Workload Partitioning and Scheduling on Heterogeneous Multi-Core Systems

Yen-Ting Chen

A Thesis Submitted to
Institute of Computer Science and Information Engineering
College of Engineering
National Chung Cheng University
for the Degree of Master
in
Computer Science and Information Engineering

June 2013
Abstract

Due to the diversity in computing capabilities of processors in heterogeneous multicore systems, it is difficult to come up with a perfect task scheduling algorithm that can avoid all processors from becoming idle at some point in time during the whole schedule. The situation becomes worse when the capabilities of processors differ by a large margin or the ratio of communication time between tasks to the computation time of tasks is very large. Nevertheless, it is this imperfection that motivates this Thesis to propose a re-scheduling scheme that leverages the characteristics of divisible tasks by partitioning the workload across two different processors so as to fill the holes (idle time slots) in the schedule. Based on the type of hole, constant or varying, different strategies are proposed, including a profiling-based partitioning and an on-the-fly partitioning. Re-scheduling based on a combination of these two strategies results in a decrease in the makespan and total amount of idle time of processors. Experiment results show that the makespan can be decreased by 14% and the total amount of idle time by 50%.

Keywords: Task Scheduling, Workload Partitioning, Heterogeneous Multi-core Systems, Schedule Hole
# Contents

1 Introduction 1

1.1 Background .......................................................... 2

1.2 Motivation .............................................................. 5

1.3 Thesis Organization ...................................................... 8

2 Related Work 9

2.1 Heterogeneous Task Scheduling Algorithm .......................... 10

2.1.1 Heuristic Scheduling Algorithms .................................. 10

2.1.2 Enhanced Scheduling Algorithms .................................. 13

2.2 Workload Partitioning Method ....................................... 14

3 Preliminaries 18

3.1 System Model .......................................................... 18

3.1.1 Architecture model ............................................... 18

3.1.2 Application Model ............................................... 19
3.1.3 Schedule Model ............................................. 22
3.2 Task Partition Profile Database ............................... 25
3.3 Metrics .......................................................... 26

4 Workload Partitioning and Re-Scheduling on HMCS 27
4.1 Workload Partitioning and Re-scheduling ..................... 27
   4.1.1 Algorithm of the Refine Method ......................... 30
4.2 Workload Partitioning Method ................................ 35
   4.2.1 Profiling-based Partitioning ............................ 35
   4.2.2 On-the-fly Partitioning ................................ 36
4.3 Example of Workload Partitioning and Scheduling ............ 38

5 Experiment Results ............................................. 44
5.1 Experiment Setup ............................................... 44
   5.1.1 Architecture Setup .................................... 44
   5.1.2 Simulation Framework .................................. 45
5.2 Performance Evaluation ....................................... 47
   5.2.1 General Cases ......................................... 47
   5.2.2 Adopted Algorithms .................................. 47
   5.2.3 Performance ........................................... 48
5.3 Comparison .............................................. 53

5.3.1 Comparative Results ................................. 53

5.4 Communication to Computation Ratio (CCR) ....... 56

5.4.1 CCR Results ........................................... 56

6 Conclusions ............................................. 58
List of Figures

1.1 Task graph of example ................................................. 6
1.2 (a)HEFT Scheduling result of example (b)Scheduling result of example after the partition finished ............................................. 7
3.1 Task graph model ........................................................ 20
3.2 Two types of the hole while scheduling (a)Constant hole (b)Variable hole ................................................................. 24
3.3 Two types of the hole ...................................................... 25
3.4 Task partition principle ................................................... 25
4.1 Re-scheduling algorithm .................................................. 29
4.2 Task graph of example ................................................... 38
4.3 (a)HEFT Scheduling result of example (b)Scheduling result of example after first partitioning ................................................. 39
4.4 (a)Scheduling result of example after second partition (b)Scheduling result of example after third partition ................................................. 40
4.5 Scheduling result of example after fourth partition . . . . . . . . . . 43

5.1 Simulation framework . . . . . . . . . . . . . . . . . . . . . . . . . 45
5.2 Makespan of scheduling methods . . . . . . . . . . . . . . . . . . . 48
5.3 Turnaround time of scheduling methods . . . . . . . . . . . . . . . 49
5.4 Total amount of hole of scheduling methods . . . . . . . . . . . . . 50
5.5 Number of hole of scheduling methods . . . . . . . . . . . . . . . . 50
5.6 Runtime of scheduling methods . . . . . . . . . . . . . . . . . . . . 51
5.7 Percentage of partitioned tasks in a task graph . . . . . . . . . . . . 52
5.8 Distribution of partitioned tasks on task graphs . . . . . . . . . . . . 52
5.9 Makespan comparison of CCR . . . . . . . . . . . . . . . . . . . . . . 57
List of Tables

