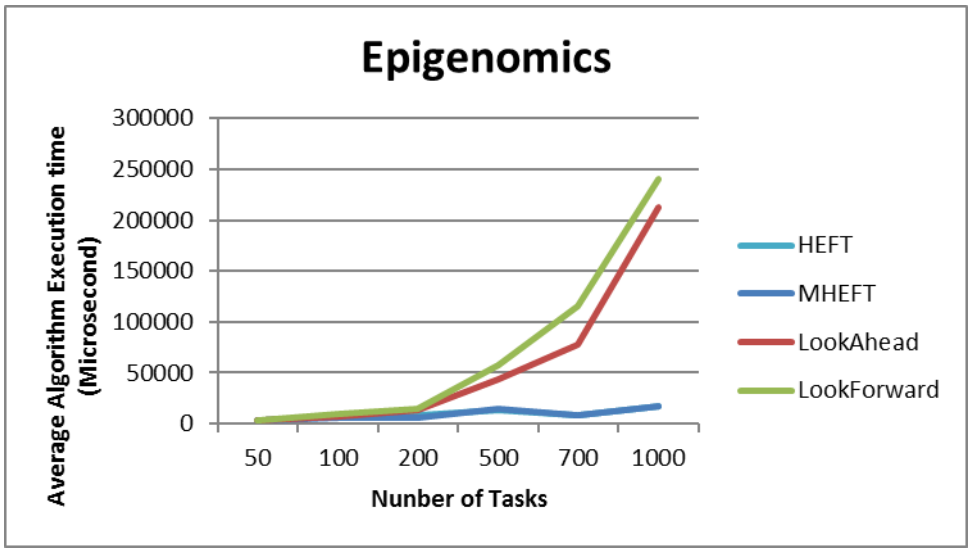

Figure 5.37: Average algorithm execution time (in microseconds) of epigenomics workflow under 2 resources

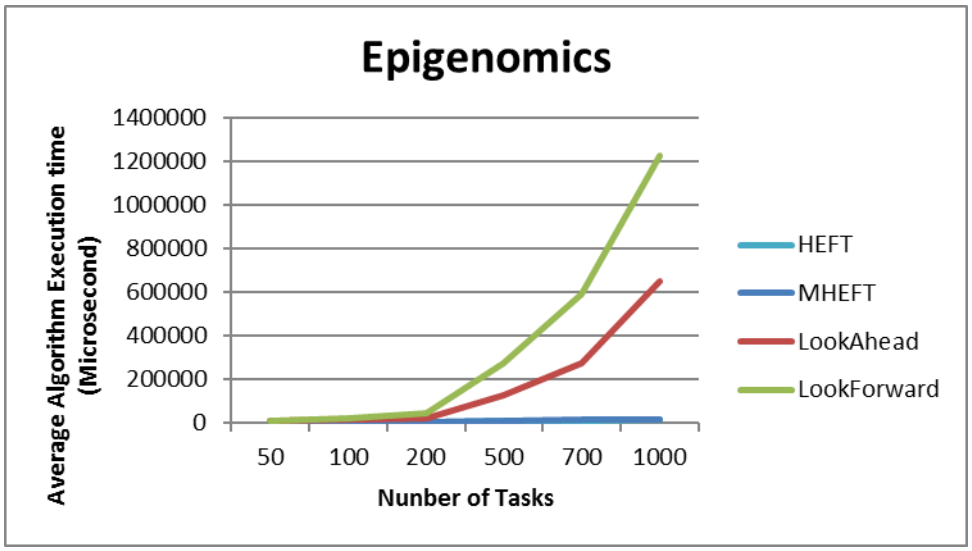

Figure 5.38: Average algorithm execution time (in microseconds) of epigenomics workflow under 10 resources

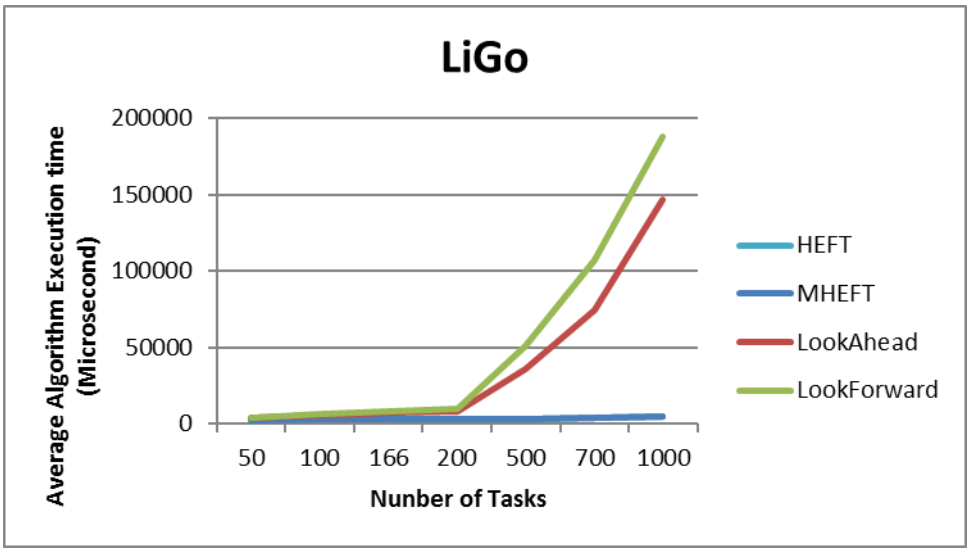

Figure 5.39: Average algorithm execution time (in microseconds) of ligo workflow under 2 resources

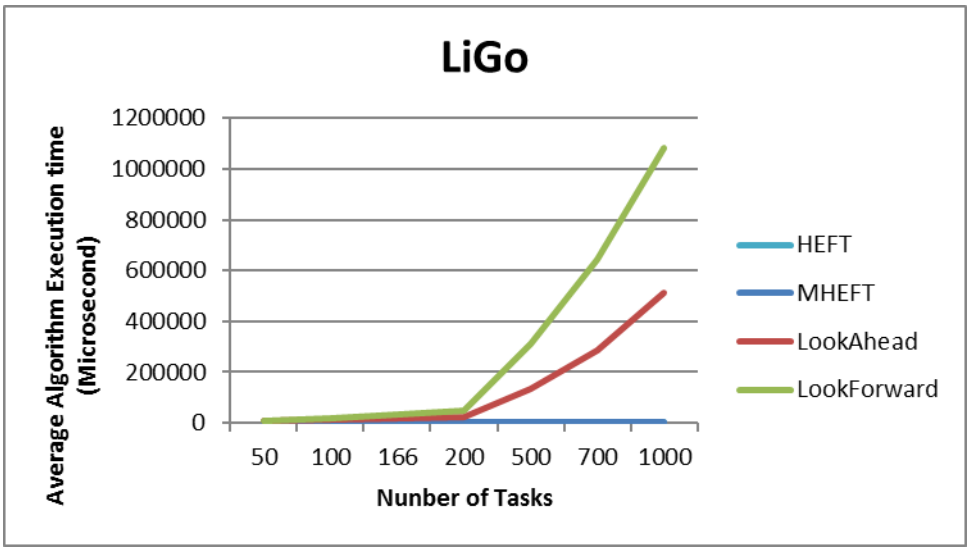

Figure 5.40: Average algorithm execution time (in microseconds) of ligo under 10 Resources

Chapter 6

Look-forward scheduling in the contention model

Section 3.2 has provided an introduction to the contention model, has detailed the differences between the contention model and the classic model and described one important contention model called the one-port model. In this chapter, scheduling in the contention model will be studied further.

6.1 List scheduling in the one-port model

As there are many differences between the classic model and the contention model when it comes to handling contention, new algorithms which take into account the properties of the contention model, should be developed. In this work, the one-port model is chosen as the contention model. The main differences between the one-port model and the classic model are that communication cannot be performed in parallel under the one-port model and that any communication must pass through an out-port of the sender processor and an in-port of the receiver processor. As communication cannot be performed in parallel, any communication should consider the conflict with other data transfers. The communication may need to wait for the processor to finish transferring the other data, which will increase the whole schedule length. Besides, the out-port and in-port on the path of each communication transfer may have different slack time which will require the relevant algorithms to handle the out-port and in-port separately. As a

result of the communication transfers from the out-port to the in-port, the start time of the communication on the in-port cannot be earlier than the start time on the out-port.

In order to solve these problems, a technique called edge scheduling [SSS06, WSF89, GDP08] (see Section 2.2) needs to be introduced first. Compared with scheduling in the classic model, the edges in the one-port model should be scheduled carefully. (Note that this does not mean edges in the classic model need not be scheduled. Because one edge e_{ij} in the classic model needs only a start time $t_s(e_{ij}) = t_f(p_i)$ and a finish time $t_f(e_{ij}) = t_s(e_{ij}) + w(e_{ij})$, which is easy to determine, it is not common to refer to edge scheduling in the classic model). The edge scheduling technique schedules an edge e_{ij} between two processors p_i and p_j, which will consider the conflicts between the communications and the transfer on the out-port of p_i and the in-port of p_j. The generic algorithm of edge scheduling can be seen in Algorithm 6.1.

In Algorithm 6.1, $t_s(e_{ij}, p_{(i,out)})$ means the start time of edge e_{ij} on processor p_i's out-port, $t_f(e_{ij}, p_{(i,out)})$ means the finish time of edge e_{ij} on processor p_i's out-port, $t_s(e_{ij}, p_{(j,in)})$ means the start time of edge e_{ij} on processor p_j's in-port, $t_f(e_{ij}, p_{(j,in)})$ means the finish time of edge e_{ij} on processor p_j's in-port, $t_f(p_{(i,out}$ means the finish time of processor p_i's out-port, $t_f(p_{(i,in}$ means the finish time of processor p_i's in-port. This algorithm schedules edge e_{ij} on the out-port and in-port separately. $t_s(e_{ij}, p_{(i,out)})$ should be the later of processor p_i's finish time and the out-port of processor p_i's finish time $t_f(p_{(i,out)})$). $t_s(e_{ij}, p_{(j,in)})$ should be the later of $t_s(e_{ij}, p_{(i,out)})$ and the in-port of processor p_j's finish time $t_f(p_{(j,in)})$. When the edge is scheduled on the out-port or in-port, the completion of the out-port or in-port should be renewed. Note that p_i means the allocated processor of edge e_{ij}'s incoming node, p_j means the allocated processor of edge e_{ij}'s outgoing node, $p_{(i,out)}$ means the out-port of processor p_i and $p_{(j,in)}$ means the in-port of processor p_j. At the very beginning, all the start time and finish time of processors, ports, nodes and edges are zero.

Algorithm 6.1 gives a way to schedule one edge e_{ij} from processor p_i to processor p_j. Some nodes have more than one incoming edge, for example in Figure 6.1, node OUT has 10 incoming edges. According to the basic steps of list scheduling, before scheduling node OUT, the finish time and data ready time (DRT) of OUT should be determined. When scheduling node OUT, all the 10 incoming edges should be scheduled. The following algorithm shows how to get the DRT for one node n_i on one processor p_i. In Algorithm 6.2, the $Incom(n_i)$ means all the incoming edges of n_i, $proc(n_j)$ means

Algorithm 6.1 Edge Scheduling. e_{ij} from processor p_i to processor p_j (one-port model)

1: Input: processor p_i, processor p_j, edge e_{ij}
2: Output: finish time of processor p_i's out-port $t_f(p_{(i,out)})$
3: Output: start time of edge e_{ij} on p_i's out-port $t_s(e_{ij}, p_{(i,out)})$
4: Output: finish time of edge e_{ij} on p_i's out-port $t_f(e_{ij}, p_{(i,out)})$
5: Output: finish time of processor p_j's in-port $t_f(p_{(j,in)})$
6: Output: start time of edge e_{ij} on p_j's in-port $t_s(e_{ij}, p_{(j,out)})$
7: Output: finish time of edge e_{ij} on p_j's in-port $f_s(e_{ij}, p_{(j,out)})$
8: Get finish time $t_f(p_i)$ of p_i
9: Get finish time $t_f(p_{(i,out)})$ of p_i's out-port
10: $t_s(e_{ij}, p_{(i,out)}) \leftarrow \max(t_f(p_i), t_f(p_{(i,out)}))$
11: $t_f(e_{ij}, p_{(i,out)}) \leftarrow t_s(e_{ij}, p_{(i,out)}) + c(e_{ij})$
12: $t_f(p_{(i,out)}) \leftarrow t_f(e_{ij}, p_{(i,out)})$
13:
14: Get finish time $t_f(p_j)$ of p_j
15: Get finish time $t_f(p_{(j,in)})$ of p_j's in-port
16: $t_s(e_{ij}, p_{(j,in)}) \leftarrow \max(t_s(e_{ij}, p_{(i,out)}), t_f(p_{(j,in)}))$
17: $t_f(e_{ij}, p_{(j,in)}) \leftarrow t_s(e_{ij}, p_{(j,in)}) + c(e_{ij})$
18: $t_f(p_{(j,in)}) \leftarrow t_f(e_{ij}, p_{(j,in)})$

the allocated processor of edge e_{ji}'s incoming node n_j, $t_f(e_{ji}, proc(n_j), p_i)$ means the finish time of edge e_{ji}, $p_{(i,in)}$ means the in-port of processor p_i, $t_f(e_{ji}, p_{(i,in)})$ means the finish time of edge e_{ji} on the in-port of processor p_i. The DRT (see Equation 2.1) should be the time after all the incoming edges are scheduled. This algorithm uses the equation $DRT = max(t_f(e_{ji}, proc(n_j), p_i), DRT)$ to get the final data ready time of the target node n_i in each loop. Compared to the classic model (see Algorithm 4.1), this algorithm schedules the incoming edges and considers the contention between the out-port and in-port of the processor in the one-port model. Each edge may have an effect on the final data ready time in the one-port model.