1.1 Computation time for the example .................................. 5

4.1 Computation time for the example .................................. 38

5.1 Comparison of re-schedule methods ................................. 55
Chapter 1

Introduction

Computers have been used worldwide. We use it for entertainment such as watching videos, playing games. On the other hand we use it for work such as science calculation, testing program efficiency, etc. People may want to do a lot of things at the same time, and this challenges the hardware capability of our computer. How to execute the instructions in parallel efficiently is an important issue. When there are too many tasks to do and when we cannot execute all of them simultaneously, we have to determine the order of the tasks. It is also called task scheduling. In addition to task scheduling, workload partitioning is also a method to improve the efficiency of the system. If there is a task that has a large amount data and computations, we can use workload partitioning to share its workload to more than one device. Workload partitioning can reduce waiting time of other tasks and improve the utilization of the system because tasks can be partitioned to fill the idle time. Due to the workload sharing, workload partitioning also makes the system load across different processors more balanced.
Formerly, a system had a single processing unit such as CPU. Gradually, not only the number of cores kept increasing in a system, but multiple types of cores were also designed for different programs such as CPU, GPU, and DSP. When there is a single type of core, the system is called a homogeneous multi-core system and when there are multiple types of cores, then the system is called heterogeneous multi-core systems (HMCS). Example of such systems include AMD Accelerated Processing Units (AMD Fusion) [1], Nvidia Tegra 4 [2]. To fully utilize the heterogeneous computing resources, different kinds of tasks should be allocated to the processors according to their characteristics. This is called task scheduling on HMCS.

Research on both workload partitioning and task scheduling have been conducted for many years. These two areas of research have been largely independent, especially for HMCS. However, if these two areas can be integrated, then we anticipate that the target system resources can be better utilized, with significant enhancement of performance.

1.1 Background

The target platform in this work is an HMCS, equipped with multiple CPU cores and GPU cores. In HMCS, we can use compute unified device architecture (CUDA) [3] for GPU programming in NVIDIA card. As to CPU, we can use Intel’s Threading Building Blocks (TBB) [4] to improve the performance of a parallel program. For an application running on both CPU and GPU, we use either both CUDA and TBB or we can use the Open Computing Language (OpenCL) [5], which is a
framework that could program on CPU and GPU directly.

There are two kinds of task scheduling. One is real-time task scheduling, in which every task should be finished before a given deadline. In real-time systems, we make sure that tasks do not miss their deadlines. If the tasks are non-real-time, then the goal is to finish all tasks as soon as possible. In other words, we minimize the makespan of the tasks.

Tasks also have their own characteristics. A task is defined as preemptive or non-preemptive. Further, task is classified into divisible or indivisible by its computational characteristics. There may or may not be dependency between tasks. Task scheduling can research on the above different kinds of conditions.

This Thesis does not consider real-time task scheduling. We adopt non-real-time task scheduling and use makespan as the metric of the task schedule. To obtain the smallest makespan, our schedule method must consider several aspects of the tasks. In homogeneous multi-core systems, availability, locality and critical path can be taken into consideration while scheduling. Availability of the device is used to maintain workload balance of the system. If availability is not taken into consideration, there may be some devices that are idle while other devices are busy all the time. This situation indicates that resources are being wasted. Critical path is a sequence of tasks that takes the largest amount of time. Scheduling the critical path first usually results in the biggest improvement in the system performance. Locality of the predecessor task is normally related to the communication cost. The communication time required by a task from a device to another device may be different for
different pairs of devices and for different tasks. If the tasks are independent, there is no resource sharing among tasks, that is, there is no communication time between independent tasks. In contrast, if tasks have dependencies, locality must be taken into consideration. Otherwise, the devices may be just waiting and doing nothing. In addition, these criteria are often inter-dependent. All of them are affected by the computation time or the communication time.

In the case of HMCS, suitability of tasks to devices should be considered. A task may exhibit different execution times on different computation devices. A task may have its own preferred device, that is, the task can have a smaller execution on the preferred device. One greedy method of scheduling is to assign each task to its preferred device. If the preferred devices of all tasks are the same, then that device would have a lot of tasks queued up, while the other devices would be mostly idle. This results in unbalanced scheduling and is not preferable. Thus, other scheduling criteria such as device availability should be considered. However, if we consider too many criteria, the scheduler might perform poorly.

Workload partitioning is often used on tasks that have divisible computations, for example, matrix multiplication. After workload partitioning, the different proportions of workload varies in performance. Thus, how to determine the partition percentage is an interesting research issue.
1.2 Motivation

Due to the widespread use of HMCS, scheduling parallel tasks on HMCS has become a popular research topic. Although there are many task scheduling algorithms, most of them do not consider divisible workload, whereas in parallel throughput driven systems, the task data can often be partitioned. Task scheduling is also known as a NP-complete problem [6]. In the results of executing scheduling algorithms, one would often find that devices are not fully utilized [7, 8]. The main reason for under utilized device is the dependencies among tasks and the varying workloads of tasks. Due to data dependency, a task has to wait for its predecessor task to complete processing the corresponding data. The idle time of processors is called a hole in a schedule. To eliminate the holes, the method backfilling [9, 10] moves the tasks backward to fill the hole. The dependencies of the backward tasks are not disordered. The defect of backfilling is that there is not a task to be backfilled sometimes. Besides, the task may not suitable to the hole due to the workload of the task is fixed. Therefore, we can partition the workload of a task onto different processors, such that the idle time of the device can be used to execute a part of the partitioned task. As a result, the makespan can be shortened and overall performance improved.

<table>
<thead>
<tr>
<th>$t_i$</th>
<th>$\text{Comp}_i(\text{CPU})$</th>
<th>$\text{Comp}_i(\text{GPU})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>$t_2$</td>
<td>16</td>
<td>7</td>
</tr>
<tr>
<td>$t_3$</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>$t_4$</td>
<td>10</td>
<td>6</td>
</tr>
</tbody>
</table>
Let us use the example task graph given in Figure 1.1 to illustrate why existing algorithms can be improved. There are four tasks, namely, $t_1$, $t_2$, $t_3$, $t_4$. The execution time of these tasks on the CPU and on the GPU are given in Table ??.

Further, the workload of each task is also given in Table ??, where workload means the amount of data to be processed by each task in terms of data units such as byte. In the task graph, each node represents a task and each edge represents the communication between two tasks. The label on an edge ($t_i$, $t_j$) represents the amount of communication time required for transmitting all the data required by the task $t_j$.

For illustration, we use the heterogeneous earliest-finish-time (HEFT) [11] to schedule the task graph. The scheduling results are given in Figure 1.2(a). The makespan of the HEFT schedule is 29 time units. In this schedule, we can observe that there
are three holes. If workload partitioning is used, as prepared in the Thesis, then the schedule will have a smaller makespan.

The result of workload partitioning is shown in Figure 1.2(b). The makespan now becomes 17.08. The improvement is almost half of the makespan of the original HEFT schedule. This example demonstrates that the result of workload partitioning efficiently improves current scheduling algorithms.
1.3 Thesis Organization

In Chapter 2, we present the related work on task scheduling and workload partitioning. In Chapter 3, we explain some definitions that will be used in Chapter 4. In Chapter 4, we give a complete illustration of our method. In Chapter 5, we evaluate our work and compare it with the state-of-the-art methods. Finally, we draw a conclusion in Chapter 6.
Chapter 2

Related Work

Task scheduling can be classified as static task scheduling and dynamic task scheduling. Dynamic task scheduling determines the task schedule at run time. When information about the tasks is retrieved during scheduling and used to change the schedule in a period, it is also called history-based scheduling algorithm. On the other hand, static task scheduling profiles all of the tasks before execution. Tasks are executed right after scheduling and the schedule is not changed at run time.

In this Chapter, we introduce several classic static task scheduling algorithms, for example, heterogeneous earliest finish time (HEFT) [11], Min-min [12, 13], and Max-min [12, 13]. Some work try to integrate scheduling methods based on the characteristics of the methods [14, 15]. They classify over ten algorithms and introduce them in detail. The performance of the algorithms is also compared in the paper. Besides, some workload partitioning methods are also introduced. Task scheduling methods usually consider coarse-grained tasks which cannot be partitioned. However in this work, we consider fine-grained tasks that can be partitioned. We learn how to
schedule and partition tasks efficiently from these methods.