After getting the data ready time, the start time $t_s(n_i, p_i)$ of a node n_i on processor p_i is given as $t_s(n_i, p_i) = max(DRT, t_f(p_i))$. This formula gives a way of finding n_i's start time on a specific processor p_i. In a parallel/distributed system, there are many processors to which a node can be allocated. In order to find one suitable processor for

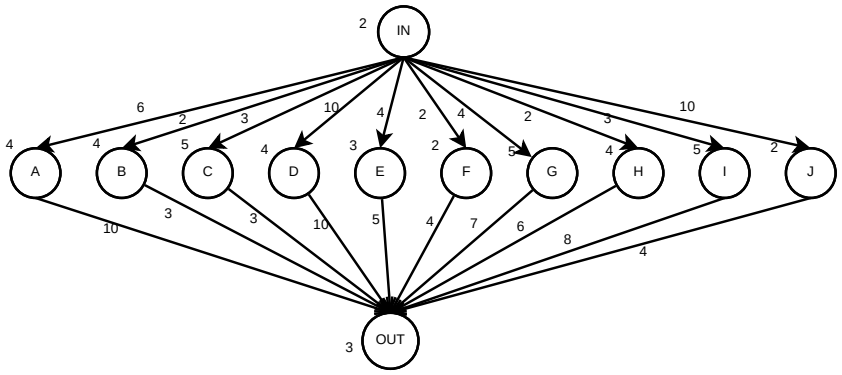

Figure 6.1: Graph example for edge scheduling

Algorithm 6.2 Get the DRT of n_i on processor p_i (one-port model)

1: Input: processor p_i, node n_i
2: Output: data ready time (DRT) of node n_i
3: DRT $\leftarrow 0$
4: Find all incoming edges $Incom\{n_i\}$ of n_i
5: **for all** edge $e_{ji} \in Incom\{n_i\}$ **do**
6: Schedule edge e_{ji} on out-port of $proc(n_j)$
7: Schedule edge e_{ji} on in-port of p_i
8: Finish time $t_f(e_{ji}, proc(n_j), p_i) \leftarrow t_f(e_{ji}, (p_{(i,in)}))$
9: DRT $\leftarrow \max(t_f(e_{ji}, proc(n_j), p_i)$, DRT)
10: **end for**

node n_i, the algorithm needs to check all the processors and find the earliest start time amongst these processors. The processor, which allows node n_i to be started earliest, will be chosen as n_i's processor. Algorithm 6.3 shows how to choose a processor p for node n_i. No matter whether in the classic model or in the one-port model, both algorithms use the principle of choosing the minimal start time first. So, from Algorithm 4.2 in classic model and Algorithm 6.3 in the one-port model, it is easy to see that these two algorithms are similar in every way, except in how to obtain the data ready time for a node n_i. Every time the algorithm checks a processor and the start time and finish time of all the processors and processor's ports, it should go back to the previous status when Algorithm 6.3 begins. That is because Algorithm 6.3 only chooses a processor for node n_i and it will not change any other status of the nodes and processors.

Algorithm 6.3 Choose a processor p_i for node n_i (one-port model)

1: Input: node n_i
2: Output: processor p_i for node n_i
3: $t_{temp} \leftarrow \infty$, $p_{temp} \leftarrow null$
4: **for all** processors $p_i \in P$ **do**
5: Get the DRT of n_i on processor p_i (Algorithm 6.2)
6: **if** $t_{temp} >$ max(DRT, $t_f(p_i)$) **then**
7: $t_{temp} \leftarrow$ max(DRT, $t_f(p_i)$)
8: $p_{temp} \leftarrow p_i$
9: **end if**
10: **end for**
11: $t_s(n_i) \leftarrow t_{temp}$, $proc(n_i) \leftarrow p_{temp}$

Algorithm 6.4 shows the steps of the static list scheduling algorithm in the one-port model. It has the same steps as the basic steps of static list scheduling, except for the priority part and how to choose a processor for a node n_i. In fact, the greatest differences between the one-port model and the classic model are edge scheduling (Algorithm 6.1) and data ready time (Algorithm 6.2).

Algorithm 6.4 List scheduling (one-port model)

1: Calculate the priority of each node n_i as $\sum_{j \in pred(i)} c(e_{j,i})/w_i$
2: Order all nodes $n_i \in V$ into list L by priority and precedence constraints (non-ascending order)
3: **for all** node $n_i \in L$ **do**
4: Choose a processor p_i for node n_i (Algorithm 6.3)
5: Schedule n_i on processor p_i
6: **end for**

6.2 Look-forward algorithms under the contention model

Even though most list scheduling algorithms can be converted into the contention model by using Algorithms 6.4, 6.3 and 6.2, this kind of conversion is still not enough. Because of the communication contention, the resource competition moves from the processor to the communication network. Using the earliest finish time is no longer an efficient way to do the scheduling. When doing scheduling, the algorithm needs to consider both the incoming port and the outgoing port. A block in one port may lead to communication contention. In order to solve this problem, we propose to add a look-forward aspect to the contention model. It has been demonstrated in Chapter 5, even in the classic model, look-forward still has better performance. Because look-forward typically 'looks' a little farther than the other algorithms, it can consider the whole situation rather than only finding an earlier time. Algorithm 6.5 shows the detail of how to use the look-forward algorithm in the contention model. 'Status 1' means the status before the scheduling; EFT_{r_i} means the earliest finish time of task r_t.

6.3 Experimental setup

6.3.1 Graphs

In order to evaluate the algorithms, six types of DAGs (directed acyclic graphs) are chosen and introduced as follows:

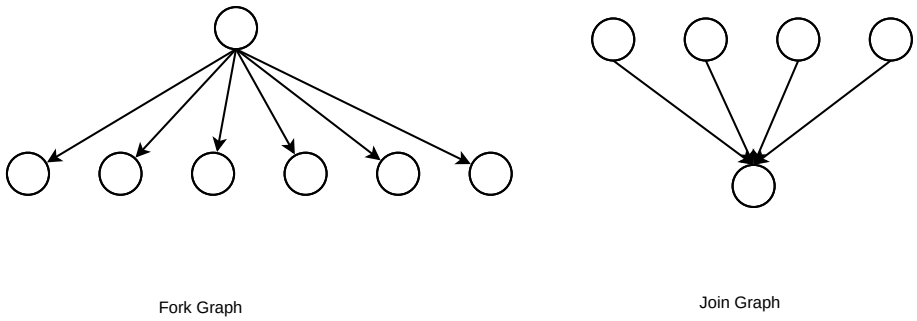

Figure 6.2: Fork and join graph

Algorithm 6.5 Look-forward algorithm, contention model

1: Input: a task graph $G = (V, E, w, c)$
2: Output: a schedule for all tasks $t_i \in V$
3: rank tasks using $t_i \in V$ the $rank_u$
4: $L \leftarrow NULL$
5: **while** there are unscheduled tasks **do**
6: $t \leftarrow$ unscheduled task with highest $rank_u$
7: **if** $t \notin L$ and children of $t \notin L$ **then**
8: $L \leftarrow$ children of t
9: **end if**
10: **if** $t \in L$ **then**
11: $L \leftarrow NULL$
12: $L \leftarrow$ children of t
13: **end if**
14: status 1
15: **for all** resource $r_i \in$ the resources set R **do**
16: schedule t on r_i using Algorithm 6.4
17: schedule all tasks $\in L$ using Algorithm 6.4
18: $EFT_{r_i} \leftarrow$ maximum EFT for tasks $\in L$
19: return to status 1 (recover the ports)
20: **end for**
21: schedule t on r_i such that $EFT_{r_i} \leq EFT_{r_k}$ using Algorithm 6.4
22: **end while**

Figure 6.2 shows the example of a fork graph and a join graph. A fork graph consists of a node with two or more outgoing edges and child nodes. If a fork node's outgoing edges all represent the same data, these edges will be called edges with identical data. The join graph is a node with two or more incoming edges and parent nodes. It is easy to see that it is only meaningful to have identical data for fork-graphs or fork parts of a graph. Identical data taken by a join graph would not be realistic. So, in this work, the join graph is not tested by the algorithm. Figure 6.3 shows the example of a fork-join graph. A fork-join graph is the combination of a fork graph and join graph. The fork part of the graph is connected by a join part. A fork-join graph is an important graph for

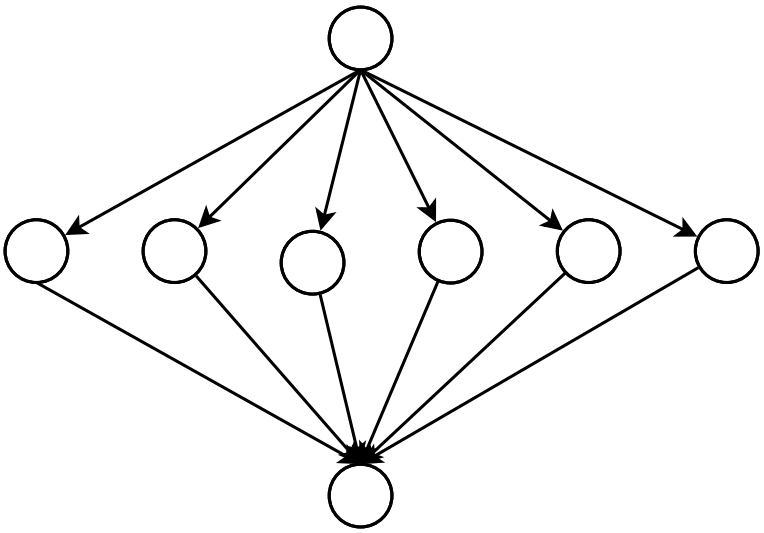

Figure 6.3: Fork-join graph

parallel task scheduling.

Figure 6.4 shows the example of a series-parallel graph. Before series-parallel is defined, the two-terminal graph needs to be introduced. A two-terminal graph is a graph with two distinguished vertices, s and t, called source and sink, respectively. Let X be a two-terminal graph, s_x,t_x are the source and sink vertices of X. Let Y be a two-terminal graph, s_y, t_y are the source and sink vertices of Y. If a composition is operated by merging s_x,s_y to be a new source vertex of Z and merging t_x, t_y to be a new sink vertex of Z, this composition is called parallel composition. If a composition is operated by merging t_x, s_y and s_x, t_y become the source and sink vertices of Z, this composition is called series composition. If a graph is constructed by a sequence of series composition or parallel composition, this graph is a series-parallel graph. The example can be seen in Figure 6.4. Figure 6.5 shows the example of an out-tree. Figure 6.6 shows the example of an in-tree.

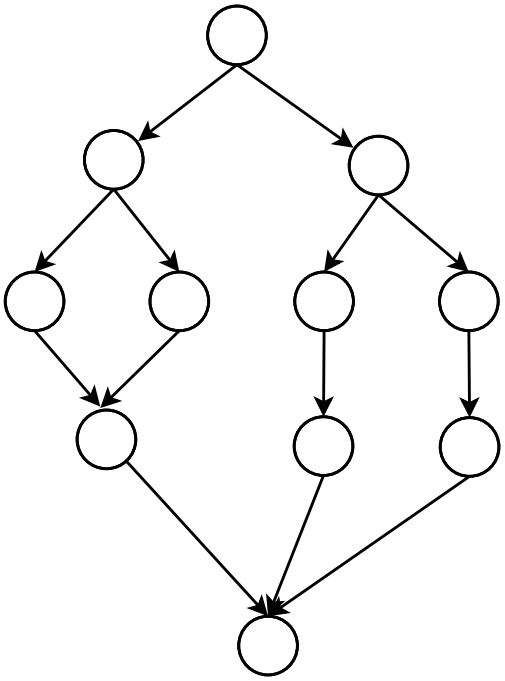

Figure 6.4: Series-parallel graph

6.3.2 DAX generator and simulator settings

As mentioned in Section 5.4.1, DAX represents the input workflow file which contains the information of tasks, input and output files, task runtime, communication file and the relationship between tasks and files. The task runtime can be used to calculate the computation time of this task and the communication file can be used to calculate the communication cost during different target systems. A DAX file is generated by the workflow generator (see Section 5.4.1). The generator generated the DAX files with the corresponding computation cost and communication cost from the interval (20, 1000). Three different CCR values (0.1, 1.0, 10) are generated for the experiments; in this experiment, the six kinds of DAXs mentioned in Section 6.3.1 are generated by the workflow generator. All the workflows are run by HEFT and look-forward algorithms. Each algorithm runs 200 randomly generated workflows under 2 resources and 10 resources for each type of workflow.

A simulator called WorkflowSim is used to simulate the running of the workflows.

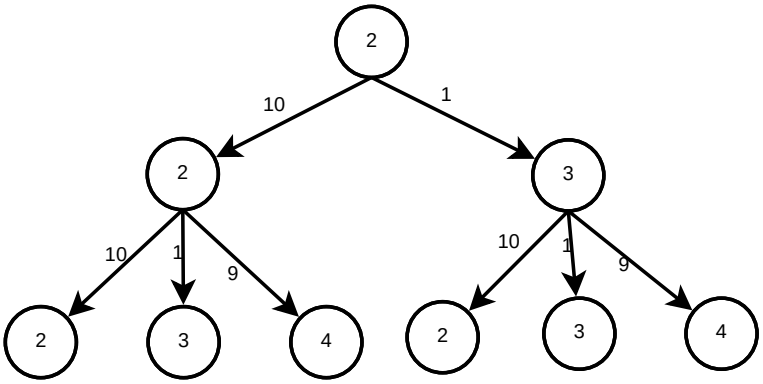

Figure 6.5: Out-tree graph

The details of WorkflowSim can be found in Section 5.4.2. DAXs are used as the input file of WorkflowSim. The HEFT planning algorithm and the look-forward based planning algorithm under the contention model are implemented to do scheduling in the simulator.