2.1 Heterogeneous Task Scheduling Algorithm

Many algorithms [16] have been proposed for scheduling tasks in homogeneous multi-core systems. The methods proposed for scheduling tasks in homogeneous multi-core systems are adopted to schedule tasks in HMCS. However, the methods may not be suitable for HMCS. In HMCS, due to heterogeneity, different characteristics of the heterogeneous devices need to be considered. For example, the CPU is used to perform I/O jobs, control-oriented jobs, and light-parallel jobs. On the other hand, GPU is used for executing data-parallel tasks. Static task scheduling methods include heuristic scheduling algorithms such as Min-min, Max-min, and Sufferage [17]. Both Min-min and Max-min exploit the concept of minimum completion time (MCT) [12].

2.1.1 Heuristic Scheduling Algorithms

Min-min [12, 13], a classical method for task scheduling on heterogeneous multiprocessor systems was proposed in 1977. Min-min calculates the MCT of each ready task on each device. A task that has the minimum MCT is chosen and the task is assigned to the corresponding device. The MCT of the other tasks will be recalculated and updated. Then, the process of finding the task that has the minimum MCT is repeated until there is no task to be scheduled.

The algorithm of Max-min [12, 13] is similar to the algorithm of Min-min. Max-
Min also calculates the MCT of each ready task. However, Max-min instead chooses a task that has the maximum MCT and assigns it to the corresponding device. The MCT of other ready tasks are then updated. This process is repeated until there is no task to be scheduled.

The Sufferage [17] method helps to alleviate the problem of time wasted when a task is assigned to an inappropriate device. In the algorithm, all devices are assumed unassigned first and it calculates the MCT of ready tasks. Sufferage chooses a task randomly from the ready tasks and finds the device that allows the task to complete earliest. The order of choosing task is not important because each assignment of task scheduling considers all of the tasks. While the device is available, the chosen task is assigned to its corresponding device directly. Otherwise, it compares the sufferage value between the chosen task and the assigned task. The sufferage value is the difference between the minimum MCT and the secondary minimum MCT of the task, that is, the effect of assigning an inappropriate task. The task that has a larger sufferage value is assigned to the corresponding device. After all tasks are considered, it updates the MCT of the tasks and repeats the procedure above.

Among the above mentioned algorithms, Min-min performs well in many conditions. Sufferage performs as well as Min-min in some cases. However, Min-min does not perform well if most of the tasks have short execution times. Since Min-min executes the shorter tasks first, the few longer tasks left behind executes in an unbalanced manner. In this kind of example, Max-min executes the longer tasks earlier so that the shorter tasks keep the workload balancing in a schedule. Therefore, Max-min
performs better and more balanced than Min-min in the above situations.

In heuristic scheduling algorithms, there are some algorithms with list-based characteristic. List-based algorithms determine the priority of tasks first. HEFT calculates the ranks of the tasks using the computation time and communication time of the task and then it sorts ranks in descending order. The rank determines the priority that the rank and the priority are in direct proportion. Moreover, the rank makes sure that the dependency of the task in the schedule is taken into consideration because the rank of the task must be lower than the rank of its predecessor and higher than the rank of its successor. HEFT schedules with the task which has the highest rank. After determining their priorities, HEFT schedules the tasks by the priorities and calculates the earliest finish time of the tasks after each assignment. The procedure repeats until the last task finish scheduling.

Path-based Heuristic Task Scheduling (PHTS) [18] is also a list-based scheduling algorithm and it also uses the concept of critical path to assign the priorities to tasks. PHTS finds every possible paths from the entry node to the exit node in the task graph and computes ranks for each path by summing the computation time and communication time of tasks on a path. It sorts ranks of the paths in descending order and selects the first path. Afterward, PHTS selects from the beginning task of the selected path. If the task has no predecessor or the predecessors are scheduled, PHTS assigns it to its corresponding device with its earliest finish time. The dependencies among the tasks are not violated because the predecessors of the task are scheduled before the task. If there is no appropriate task, PHTS selects the next
path. It schedules tasks in the above mentioned manner until all tasks are scheduled.

For task scheduling, there are four important principles, namely suitability, locality, availability, and criticality (SLAC) especially for scheduling on heterogeneous parallel systems. The four principles are usually used in many scheduling algorithms, such as the algorithms we introduced above. Model driven runtime (MDR) [19] considers SLAC in its scheduling method. It adjusts the schedule step by step using the elements to make the schedule better. MDR considers suitability first because the performance of a task on the device is a key element. Afterward, it also takes locality, availability, and criticality into consideration. Locality and availability are meant to reduce the communication time and idle time on HMCS, respectively. The goal of using the critical path is to achieve significant reduction in the makespan of the schedule.

2.1.2 Enhanced Scheduling Algorithms

Apart from the conventional task scheduling, more criteria can be taken into consideration for scheduling algorithms. Some scheduling methods modify the schedule again after scheduling, and it is also known as rescheduling, such as backfilling [9, 10]. There are two backfilling methods: conservative-backfilling and EASY-backfilling. Conservative-backfilling moves the suitable tasks back to get the deadlines, while the movements do not disturb other tasks. EASY-backfilling does the same work, but the movements of EASY-backfilling are allowed to affect neighbor tasks. Both of them are static task scheduling methods and they estimate the execu-
tion time of the tasks. The accuracy of the estimated execution time of tasks is also an important issue for backfilling methods. If the execution time is overestimated, the task cannot be moved though they can be moved actually. For example, if the hole is 3 time units and the task that can be backfilled is 3 time units but it is estimated as 4 time units, we determine the task cannot fill the hole. On the other hand, if the execution time is underestimated, the task moves to the wrong positions and they may be rejected because it cannot be executed. For example, if the hole is 4 time units and the task that can be backfilled is 5 time units but it is estimated as 4 time units, the task is selected to fill the hole, but it is rejected.

The bin packing technique is used in [20], which targets on task scheduling in real-time systems. It adopts three scheduling policies, namely earliest deadline first (EDF), highest level first (HLF), and least space-time first (LSTF) to select the task and uses earliest estimated start time (EST) to select the processor. After adopting these policies, there are many holes in the schedule pattern. To eliminate these holes, the problem can be formulated as bin packing for filling the holes. Among the three policies First-Fit, Best-Fit and Worst-Fit of bin backing, First-Fit performs the best in most conditions.

2.2 Workload Partitioning Method

No matter how the scheduling algorithms are improved, as the applications change, it is impossible that there is no hole between tasks. In other words, there are inevitably some holes between tasks after scheduling. The holes are generated
because the task has to wait for its predecessor task to finish or there is not a suitable ready task to execute. There are some methods to alleviate the problem, for example, backfilling tries to shift the tasks to fill the holes. While using existing tasks to fill the holes, the policy is restricted by the sizes of holes, the sizes of tasks, and the dependency of the tasks. To achieve a high ratio of successfully filling holes, in this work, we try to “partition” a task into pieces to fit the holes. There are some workload partitioning methods as introduced below.

Workload partitioning is usually used to deal with applications with large workload, for example, astronomy application, seismology application and bioinformatics application [21]. One reason to partition the task is because of the storage constraint. In workload partitioning, a complex task graph can be partitioned into sub-graphs which makes the graph simpler because the complexity in each sub-graphs are reduced. To reduce the time complexity of the method, tasks are classified into three types: fan-in, fan-out, and pipeline jobs. Therefore, it does not have to check all of the tasks. Note that it should avoid deadlock in the schedule. After partitioning, the tasks are scheduled using the estimating factors, such as critical path, average CPU time and earliest finish time, and HEFT and Min-min algorithms are applied to the scheduler.

Workload partitioning technique can also be used on independent tasks. Workload Aware Task Scheduling (WATS) [22] focuses on homogeneous multi-cores, which are asymmetric due to the different frequencies of the cores. It sorts tasks in descending order and clusters them into groups. To reduce the makespan, some parts of the
task are partitioned from the slower core to the faster core. It also uses history-based method and preference list for task stealing to improve the schedule.

The above methods partition tasks into task groups and schedule them in an efficient way. However, they do not consider data partitioning of the task. Qilin [23] proposes an adaptive mapping method on HMCS. The purpose of Qilin is to make tasks balanced across cores on HMCS, so it maps the partitioned tasks to all processors in the platform. First, it samples the program for execution time with different problem sizes, and then uses linear regression for constructing the prediction models. Second, by finding the intersection point of models, the best partition proportion on the CPU and the GPU for the task is obtained. All of the tasks are sampled and recorded in a database. Qilin searches the database for the partition proportion while partitioning tasks on HMCS. Qilin proposes an effective method for workload partitioning. However, it does not propose a suitable scheduling method.