6.3.3 Results and evaluation

As shown in Figures 6.7, 6.8, 6.9, 6.10, 6.11, three observations can be made as follows: firstly, when the CCR is small, for instance CCR=0.1 (see Figures 6.7, 6.8, 6.9, 6.10, 6.11), both HEFT scheduling and look-forward scheduling have similar results. That is because when the CCR is small, the communications are very small too. Under these conditions, scheduling is more likely to be affected by the computation costs of the nodes; the communications are less important to the scheduling system. Secondly, when the CCR is large, for instance CCR=10, look-forward works better than HEFT under the join structures, such as in the In-Tree graph and fork-join graph (see Figures 6.8, 6.10). The reason why look-forward works better is that look-forward considers both the incoming communications and the outgoing communications of each node. This will be useful, especially in a join-structure. Lastly, both the HEFT algorithm and the look-forward algorithm can be transferred from the classic model to the contention model using Algorithms 6.5 and 6.4. Both of them can create good schedules under the one-port/contention model.

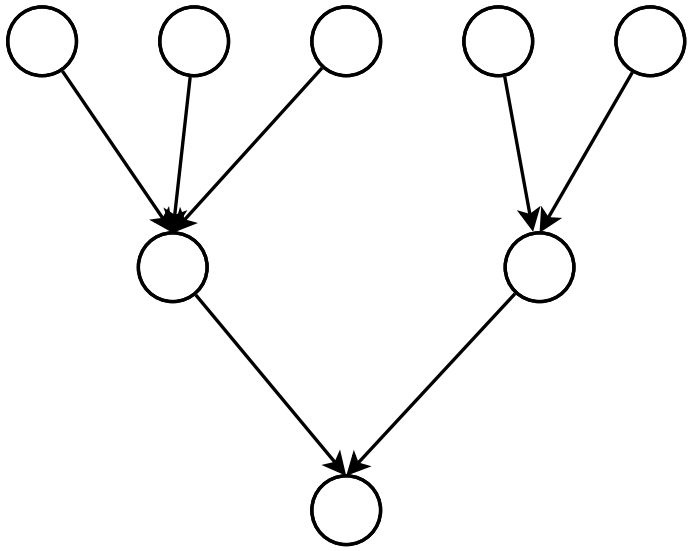

Figure 6.6: In-tree graph

6.4 Summary

This chapter has investigated the look-forward algorithm under the contention model. The next chapter will analyze this further taking into account an additional property of task graphs concerning identical data.

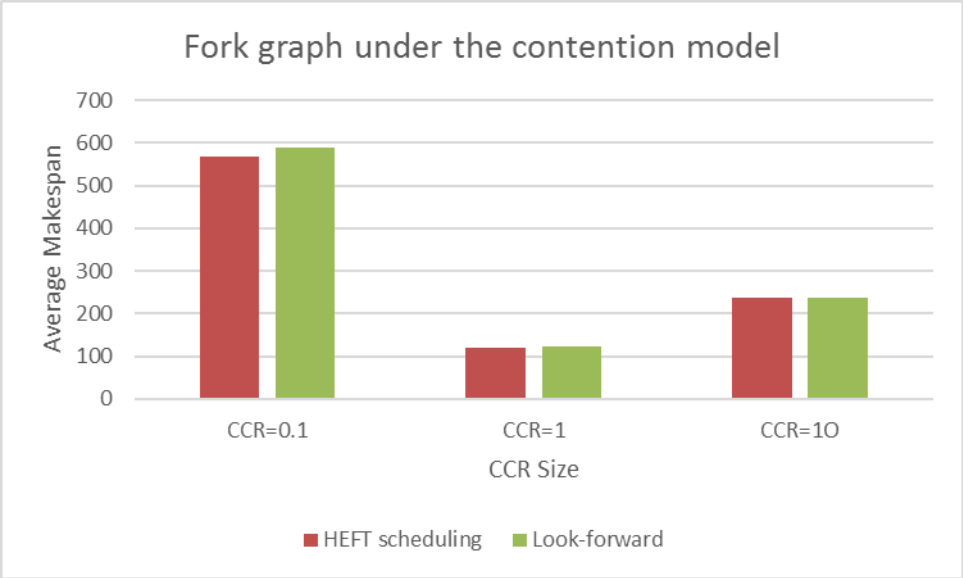

Figure 6.7: Fork graph under the contention/one-port model

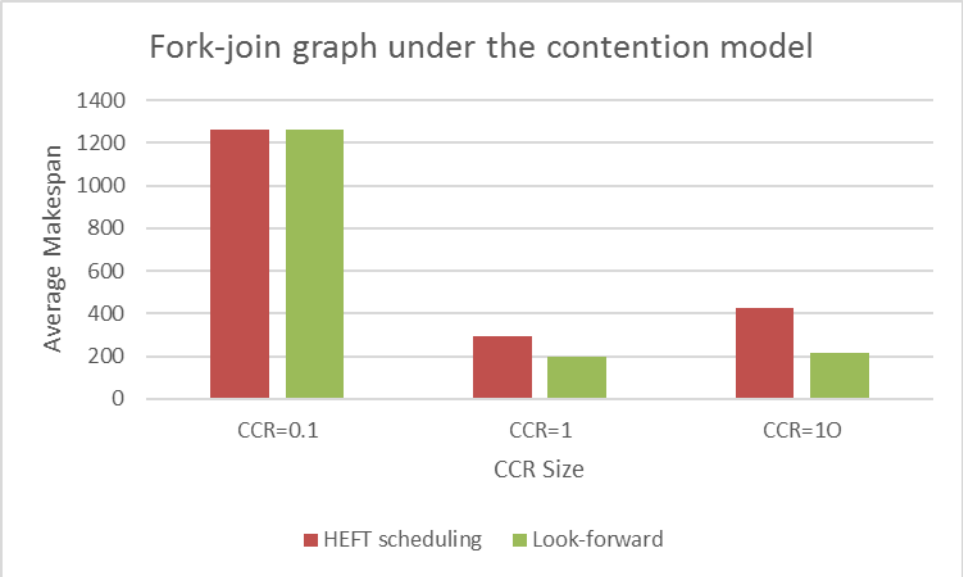

Figure 6.8: Fork-join graph under the contention/one-port model

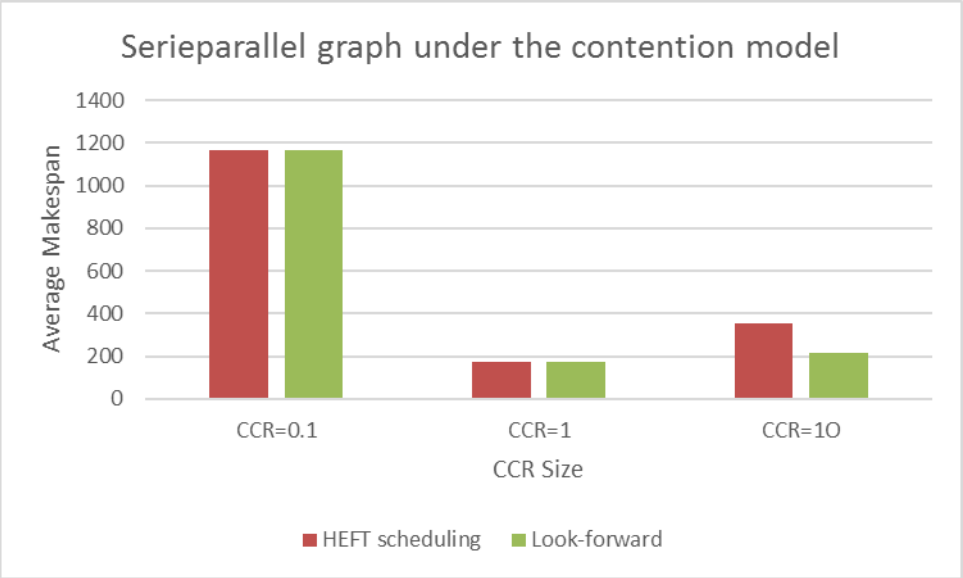

Figure 6.9: SerieParallel graph under the contention/one-port model

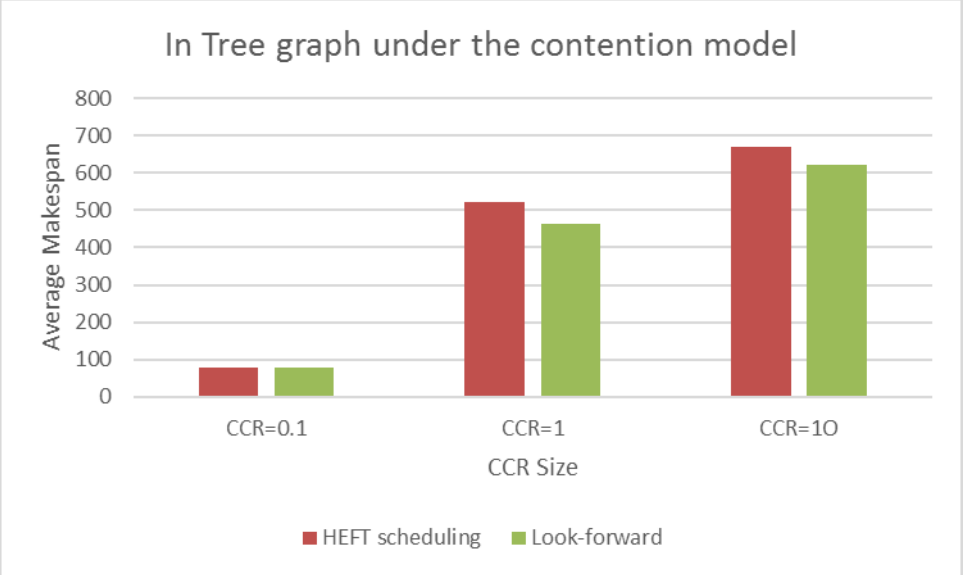

Figure 6.10: In Tree graph under the contention/one-port model

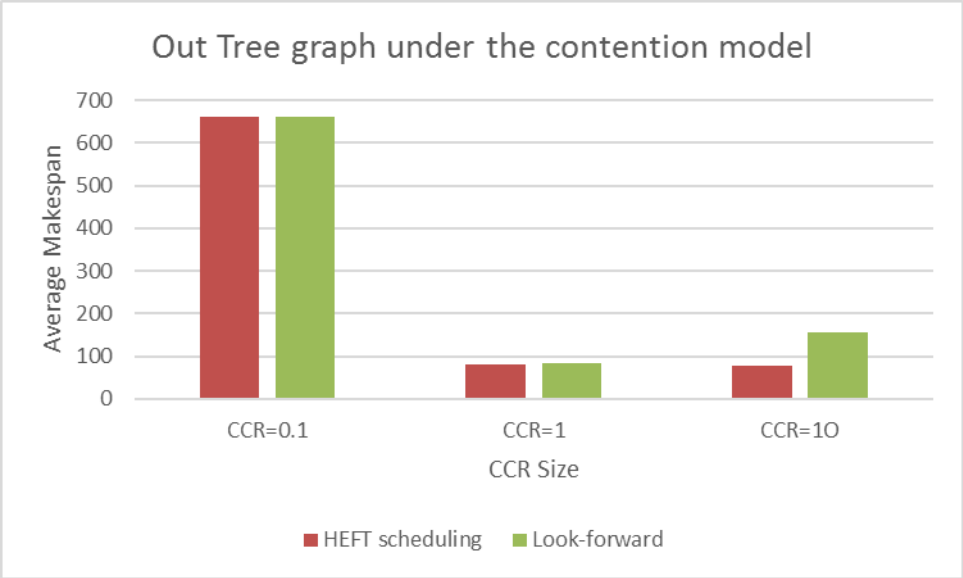

Figure 6.11: Out Tree graph under the contention/one-port model

Chapter 7

The identical data problem

In scheduling algorithms for parallel computing, communication between tasks plays an important role in the achievable performance. When a task needs to send the same data to more than one task, this data is called *identical data*. Identical data will be sent many times to different target tasks, each of which will receive the same data. If some of the target tasks are on the same processor, the processor only needs to receive the identical data once. Based on this property, this research takes into account identical data in scheduling algorithms in order to reduce the schedule length for task scheduling. As already known, in parallel or distributed systems, when communication costs and data transmission contention are both considered, it becomes difficult to find an efficient schedule for tasks. Clearly, identical data relate to the outgoing data, which is data that goes out from a source task. An algorithm needs to consider incoming data too to find the earliest finish time of tasks while the use of identical data in outgoing data is also considered.

Most of the existing scheduling approaches only care about the incoming edge for a task. This is because these algorithms prefer to execute a target task as soon as possible. As mentioned, the look-forward algorithm allocates a task by the earliest finish time of the children tasks and some other low-priority tasks. When look-forward is applied, it automatically considers the incoming edges of one task and the outgoing edges between this task and its children tasks. This means that, naturally, the look-forward algorithm can be used for the identical data problem too.

7.1 The identical data problem description

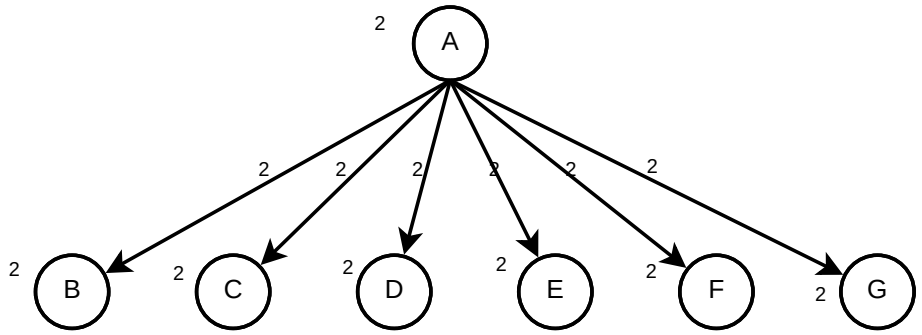

Figure 7.1: Identical data

Consider Figure 7.1, which shows a fork graph. Node A sends data to nodes B, C, D, E, F and G. All the communication costs are assumed to be 2 (units do not matter in this discussion). When all data transmitted from node A to children nodes are the same, these data are called identical data. Note that this definition is based on content and not on the communication cost. Sending identical data can occur in any graph, not only fork graphs, whenever a node has at least two out-edges. To consider identical data in the scheduling model, the following new property is introduced:

1. No repetitive transfer. This means that if one processor p_1 transfers several identical data items to another processor p_2, p_1 needs only to transfer these identical data only once to p_2 and p_2 only needs to receive these identical data from p_1 only once.