Another paper related to workload partitioning proposes a waterfall energy consumption model [24] for the power issue. It uses some task mapping methods, such as $\beta$-migration on GPU. Tasks could be partitioned into CPU sub-task and GPU sub-task. $\beta$-migration does not move part of CPU sub-task to GPU because some works on CPU cannot work on GPU. While CPU sub-task and GPU sub-task do not work in a balanced way, for a task on the GPU we may move $\beta$ proportion of the task to the CPU. The waterfall energy consumption model not only performs well in power reducing, but also decreases the execution time of the tasks. Although our target is not power consumption, we can still learn from these methods and try
to combine task scheduling and workload partitioning to fully use the resources of the devices.
Chapter 3

Preliminaries

We will introduce our system model and the related evaluation metrics in this Chapter. There will be explicit definitions for our models. The definitions will be used in Chapter 4.

3.1 System Model

The system model defines the architecture model and the application model. The architecture model introduces the scheduling environment and the application model defines the target tasks of scheduling. The details are described in the following subsections.

3.1.1 Architecture model

Our target system, HMCS includes different kinds of processors with multiple cores, including CPU and GPU. The multiple cores of CPU are all homogeneous, that is, we do not consider cores with different capabilities. This also holds for GPU cores
which are homogeneous. Generally, the CPU is designed to deal with all kinds of control-oriented jobs and the GPU is designed for parallel execution of data-oriented jobs. Therefore, the CPU is usually used to determine the mapping and the scheduling of the tasks. We use HMCS devices to execute tasks in parallel. The tasks are mapped to the corresponding devices according to the characteristics of the devices and the tasks. The definition of the device is given in Definition 3.1.

**Definition 3.1 (D)** \( D = \{d_1, d_2, \ldots, d_n\} \) is a set of \( n \) devices.

The assumptions of the device list as follows.

1. Our target system, HMCS includes different kinds of multiple CPU cores and multiple GPU cores.

2. We adopt static task scheduling in non-real-time systems.

### 3.1.2 Application Model

We usually use a task graph, such as a Directed Acyclic Graph (DAG) [25] to represent tasks with dependencies. Figure 3.1 shows a task graph model with four tasks, \( t_i, t_j, t_k, \) and \( t_l \). The tasks are allowed to execute on both the CPU and/or the GPU. The computation time of a task is also called the execution time, that is, the time duration required to execute a task. The computation time of the tasks are different on different devices. In addition to the computation time, the communication time between the tasks are usually depicted on the edge of the task.
graph. For example, $e_{i,j}$, $e_{i,k}$, $e_{j,k}$, and $e_{i,j} = (t_i, t_j)$ is the communication time between $t_i$ and $t_j$. The communication time is the amount of time required for transferring data from a task to another task. It is assumed that the communication between two tasks on the same device is negligible.

![Task graph model](image)

**Figure 3.1: Task graph model**

**Definition 3.2 (G)** A task graph $G = (V, E)$, is defined as follows.

1. $V$ is a set of nodes representing the tasks of an application, where each task $t_i = \text{Comp}_i$, $\text{Comp}_i(d)$ is the computation time on device $d$. Let $T = \{t_1, t_2, ..., t_n\}$ is a set of $n$ tasks.

2. $E$: $V \times V$ is a set of edges representing the dependency between two tasks, $(t_i, t_j) \in E$ indicates task $t_j$ must be executed after task $t_i$, where each $\text{Comm}_{i,j}$
is defined as the communication time transferring from $t_i$ to $t_j$. We assume $\text{Comm}_{i,j} = 0$ if $t_i$ and $t_j$ are mapped onto the same device.

**Definition 3.3 (S)** Each schedule $(S)$ corresponds to a device $d$. The schedule is represented by 3-tuple $(T, \text{Starttime}, \text{Endtime})$. where

1. $st_j \in \text{Starttime}$ is the start time of task $j$.
2. $et_j = st_j + \text{Comp}_j(d) \in \text{Endtime}$ is the end time of task $j$ when executed on device $d$.

**Definition 3.4 (Task partitioning)** The task partitioning is defined as follows.

1. Task partitioning divides the workload into two parts, one of which is processed by the task on the originally scheduled device, while the other part is processed by a similar task on another device that has some idle time. The two parts of the task can be executed in parallel.
2. The computation time $\text{Comp}$ and the communication time $\text{Comm}$ for the two similar tasks (resulting from partitioning) are proportionately re-calculated based on the amount of workload to be processed by each task.