This property is appropriately relevant to the contention model and the one-port model. When one processor p_1 sends more than one data item to a processor p_2 under the one-port model at the same time, for the reason that p_1 has only one out-port and this out-port can only send one data item at a time, there would be contention on the out-port of p_1; also there would be contention on the in-port of p_2, which will have an in-port conflict when receiving data from p_1. So p_1's out-port and p_2's in-port need to send and receive these data items one by one, something which will expand the schedule length. However, if these data are identical data, according to the new property of no repetitive transfer (see identical property 1), these data will need to be transferred only

once, both on the out-port of p_1 and the in-port of p_2 and the scheduling length will be reduced. In the classic model, the communication can be performed in parallel which means even if one processor p_1 sends many data items to processor p_2, these data can be transferred in parallel without effect to each other and the whole schedule length will not be increased, as in the one-port model. As the identification of identical data implies only one transfer to another processor, this is an efficient way to reduce the scheduling length in the one-port model.

For example, assuming that the nodes of Figure 7.1 are scheduled in alphabetic order and are allocated to the processor in a round-robin fashion starting with p_1, the following examples show the schedule of Figure 7.1 with identical data.

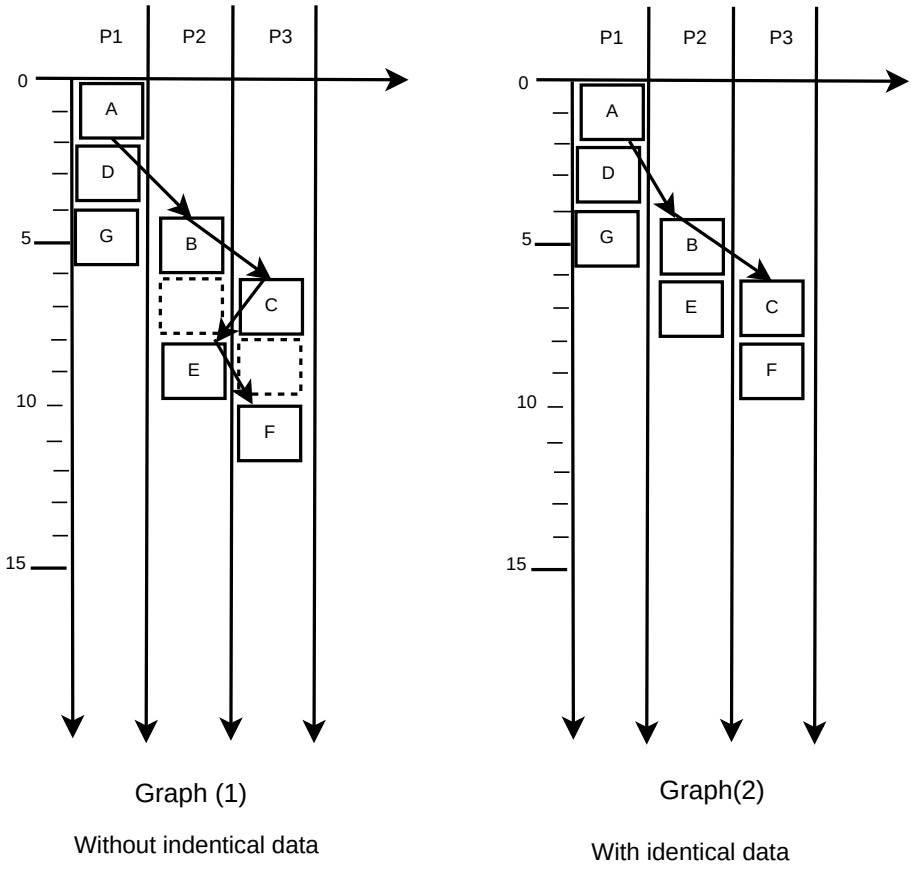

Figure 7.2: Identical data in one-port model

Figure 7.2 gives a scheduling of Figure 7.1. In Figure 7.2, Graph (1) shows the schedule without considering identical data. Graph (2) shows the schedule considering

the identical data. The arrows indicate the communication from node A to the other nodes. The start of the arrow corresponds to the start time of this communication; the end of the arrow corresponds to the finish time and destination of this communication. The number of arrows represents the number of data items that should be sent by node A. In graph (1), node c needs to wait for processor p_1 to send data identified by the communication edges $w(e_{AB})$ and $w(e_{AC})$ one by one. That is because the target system is the one-port model; the contention on the out-port of p_1 needs to be considered. In this model, there is a delay of four time units for node C. In the same way, node E should wait the transfer of $w(e_{AB})$, $w(e_{AC})$ and $w(e_{AE})$. So there will be a two time units gap between nodes B and E. Similarly, there is a delay of two time units between nodes C and F. The whole schedule length of Graph (1) in Figure 7.2 is 12. When considering identical data, the processor p_1 only needs to send the data once to processors p_2 and p_3. Even though they are in the one-port model, node E and F do not need to wait for the data $w(e_{AE})$ and $w(e_{AF})$. So there would be no gaps between nodes B and E and nodes C and F. Compared to Graph (1), the whole schedule length can be reduced from 12 to 10. The more out-degrees node A has, the more the schedule length can be reduced.

Figure 7.3 shows a schedule for the graph in Figure 7.1. The difference between Figure 7.3 and Figure 7.2 is that the nodes in Figure 7.2 are scheduled in a contention/one-port model in order to show that the property of identical data is appropriate for the contention model or the one-port model but the nodes in Figure 7.3 are scheduled in a classic model in order to demonstrate that the property of identical data is not appropriate for the classic model. Figure 7.3 Graph (1) shows the schedule without considering the identical data. Graph (2) gives the schedule considering the identical data. The arrows have the same meaning as in Figure 7.3. The number of the arrows indicates how many data items should be sent by node A. As shown in Graph (2), when considering identical data, processor p_1 does not need to send communication edges $w(e_{AE})$ and $w(e_{CF})$. Processor A needs only to transfer two data items to the other processors. In Graph (1), processor p_1 needs to send four communications to the other processors. Even though Graph (1) and Graph (2) have different communication to be transferred these two schedules have the same length. This is because processor p_1 is in the classic model; no matter how many data it should send at a time, all these data can be sent in parallel, without any interference among them.

In conclusion, identifying identical data will be useful in the contention or one-port

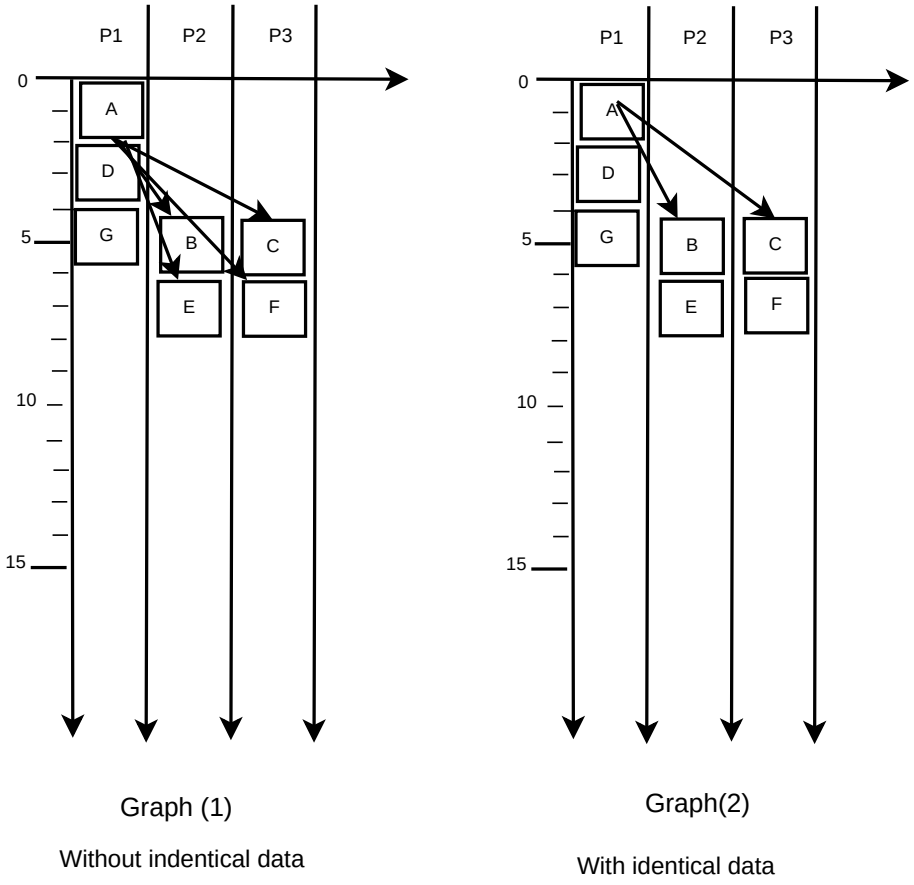

Figure 7.3: Identical data in classic model

model but will not make a difference in the classic model. The following algorithm, when considering identical data, will always assume that the contention or one-port model is used.

7.2 Identical data in list scheduling algorithms

To include the consideration of identical data in the list scheduling algorithm, a technique called 'sibling checking' is introduced in this section. When scheduling an edge e_{ij} from processor p_i to p_j, the algorithm has to check whether e_{ij} contains identical data. This step can be done by an identical tag in the algorithm. If e_{ij} contains identical data, then the algorithm needs to know whether e_{ij} needs to be scheduled. This

is because if any other identical data has been scheduled to be sent from processor p_1 to processor p_j, e_{ij} does not need to be scheduled again. For example, in Figure 7.4, when edge e_{AE} is being scheduled, and if the algorithm knows edge e_{AB} has been scheduled, then the algorithm will know edge e_{AE} does not need to be scheduled again. The sibling checking technique is a way to take into account identical data and address this problem. When the algorithm is scheduling an edge e_{ij}, node n_i and node n_j are the incoming node and outgoing node of e_{ij} respectively, the sibling checking technique will check all n_j's sibling nodes whose parent node is n_i; if any of these nodes are scheduled on the same processor $proc(n_j)$ with n_j, e_{ij} does not need to be checked again. For example, in Figure 7.4, when the algorithm is scheduling edge e_{AF}, the sibling checking will check all of node F's siblings nodes B, C, D, E, G. If it finds that node C and node F have the same parent node A, then it will tell the algorithm that edge e_{AF} does not need to be scheduled.

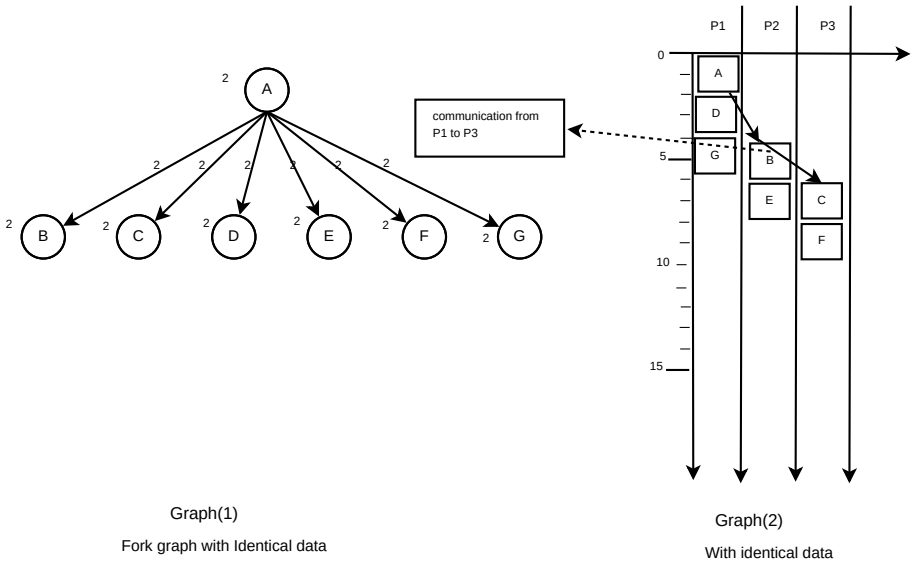

Figure 7.4: Schedule identical data

Algorithm 7.1 shows how to use the sibling checking technique to schedule an edge in the one-port model. In Algorithm 7.1, e_{ij} is the edge that needs to be scheduled. Nodes n_i, n_j are the incoming and outgoing nodes of e_{ij}. $proc(n_j)$ is the processor to which node n_j is allocated. $v_{children}(n_i)$ denotes all the children nodes of n_i. The other expressions have the same meaning as Algorithm 6.1. The first part of Algorithm 7.1 is

the sibling checking technique which is used to decide whether edge e_{ij} is an identical data that need not be scheduled. According to the results of the first part, the second part schedules edge e_{ij} as a common edge or does it as an identical edge which can be zeroed.

7.3 Using the look-forward algorithm for identical data under the contention model

Algorithm 7.2 (see page 121) shows the look-forward-based algorithm for identical data under the contention model.

7.4 Experimental evaluation

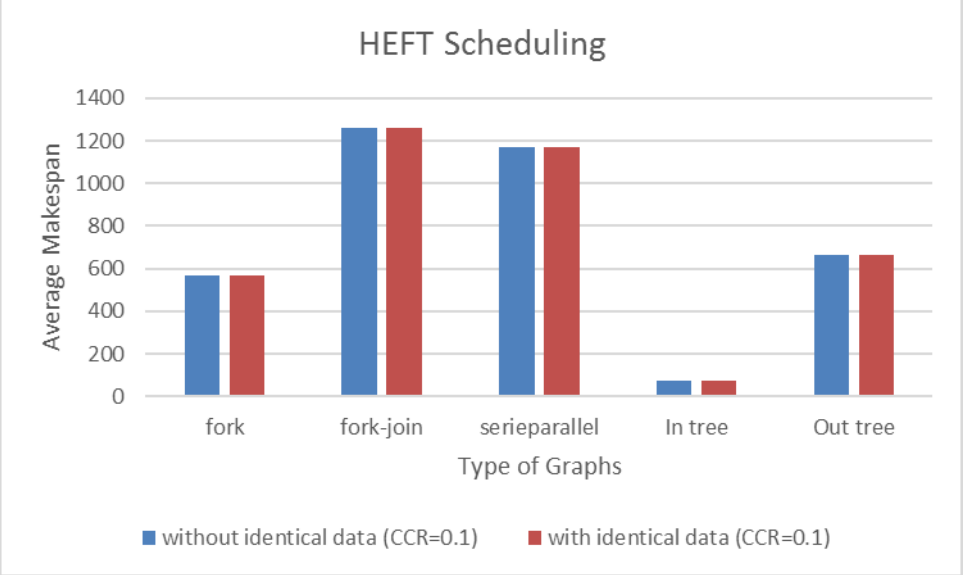

Figure 7.5: HEFT scheduling evaluation (CCR=0.1)

As shown in Figures 7.5, 7.6 and 7.7, there are some observations that can be made for list scheduling under the one-port model. Firstly, when $CCR = 0.1$, no matter whether or not identical data are considered, the results will not differ much. In fact the results are the same, which can be seen in Figure 7.5. That is because considering

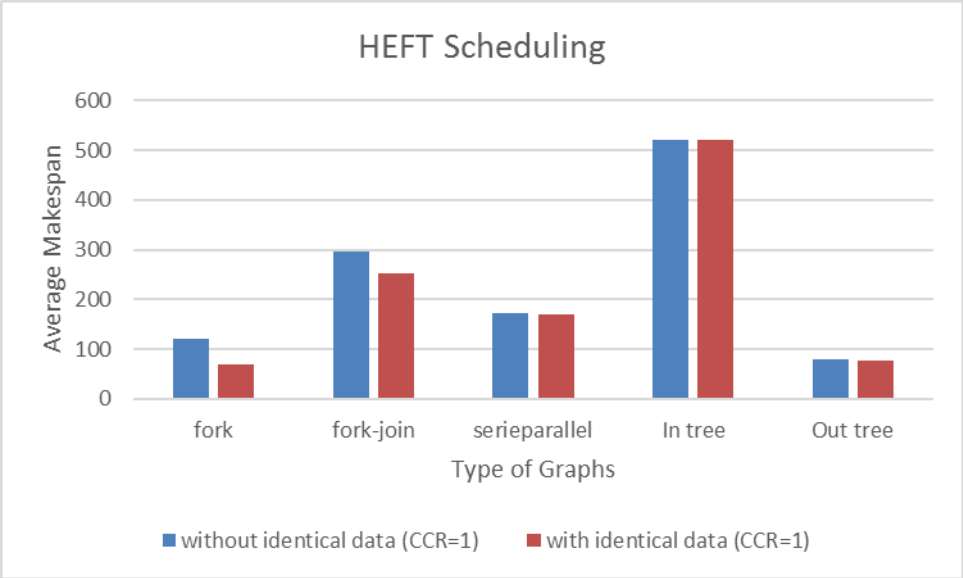

Figure 7.6: HEFT scheduling evaluation (CCR=1)

the identical data reduces the communication effect for the whole schedule, but when $CCR = 0.1$ the communication costs have little effect and the identical data awareness will not help to reduce the schedule length. Secondly, when $CCR = 1.0$ or 10, the communication costs have a greater effect on the schedule length. For the fork graph, the schedule length is reduced when considering identical data (see Figures 7.6 and 7.7). This is because all the edges of the fork graph are edges of identical data, which can be used to reduce the schedule length. For the fork-join graph, when considering identical data, the schedule length is reduced a little; sometimes it is even increased (Figure 7.7 shows when considering the identical data, the schedule length increases in the fork-join graph). The reason for this situation is that list scheduling with identical data can reduce the length of the fork part, but when it comes to the join part, it cannot help. The scheduling of the fork part might make decisions that will increase the inter-processor communication of the join part, which will finally have a negative effect on the overall schedule length. For the series-parallel graph, the result is similar to the fork-join graph (see Figures 7.6 and 7.7). The fork parts can be reduced, but the communication in the other parts reduces the benefit gained due to the identical data in fork parts. For the in-tree, as no node has more than one out-going edge, there is no identical data in the in-tree. In Figures 7.5, 7.6 and 7.7, the results before and after considering the identical

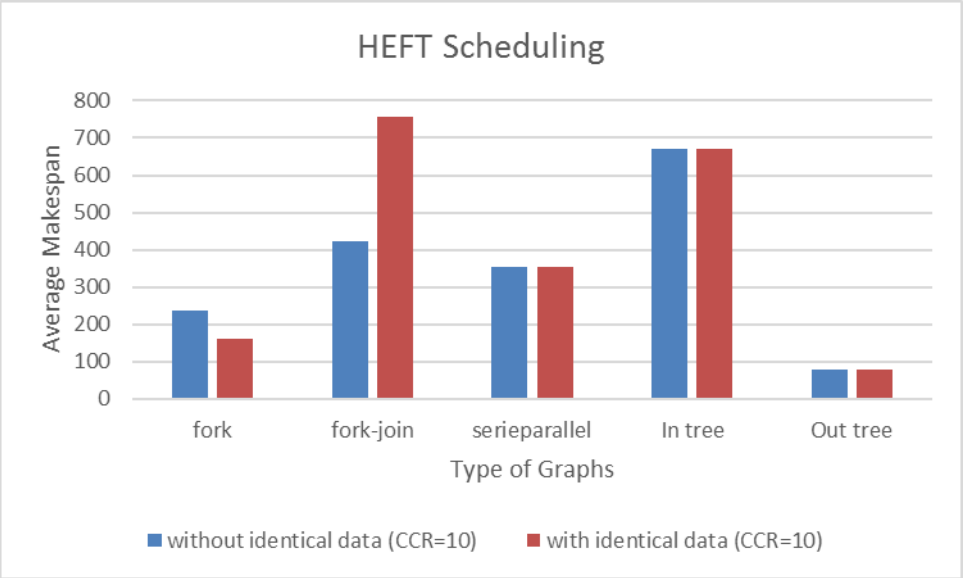

Figure 7.7: HEFT scheduling evaluation (CCR=10)

data are the same. As there are many fork parts in the out-tree, the schedule length is expected to reduce when considering identical data. However, the result shows only a small reduction. The reason is that the max branch (the biggest number of out-going edges of a node) of the out-tree is only 3, hence, considering identical data has little benefit.

Figures 7.8, 7.9 and 7.10 give the results for the look-forward algorithm under the one-port model. Firstly, the lower CCR, that is $CCR = 0.1$, will weaken the benefit associated with the consideration of identical data. Figure 7.8 shows that no matter whether identical data is considered or not, the results are the same. Secondly, for the fork graph, the identical data reduces significantly the schedule length as the CCR increases (see Figures 7.9 and 7.10 fork graph.). Thirdly, for the fork-join graph, as look-forward can 'look' into the join parts and make the decision at the same time as considering the nodes of the fork part, it can reduce the schedule length by considering the identical data. This is a clear advantage of the look-forward algorithm. The bar charts depicted in Figures 7.9 and 7.10 show that the look-forward algorithm can reduce the schedule length of the fork-join graph. Fourthly, as there is no identical data in the in-tree, the result is the same as when considering identical data (see Figures 7.8, 7.9 and 7.10). As for the out-tree, the small max-branch property, as discussed above, makes

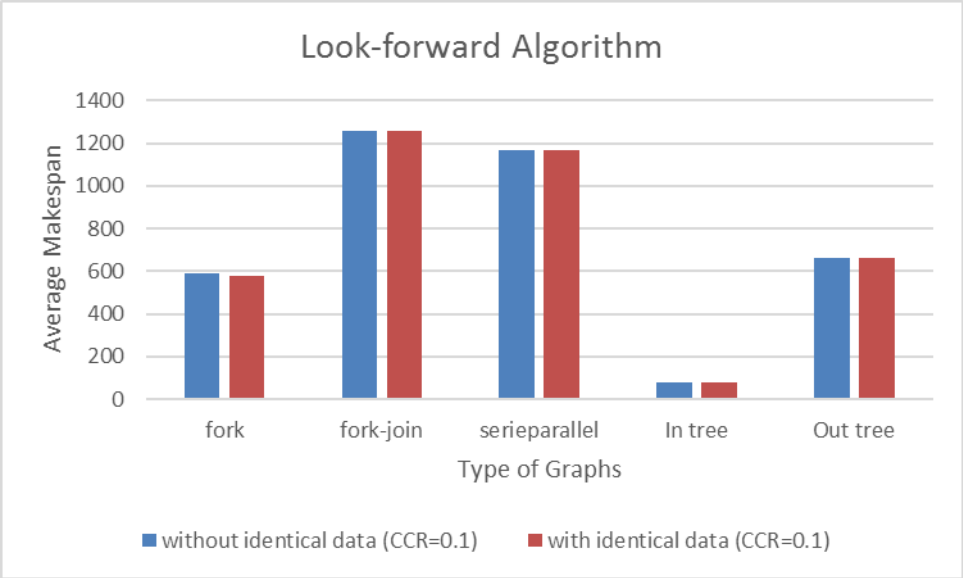

Figure 7.8: Look-forward algorithm(CCR=0.1)

the benefit of identical data disappear (see Figure 7.9 and 7.10).

7.4.1 HEFT scheduling versus look-forward

Comparing the results for HEFT scheduling and look-forward, there are some results that demonstrate the differences between these two algorithms. These can be seen in Figures 7.11, 7.12, 7.13, 7.14, 7.15, 7.16, 7.17, 7.18, 7.19 and 7.20. First, when the CCR is too small, for instance $CCR = 0.1$, both HEFT scheduling and look-forward scheduling will have a similar result and cannot make full use of the identical data. The first part of Figures 7.11 to 7.20 can show this result. Second, Figures 7.11 and 7.12 show that for the fork graph, both HEFT scheduling and look-forward can obtain the benefit of considering identical data and achieve a similar result. For the fork-join (see Figure 7.14), series-parallel (see Figure 7.16) and in-tree (see Figure 7.20), because look-forward can consider the entire graph and HEFT scheduling can only consider the fork part of a graph, look-forward produces a better schedule than list scheduling. For the out-tree graph, with the increase of CCR, HEFT scheduling may provide a better schedule length than look-forward (see the third part of Figures 7.17 and 7.18). That is because out-trees have only fork parts which is beneficial for HEFT scheduling and

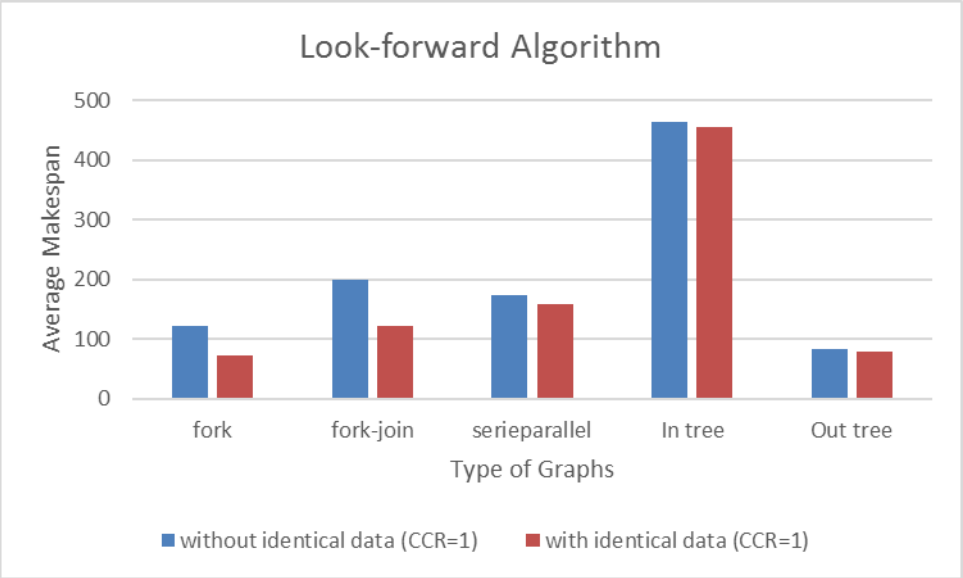

Figure 7.9: Look-forward algorithm(CCR=1)

in the proposed HEFT scheduling algorithm, HEFT scheduling gives a better way to assign the priorities of nodes than look-forward. There are only join parts in the in-tree graph. As look-forward can look forward for the fork parts and HEFT scheduling cannot consider the join parts of a graph, look-forward can exhibit a better performance than HEFT scheduling. The results in Figures 7.19 and 7.20 support the above analysis.