**Definition 3.5 (EST)** The earliest start time (EST) of $t_j$ with a predecessor $t_i$ is $\text{max}(et_i + \text{Comm}_{i,j})$. 

21
The tasks in our target system must satisfy the following conditions:

1. The data processed by the tasks must be divisible, that is, we can partition the data of the tasks into two or more partitions.

2. The tasks are non-preemptive, such that each task can finish its work without interference.

3. Target tasks have dependencies with each other. Therefore, we should consider their precedence relationships while scheduling.

### 3.1.3 Schedule Model

Based on the computation time of tasks and the communication time among tasks, the allocation tasks to the devices and the order of executing the tasks are decided by a scheduling algorithm. A schedule consists of task execution times and idle times. The idle times of a schedule are due to the waiting for the completion of communication between tasks and the unbalanced utilization of the devices. We call each idle time period as a **hole**.

**Definition 3.6 (Hole)** A hole in a schedule is an idle time period in the schedule on the device $d$ with the start time $Starttime$ and the end time $Endtime$ of hole. It is defined as $h = (Starttime, Endtime, d)$. A hole can be classified into two types: closed hole and open hole.
1. Assume $t_i$ and $t_j$ are mapped to device $d$. A closed hole is the idle time between $t_i$ and $t_j$. The hole at the beginning of the schedule is also a closed hole. The ending time of a closed hole is the beginning time of the task after the hole.

2. An open hole is the idle time of device $d$ from the last $t_i$ to the end of the schedule. The ending time of an open hole is taken as the makespan of the schedule.

While scheduling, the hole can be classified into different two types: constant hole and variable hole.

1. Assume $t_i$ is a candidate task used to fill the closed hole between $t_j$ and $t_k$, $t_k$ is the successor of $t_j$. If $t_i$ has a dependency with $t_k$, then the hole is a variable hole, with respect to $t_i$. Similarly, the open hole on the device is also a variable hole.

2. Assume $t_l$ is a candidate task for the closed hole between $t_j$ and $t_k$, $t_k$ is the successor of $t_j$. If $t_l$ has no dependency with $t_k$, the hole is constant with respect to $t_l$.

Figure 3.2(a) and Figure 3.2(b) illustrate the different types of holes of the schedule. In Figure 3.2(a), assume $t_2$ is a candidate task used to fill the hole from time 0 to time 6 on the GPU device. Because $t_2$ has no dependency with $t_3$, the hole is a constant hole, with respect to $t_2$. The open hole after $t_3$ on the GPU is a variable
Figure 3.2: Two types of the hole while scheduling (a) Constant hole (b) Variable hole

hole. In Figure 3.2(b), we assume that $t_1$ is a candidate task used to fill the hole from time 0 to time 6 on the GPU. It is assumed that $t_1$ has a dependency with $t_3$ and the hole is a variable hole, with respect to $t_1$.

In Figure 3.3, there are one closed hole and one open hole in the schedule. The closed hole is generated because of the task dependency $(t_1, t_3)$, that is, $t_3$ has to not only wait for $t_1$ to finish execution, but also wait for the data communication from $t_1$ to $t_3$ to complete before it can start. In contrast, an open hole is produced because of unbalanced workload across different devices. In our example, the computation time of $t_2$ is so long that the schedule of the GPU is idle for a long time.
3.2 Task Partition Profile Database

We employ a database of program profiling similar to the one proposed in [23]. Tasks are first profiled on each device and the task partition that results in balanced workload across two or more devices is considered to be the best partition and recorded in the database. This database helps save the time for calculating the partition proportions by a priori profiling. The time of calculation does not affect the makespan scheduling result, but it can reduce the time of static analysis.

![Diagram](image)
3.3 Metrics

In static task scheduling, the goal is to complete the execution of all tasks as soon as possible. Therefore, our target is to obtain the smallest makespan.

**Definition 3.7 (Makespan)** The makespan of a schedule is the time interval from the start of the first task to the end of the last task across all devices.

The makespan is usually meaningful only in non-realtime task scheduling. Another important metric is the EST, which is used widely in scheduling algorithms. We calculate the EST of the tasks using the computation time and the communication time of the tasks. The task can start executing after its predecessor has finished and the required data are transferred.