7.5 Summary

To sum up, when the communication costs have enough of an effect on the whole graph, the proposed HEFT scheduling algorithm and look-forward can create good schedules under the one-port/contention model. When considering identical data for a further reduction in schedule length, both HEFT scheduling and look-forward can reduce the schedule length significantly for the fork graph (see Figures 7.7, 7.10, 7.11, 7.12). HEFT scheduling is not efficient at considering identical data in the fork-join graph (see Figures 7.7 and 7.6), but look-forward can overcome this shortcoming and can further reduce the schedule length of fork-join graph after considering identical data (see Figures 7.10 and 7.14). For the series-parallel and in-tree graphs, look-forward also performs better than HEFT scheduling (see Figures 7.16 and 7.20), when considering identical

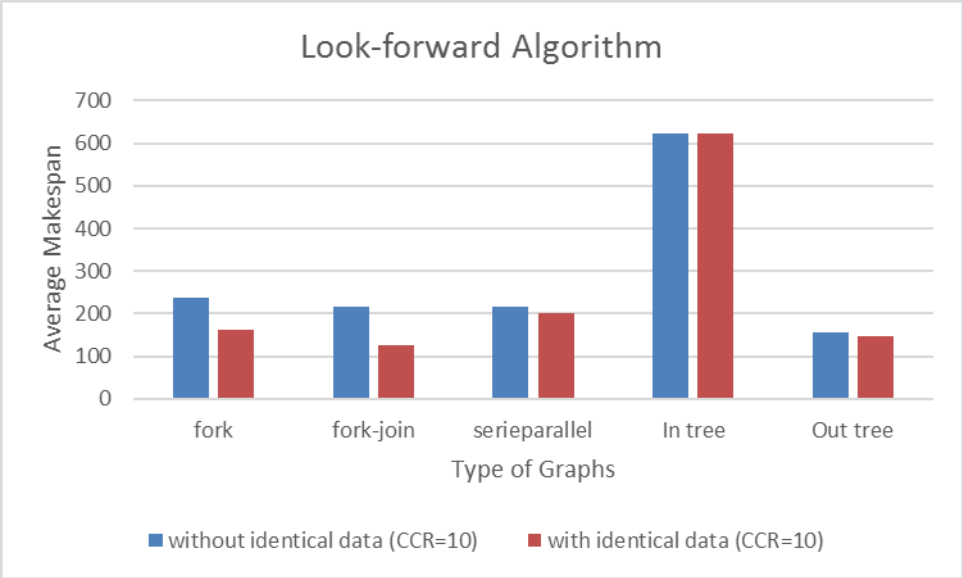

Figure 7.10: Look-forward algorithm(CCR=10)

data. Note that even though look-forward achieves a better result in many graphs than HEFT scheduling, HEFT scheduling has a better priority algorithm than look-forward, which may make it better in a few situations, such as the out-tree graph in Figures 7.18 and 7.17.

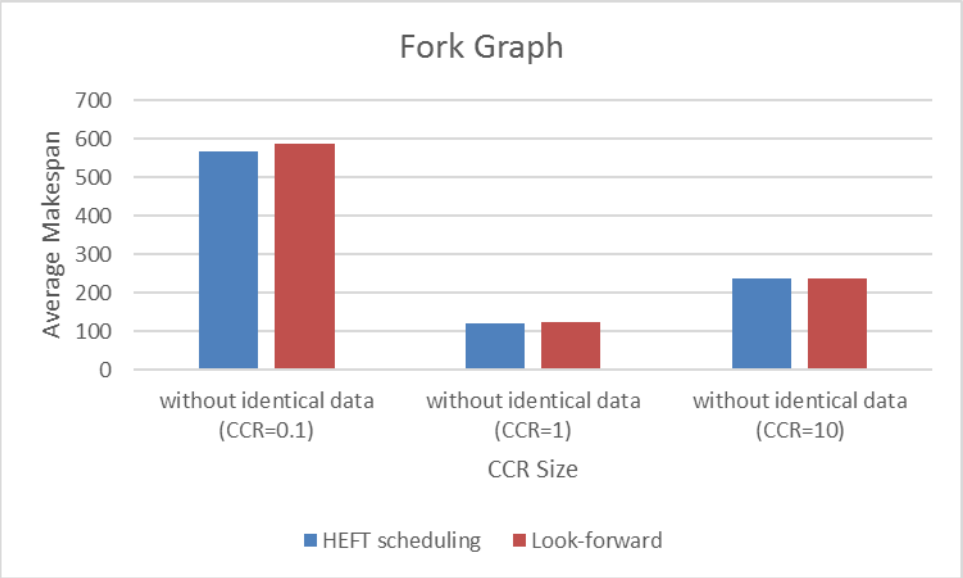

Figure 7.11: Fork without identical data comparison

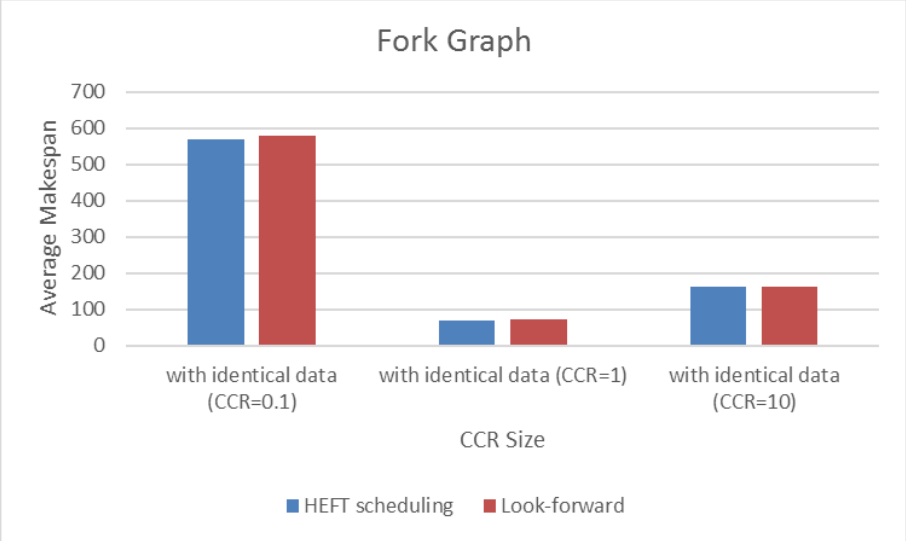

Figure 7.12: Fork with identical data comparison

Algorithm 7.1 Edge scheduling e_{ij} with identical data (one-port model)

Input: processor p_i, processor p_j, edge e_{ij}
Output: finish time of processor p_i's out-port $t_f(p_{(i,out)})$
Output: start time of edge e_{ij} on p_i's out-port $t_s(e_{ij}, p_{(i,out)})$
Output: finish time of edge e_{ij} on p_i's out-port $t_f(e_{ij}, p_{(i,out)})$
Output: finish time of processor p_j's in-port $t_f(p_{(j,in)})$
Output: start time of edge e_{ij} on p_j's in-port $t_s(e_{ij}, p_{(j,out)})$
Output: finish time of edge e_{ij} on p_j's in-port $f_s(e_{ij}, p_{(j,out)})$
$temp \leftarrow 0$
if e_{ij} is identical data **then**
    Get all children nodes $v_{children}(n_i)$of n_i
    for all node $n \in v_{children}(n_i)$ **do**
        if n is scheduled in $proc(n_j)$ **then**
            $temp \leftarrow 1$
        end if
    end for
end if
.....The above part checks whether e_{ij} should be scheduled.....
if $temp \equiv 0$ **then**
    Get finish time $t_f(p_i)$ of p_i
    Get finish time $t_f(p_{(i,out)})$ of p_i's out-port
    $t_s(e_{ij}, p_{(i,out)}) \leftarrow \max(t_f(p_i), t_f(p_{(i,out)}))$
    $t_f(e_{ij}, p_{(i,out)}) \leftarrow t_s(e_{ij}, p_{(i,out)}) + c(e_{ij})$.
    $t_f(p_{(i,out)}) \leftarrow t_f(e_{ij}, p_{(i,out)})$

    Get finish time $t_f(p_j)$ of p_j
    Get finish time $t_f(p_{(j,in)})$ of p_j's in-port
    $t_s(e_{ij}, p_{(j,in)}) \leftarrow \max(t_s(e_{ij}, p_{(i,out)}), t_f(p_{(j,in)}))$
    $t_f(e_{ij}, p_{(j,in)}) \leftarrow t_s(e_{ij}, p_{(j,in)}) + c(e_{ij})$
    $t_f(p_{(j,in)}) \leftarrow t_f(e_{ij}, p_{(j,in)})$
else
    $t_s(e_{ij}) \leftarrow 0$
    $t_f(e_{ij}) \leftarrow 0$
end if

Algorithm 7.2 Look-forward algorithm for the contention model

1: Input: a task graph $G = (V, E, w, c)$
2: Output: a schedule for all tasks $t_i \in V$
3: rank all tasks $t_i \in V$ using the $rank_u$
4: $L \leftarrow NULL$
5: **while** there are unscheduled tasks **do**
6: $t \leftarrow$ unscheduled task with highest $rank_u$
7: **if** $t \notin L$ and children of $t \notin L$ **then**
8: $L \leftarrow$ children of t
9: **end if**
10: **if** $t \in L$ **then**
11: $L \leftarrow NULL$
12: $L \leftarrow$ children of t
13: **end if**
14: status 1
15: **for all** resource $r_i \in$ the resources set R **do**
16: schedule t on r_i using Algorithm 7.1 and Algorithm 6.4
17: schedule all tasks $\in L$ using Algorithm 7.1 and Algorithm 6.4
18: $EFT_{r_i} \leftarrow$ maximum EFT for tasks $\in L$
19: return to status 1 (recover the ports)
20: **end for**
21: schedule t on r_i such that $EFT_{r_i} \leq EFT_{r_k}$ using Algorithm 7.1 and Algorithm 6.4
22: **end while**

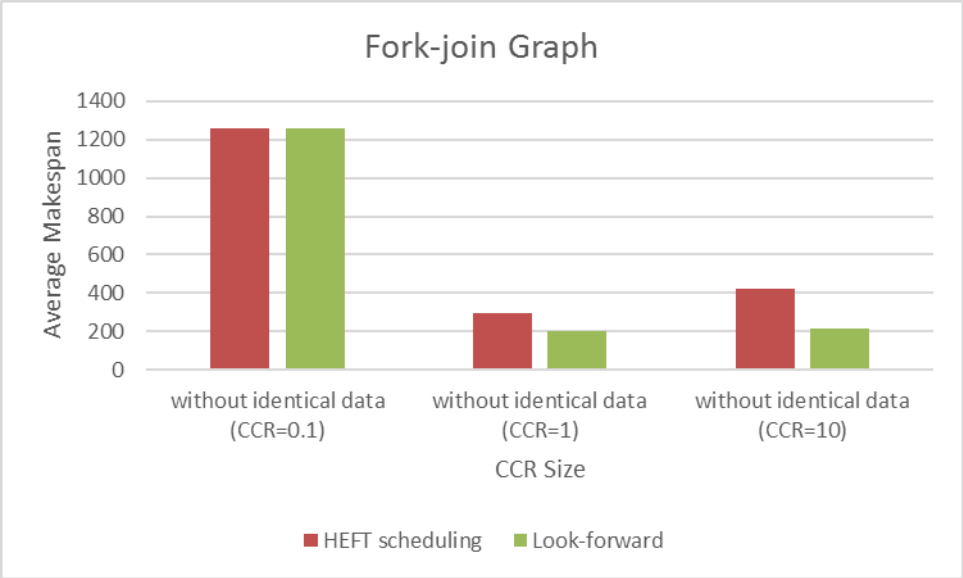

Figure 7.13: Fork-join without identical data comparison

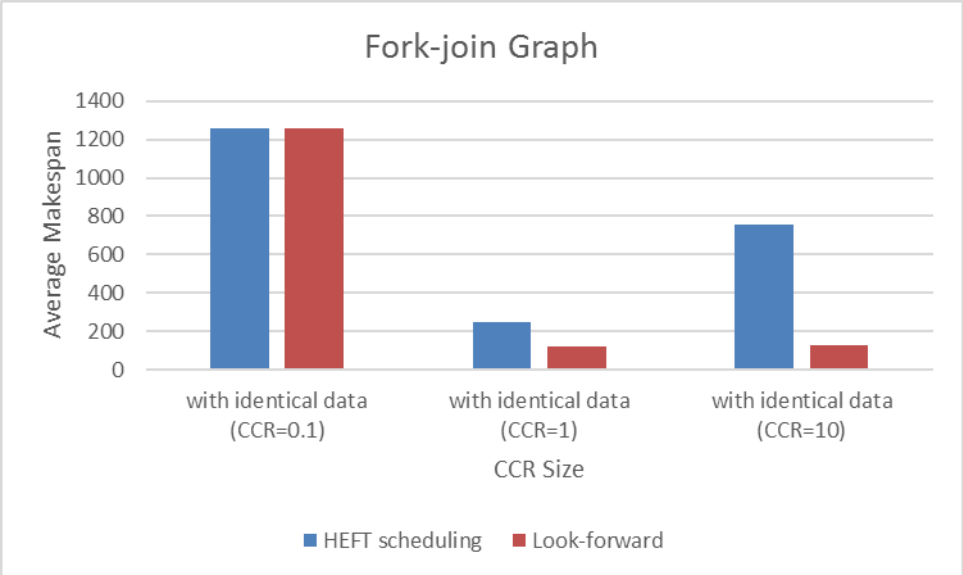

Figure 7.14: Fork-join with identical data comparison

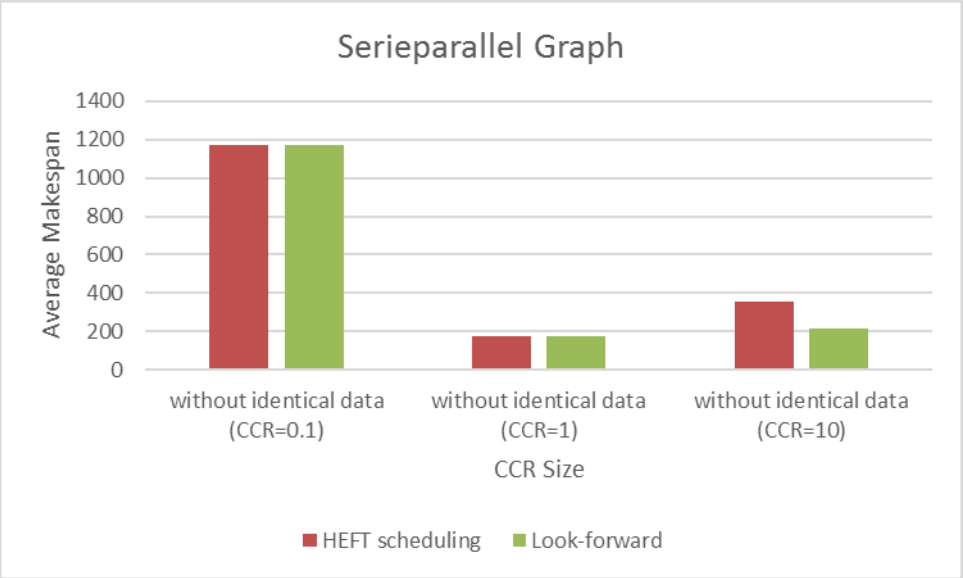

Figure 7.15: SerieParallel without identical data comparison

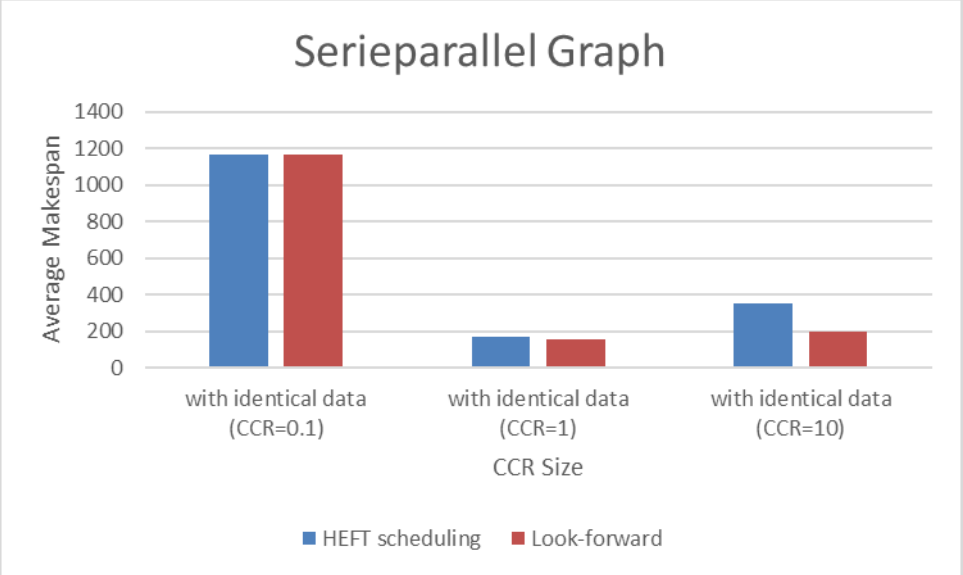

Figure 7.16: SerieParallel with identical data comparison

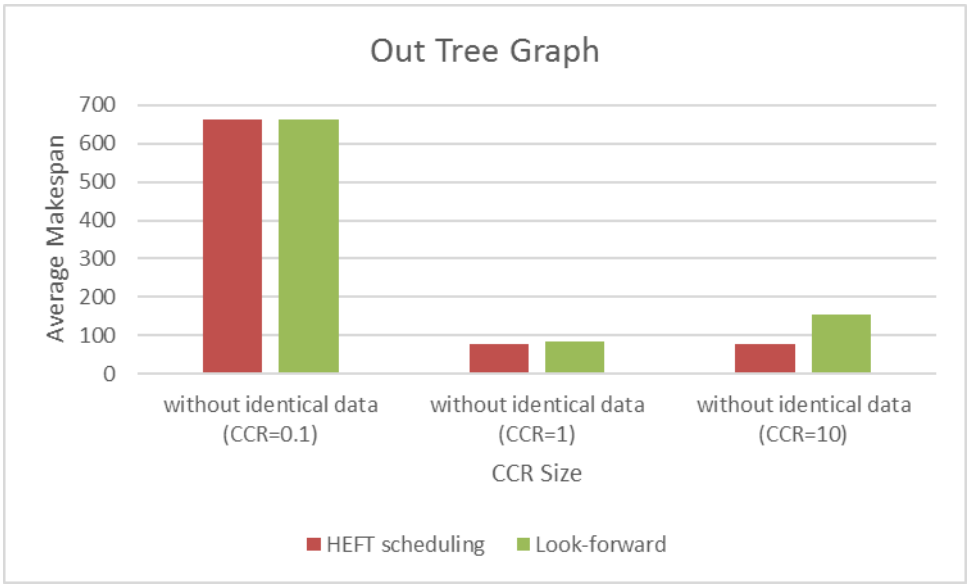

Figure 7.17: Out Tree without identical data comparison

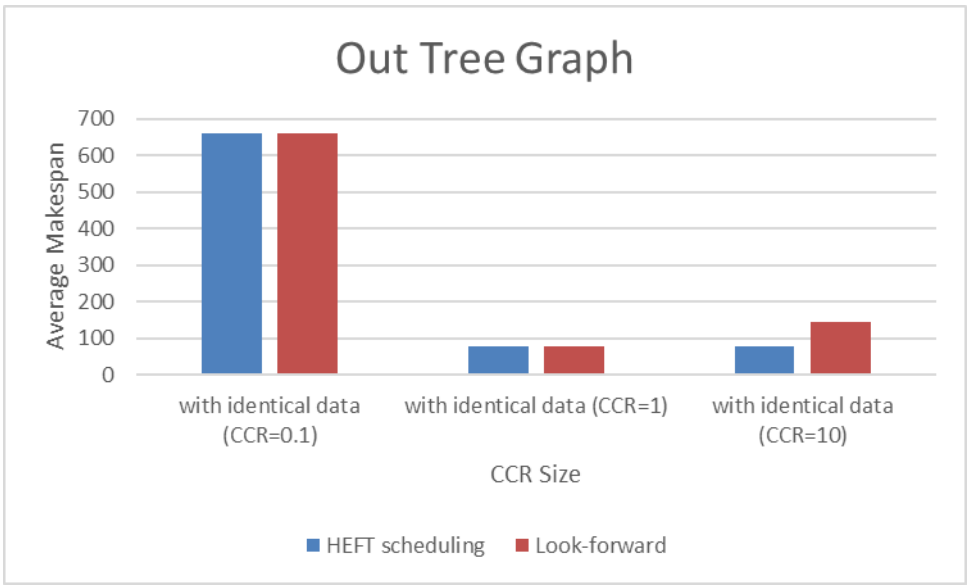

Figure 7.18: Out Tree with identical data comparison

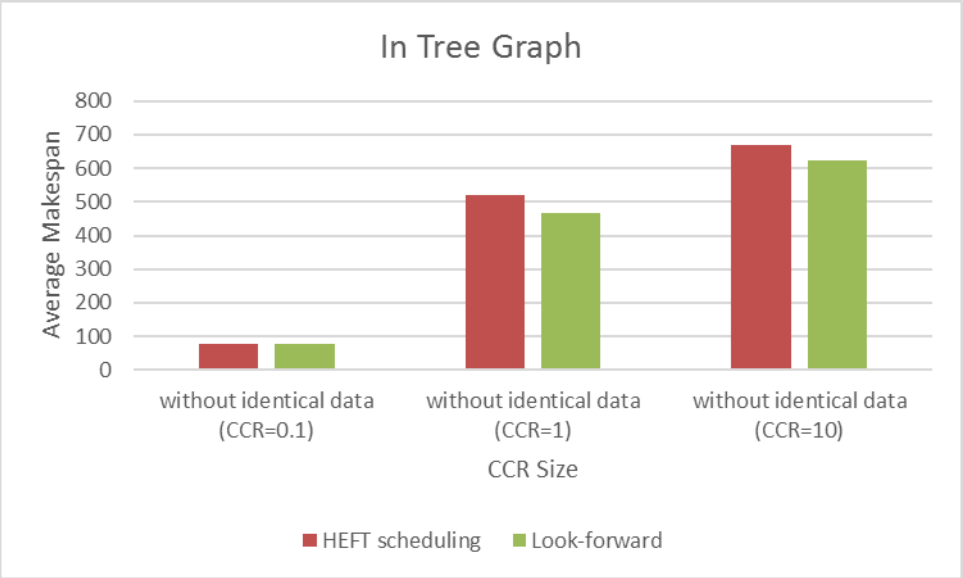

Figure 7.19: In Tree without identical data comparison

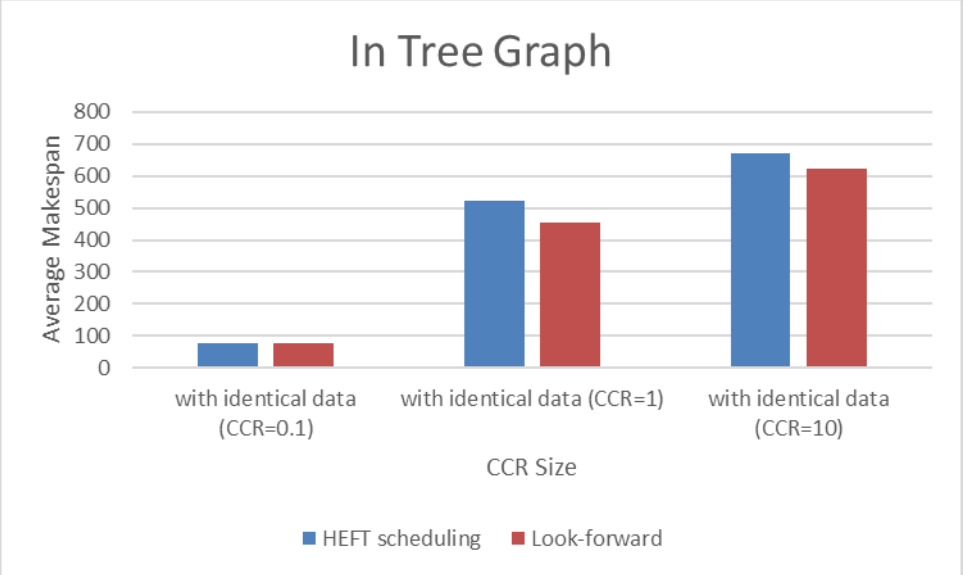

Figure 7.20: In Tree with identical data comparison

Chapter 8

Conclusion

This thesis studied DAG scheduling with respect to the classic model and the one-port/contention model. As task scheduling is an NP-hard problem, many scheduling algorithms have been proposed to solve this problem under the classic model, but very few of them function effectively under the one-port/contention model. In order to find suitable algorithms to solve the scheduling problem in both the classic and contention models, this thesis studied the two most important fundamental heuristics of task scheduling: list scheduling and clustering. On the basis of list scheduling and clustering in the classic model, this thesis suggested a new algorithm, called look-forward algorithm, to perform DAG scheduling. The experimental results show that the look-forward algorithm reduces the schedule length with an acceptable run time.

This thesis has extended the proposed look-forward algorithm into the contention model area. Different from the classic model, the contention model needs to consider communication contention. This consideration places additional constraints on the scheduling algorithm. Not only does the algorithm need to get better at scheduling, but the communication contention also needs to be fully resolved. Look-forward uses a novel method to allocate the tasks. It allocates them by the earliest finish times of their children tasks and some selected lower-priority tasks, but not by the mainstream approach of the earliest finish time of the task itself. This unique method not only helps look-forward perform better in the classical model but also allows it to fully consider the incoming and outgoing communication and structure of the DAG, which helps it perform well in the contention model.

After consideration of the contention model, this thesis focused on the communication data. An identical data problem is suggested and studied. Considering identical data can be used to reduce the scheduling length, especially in the contention model, but it is not so easy to combine this with other algorithms. After examining the experimental results, we believe the identical data can be considered by the look-forward algorithm, because look-forward can consider both the incoming data and the outgoing data, which will provide a good balance between identical data and the out-port contention. This thesis focused on how to develop good scheduling algorithms in the one-port model and how to consider identical data in this model.

Based on the above study, there is some future work that should be considered. For example, additional experiments should be undertaken to test the proposed algorithms. Also, additional graphs should be used to test identical data. When considering identical data, the algorithms assume all fork parts have identical data. This assumption could be relaxed. Future work could analyse a variety of graphs, for example, graphs where 30%, 50%, 70% and 100% of the fork parts deal with identical data.

Bibliography

[ACKZ05] M. Andrews, J. Chuzhoy, S. Khanna, and L. Zhang. Hardness of the undirected edge-disjoint paths problem with congestion. In *Foundations of Computer Science. FOCS 2005. 46th Annual IEEE Symposium on*, pages 226–241, 2005.

[AFJ+13] P.-A. Arras, D. Fuin, E. Jeannot, A. Stoutchinin, and S. Thibault. List scheduling in embedded systems under memory constraints. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2013 25th International Symposium on*, pages 152–159, 2013.

[AHK98] R. Armstrong, D. Hensgen, and T. Kidd. The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. In *Proceedings Seventh Heterogeneous Computing Workshop (HCW'98)*, pages 79–87, March 1998.

[AIAS87] George B Adams III, Dharma P Agrawal, and Howard Jay Siegel. A survey and comparison of fault-tolerant multistage interconnection networks. *IEEE Computer*, 20(6), 1987.

[AK98] Ishfaq Ahmad and Yu-Kwong Kwok. On exploiting task duplication in parallel program scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 9(9):872–892, 1998.

[ANF12] Oliver Arnold, Benedikt Noethen, and Gerhard Fettweis. Instruction set architecture extensions for a dynamic task scheduling unit. In *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*, pages 249–254. IEEE, 2012.

[ASA16] S. A. Arshad, H. F. Sheikh, and I. Ahmad. A comparison of evolutionary techniques for task-to-core scheduling algorithms with performance, energy, and temperature optimization. In *Proceedings Seventh International Green and Sustainable Computing Conference (IGSC)*, pages 1–8, November 2016.

[ASK+13] I. A. Abed, K. S. M. Sahari, S. P. Koh, S. K. Tiong, and P. Jagadeesh. Using electromagnetism-like algorithm and genetic algorithm to optimize time of task scheduling for dual manipulators. In *Proceedings IEEE Region 10 Humanitarian Technology Conf*, pages 182–187, August 2013.

[BCD+08] S. Bharathi, A. Chervenak, E. Deelman, G. Mehta, M. Su, and K. Vahi. Characterization of scientific workflows. In *Proceedings Third Workshop Workflows in Support of Large-Scale Science*, pages 1–10, 2008.

[BD04] Samuel I Brandt and Jan DeHaan. System and user interface for processing task schedule information, March 30 2004. US Patent 6,714,913.

[BHR09] Anne Benoit, Mourad Hakem, and Yves Robert. Contention awareness and fault-tolerant scheduling for precedence constrained tasks in heterogeneous systems. *Parallel Computing*, 35(2):83–108, 2009.

[BMRR06] Olivier Beaumont, Loris Marchal, Veronika Rehn, and Yves Robert. Fifo scheduling of divisible loads with return messages under the one-port model. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 14–pp. IEEE, 2006.

[BSB+01] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, and Richard F Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001.

[BSM10] Luiz F Bittencourt, Rizos Sakellariou, and Edmundo RM Madeira. DAG scheduling using a lookahead variant of the heterogeneous earliest finish

time algorithm. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 27–34. IEEE, 2010.

[CB76] Edward Grady Coffman and John L Bruno. *Computer and job-shop scheduling theory*. John Wiley & Sons, 1976.

[CC91] Jean-Yves Colin and Philippe Chrétienne. CPM scheduling with small communication delays and task duplication. *Operations Research*, 39(4):680–684, 1991.

[CCK12] Pravanjan Choudhury, PP Chakrabarti, and Rajeev Kumar. Online scheduling of dynamic task graphs with communication and contention for multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 23(1):126–133, 2012.

[Che13] *WorkflowSim: A Toolkit for Simulating Scientific Workflows in Distributed Environments*, 2013.

[CHJC14] Chuan-Feng Chiu, Steen J Hsu, Sen-Ren Jan, and Jyun-An Chen. Task scheduling based on load approximation in cloud computing environment. In *Future Information Technology*, pages 803–808. Springer, 2014.

[CJ01] Bertrand Cirou and Emmanuel Jeannot. Triplet: a clustering scheduling algorithm for heterogeneous systems. In *Parallel Processing Workshops, 2001. International Conference on*, pages 231–236. IEEE, 2001.

[CR92] Yeh-Ching Chung and Sanjay Ranka. Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors. In *Supercomputing'92. Proceedings*, pages 512–521. IEEE, 1992.

[CS87] Ming-Syan Chen and Kang G. Shin. Processor allocation in an n-cube multiprocessor using gray codes. *Computers, IEEE Transactions on*, 100(12):1396–1407, 1987.

[CTS97] Chantana Chantrapornchai, Sissades Tongsima, and EH-M Sha. Imprecise task schedule optimization. In *Fuzzy Systems, 1997., Proceedings of the Sixth IEEE International Conference on*, volume 3, pages 1265–1270. IEEE, 1997.

[DA97] Sekhar Darbha and Dharma P Agrawal. A task duplication based scalable scheduling algorithm for distributed memory systems. *Journal of Parallel and Distributed Computing*, 46(1):15–27, 1997.

[EKP+13] Daji Ergu, Gang Kou, Yi Peng, Yong Shi, and Yu Shi. The analytic hierarchy process: task scheduling and resource allocation in cloud computing environment. *The Journal of Supercomputing*, 64(3):835–848, 2013.

[ERLA94] Hesham El-Rewini, Theodore G Lewis, and Hesham H Ali. *Task scheduling in parallel and distributed systems*. Prentice-Hall, Inc., 1994.

[Ert98] Tuna Ertemalp. Method and apparatus for arranging and displaying task schedule information in a calendar view format, April 28 1998. US Patent 5,745,110.

[FAdM+12] S.J. Filho, A. Aguiar, F.G. de Magalhaes, O. Longhi, and F. Hessel. Task model suitable for dynamic load balancing of real-time applications in NoC-based MPSoCs. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 49–54, 2012.

[FGA+98] R. F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J. D. Lima, F. Mirabile, L. Moore, B. Rust, and H. J. Siegel. Scheduling resources in multi-user, heterogeneous, computing environments with smartnet. In *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW'98)*, pages 184–199, March 1998.

[GDP08] Shashidhar Gandham, Milind Dawande, and Ravi Prakash. Link scheduling in wireless sensor networks: distributed edge-coloring revisited. *Journal of Parallel and Distributed Computing*, 68(8):1122–1134, 2008.

[Gra99] Martin Grajcar. Genetic list scheduling algorithm for scheduling and allocation on a loosely coupled heterogeneous multiprocessor system. In *Design Automation Conference, 1999. Proceedings. 36th*, pages 280–285. IEEE, 1999.

[GY92a] Apostolos Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, 1992.

[GY92b] Apostolos Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, 1992.

[GY93] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *Parallel and Distributed Systems, IEEE Transactions on*, 4(6):686–701, 1993.

[Han94] Simon Handley. On the use of a directed acyclic graph to represent a population of computer programs. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence. Proceedings of the First IEEE Conference on*, pages 154–159. IEEE, 1994.

[IK77] Oscar H. Ibarra and Chul E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of ACM*, 24(2):280–289, April 1977.

[IÖF95] Michael A Iverson, Füsun Özgüner, and Gregory J Follen. Parallelizing existing applications in a distributed heterogeneous environment. In *4th Heterogeneous Computeing Workshop(HCW'95*. Citeseer, 1995.

[JFKG13] J. Jackson, M. Faied, P. Kabamba, and A. Girard. Distributed constrained minimum-time schedules in networks of arbitrary topology. 29(2):554–563, 2013.

[JSM91] Kevin Jeffay, Donald F Stanat, and Charles U Martel. On non-preemptive scheduling of period and sporadic tasks. In *[1991] Proceedings Twelfth Real-Time Systems Symposium*, pages 129–139. IEEE, 1991.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[KA11] T Kokilavani and Dr DI George Amalarethinam. Load balanced min-min algorithm for static meta-task scheduling in grid computing. *International Journal of Computer Applications*, 20(2):43–49, 2011.

[KDR13] R. Khogali, O. Das, and K. Raahemifar. Mobile parallel computing algorithms for single-buffered, speed-scalable processors. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2013 12th IEEE International Conference on*, pages 1832–1839, 2013.

[KN84] Hironori Kasahara and Seinosuke Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, 33(11):1023–1029, 1984.

[LCC+13] Jinho Lee, Moo-Kyoung Chung, Yeon-Gon Cho, Soojung Ryu, Jung Ho Ahn, and Kiyoung Choi. Mapping and scheduling of tasks and communications on many-core SoC under local memory constraint. *IEEE Transactions on Computers*, 32(11):1748–1761, 2013.

[LLLW06] Chun-Hsien Liu, Chia-Feng Li, Kuan-Chou Lai, and Chao-Chin Wu. A dynamic critical path duplication task scheduling algorithm for distributed heterogeneous computing systems. In *Proceedings 12th International Conference Parallel and Distributed Systems - (ICPADS'06)*, volume 1, pages 8 pp.–, 2006.

[LMXZ14] X. Li, Y. Mao, X. Xiao, and Y. Zhuang. An improved max-min task-scheduling algorithm for elastic cloud. In *Proceedings of the Consumer and Control 2014 International Symposium*, pages 340–343, June 2014.

[MVZ93] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 11(2):146–178, 1993.

[MXL14] Y. Mao, C. Xi, and X. Li. Max-min task scheduling algorithm for load balance in cloud computing. In *Proceedings of International Conference on Computer Science and Information Technology*, 2014.

[PC01] Chan-Ik Park and Tae-Young Choe. An optimal scheduling algorithm based on task duplication. In *Parallel and Distributed Systems, 2001. ICPADS 2001. Proceedings. Eighth International Conference on*, pages 9–14. IEEE, 2001.

[PLW96] Michael A. Palis, Jing-Chiou Liou, and David S. L. Wei. Task clustering and scheduling for distributed memory parallel architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 7(1):46–55, 1996.

[RA00] Samantha Ranaweera and Dharma P Agrawal. A task duplication based scheduling algorithm for heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 445–450. IEEE, 2000.

[RSS90] Krithi Ramamritham, John A. Stankovic, and P-F Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *Parallel and Distributed Systems, IEEE Transactions on*, 1(2):184–194, 1990.

[RVB07] M. Rahman, S. Venugopal, and R. Buyya. A dynamic critical path algorithm for scheduling scientific workflow applications on global grids. In *e-Science and Grid Computing, IEEE International Conference on*, pages 35–42, 2007.

[RvG99] Andrei Rădulescu and Arjan JC van Gemund. On the complexity of list scheduling algorithms for distributed-memory systems. In *Proceedings of the 13th International Conference on Supercomputing*, pages 68–75. ACM, 1999.

[Sin07] Oliver Sinnen. *Task scheduling for parallel systems*, volume 60. Wiley, 2007.

[SKH95] Behrooz A Shirazi, Krishna M Kavi, and Ali R Hurson. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Society Press, 1995.

[SKS07] Oliver Sinnen, Alexander Vladimirovich Kozlov, and Ahmed Zaki Semar Shahul. Optimal scheduling of task graphs on parallel systems. *Parallel and Distributed Computing and Networks*, 551, 2007.

[SL93] Gilbert C Sih and Edward Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 4(2):175–187, 1993.

[Sny88] Lawrence Snyder. A taxonomy of synchronous parallel machines. Technical report, DTIC Document, 1988.

[SS01] Oliver Sinnen and Leonel Sousa. Comparison of contention aware list scheduling heuristics for cluster computing. In *Parallel Processing Workshops, 2001. International Conference on*, pages 382–387. IEEE, 2001.

[SS04] Oliver Sinnen and Leonel Sousa. List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. *Parallel Computing*, 30(1):81–101, 2004.

[SS05] Oliver Sinnen and Leonel A Sousa. Communication contention in task scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 16(6):503–515, 2005.

[SS10] S. Selvarani and G. S. Sadhasivam. Improved cost-based algorithm for task scheduling in cloud computing. In *Proceedings IEEE Int. Conf. Computational Intelligence and Computing Research*, pages 1–5, December 2010.

[SSS06] Oliver Sinnen, Leonel Augusto Sousa, and Frode Eika Sandnes. Toward a realistic task scheduling model. *Parallel and Distributed Systems, IEEE Transactions on*, 17(3):263–275, 2006.

[STK09] Oliver Sinnen, Andrea To, and Manpreet Kaur. Contention-aware scheduling with task duplication. In *Job Scheduling Strategies for Parallel Processing*, pages 157–168. Springer, 2009.

[STK11] Oliver Sinnen, Andrea To, and Manpreet Kaur. Contention-aware scheduling with task duplication. *Journal of Parallel and Distributed Computing*, 71(1):77–86, 2011.

[SZ04a] Rizos Sakellariou and Henan Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 111. IEEE, 2004.

[SZ04b] Rizos Sakellariou and Henan Zhao. A low-cost rescheduling policy for efficient mapping of workflows on grid systems. *Scientific Programming*, 12(4):253–262, 2004.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[TBW92] Ken W Tindell, Alan Burns, and Andy J. Wellings. Allocating hard real-time tasks: an NP-hard problem made easy. *Real-Time Systems*, 4(2):145–165, 1992.

[THW02] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[TS84] Tuomenoksa and Siegel. Task preloading schemes for reconfigurable parallel processing systems. *IEEE Transactions on Computers*, C-33(10):895–905, 1984.

[Ull75] Jeffrey D. Ullman. NP-complete scheduling problems. *Journal of Computer and System sciences*, 10(3):384–393, 1975.

[VR01] Paulo Verissimo and Luis Rodrigues. *Distributed systems for systems architects*, volume 1. Springer, 2001.

[WSF89] L Darrell Whitley, Timothy Starkweather, and D'Ann Fuquay. Scheduling problems and traveling salesmen: The genetic edge recombination operator. In *ICGA*, volume 89, pages 133–40, 1989.

[XLL11] Jin Xu, Albert Lam, and Victor OK Li. Chemical reaction optimization for task scheduling in grid computing. *Parallel and Distributed Systems, IEEE Transactions on*, 22(10):1624–1631, 2011.

[XW01] Yuan Xie and Wayne Wolf. Allocation and scheduling of conditional task graph in hardware/software co-synthesis. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 620–625. IEEE, 2001.

[YG93] Tao Yang and Apostolos Gerasoulis. List scheduling with and without communication delays. *Parallel Computing*, 19(12):1321–1344, 1993.

[YG94] Tao Yang and Apostolos Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *Parallel and Distributed Systems, IEEE Transactions on*, 5(9):951–967, 1994.

[YXP+11] Jiadong Yang, Hua Xu, Li Pan, Peifa Jia, Fei Long, and Ming Jie. Task scheduling using bayesian optimization algorithm for heterogeneous computing environments. *Applied Soft Computing*, 11(4):3297–3310, 2011.

[ZB97] Avi Ziv and Jehoshua Bruck. Performance optimization of checkpointing schemes with task duplication. *Computers, IEEE Transactions on*, 46(12):1381–1386, 1997.

[ZS03] Henan Zhao and Rizos Sakellariou. An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. In *Euro-Par 2003 Parallel Processing*, pages 189–194. Springer, 2003.

[ZS06] Henan Zhao and Rizos Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 14–pp. IEEE, 2006.

[ZTS15] Wei Zheng, Lu Tang, and Rizos Sakellariou. A Priority-Based Scheduling Heuristic to Maximize Parallelism of Ready Tasks for DAG Applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 596–605. IEEE, 2015.