To improve the task scheduling result, namely makespan, we partition the data of the tasks. In our method, it is assumed that a task can only be partitioned once. After partitioning, the computation time of a task and the communication time between the task and its predecessor and successor tasks will become the corresponding proportion of their original computation time and the original communication time, respectively. For example, suppose there is a task that executes on core1 for 5 seconds and executes on core2 for 4 seconds. Assume that the task is assigned for execution on core2. Once we partition the task as 80% on core1 and 20% on core2, the task executes 4 seconds on core1 and executes 0.8 seconds on core2. The EST of the partitioned tasks are re-calculated and the tasks scheduled after them are re-scheduled.
Chapter 4

Workload Partitioning and Re-Scheduling on HMCS

In this Chapter, we introduce our proposed method in details. We divide it to two parts while explaining our overall work, namely workload partitioning and task re-scheduling. The workload partitioning method is used in the task re-scheduling method.

4.1 Workload Partitioning and Re-scheduling

The workload partitioning and re-scheduling method proposed in this Thesis is shown in Figure 4.1. In this work, it is assumed that a task graph is scheduled using an existing scheduling algorithm, including those widely used such as Min-min[12, 13], Max-min[12, 13], HEFT[11], and Sufferage[17]. This given schedule is then refined to improve the makespan. Initially, we search for holes in the task schedule. Once a hole is found, we try to fill the hole so as to improve the makespan. For each hole, we find the candidate tasks which can possibly fill the hole and reduce the time
of the hole. The candidate tasks are those tasks whose execution times on other devices overlap with the hole. If there is no candidate task, we search for the first task that is scheduled after the hole on other devices and consider partitioning that task. If there is not any task on the other devices, we move to the next hole. Each candidate task is classified into a variable hole or a constant hole. Then, based on the hole type, a corresponding strategy is used to find the partition proportion of the candidate task. For a variable hole the profiling-based partitioning (ProfPart) described in Section 4.2.1 is used. For a constant hole, the on-the-fly partitioning (OtfPart) described in Section 4.2.2 is used. In partitioning a candidate task, if the start time of the candidate task is earlier than the start time of the hole, we consider only the part of the candidate task execution that starts at the start of the hole.
Figure 4.1: Re-scheduling algorithm
Definition 4.1 (Partition Gain) *Given a candidate task, the partition gain is the amount of time by which the end time of the hole can be made earlier after partitioning the candidate task.*

For each candidate task, its partition gain can be calculated. We choose the candidate task that has the maximum gain for workload partitioning. If the maximum gain is non-negative, we partition the candidate task with the maximum gain and move on to find the next hole. On the contrary, if the maximum gain is negative and if the candidate with this negative maximum gain was partitioned using ProfPart (i.e. the best partition from the database), then its gain is re-calculated using OtfPart. If the candidate task with the negative maximum gain was considered to be partitioned by OtfPart, we give up filling this hole, and move on to we find the next hole. We keep doing the above process until the end of the schedule.

4.1.1 Algorithm of the Refine Method

We describe our refine method in detail using Algorithm 1. The input of the algorithm is the task schedule generated by an existing method and the output is the partitioned result of the schedule. In the refine method, we need some information about the tasks and the schedule. The information we need is listed in line 1 to line 9 in the algorithm. We use $t_{Ci}$ and $t_{Gi}$ as the schedule task index. For each $CPU_i$, $GPU_i$, the tasks that are scheduled on the device are measured using a schedule task index $t_{Ci}$ or $t_{Gi}$. First, we search for hole from the beginning of the schedule. In line 12, the algorithm checks to see whether we found a true hole, which means the time
of hole, $h$, is greater than zero. If $h$ is equal to zero, the index task is followed by another task and there is no idle time between them. If we find a hole, we search for candidate tasks corresponding to the hole. In line 14 and line 15, we retrieve the partition portion of each candidate task and calculate the partition gain obtain by each of them. In line 16 and line 18, if the gain of the chosen task is negative and the partition proportion of the chosen task is not calculated by $OtfPart$, we try to get a better partition proportion and find the chosen task in the candidate tasks. Otherwise, the step moves onto line 18, where the gain of the chosen task is non-negative. From line 19 to line 21, we re-schedule the whole task schedule and set the flag of the partitioned task to true. From line 22 to 25, we move the search time index formed on the device with the processed hole. We keep performing the above steps until the end time of the index task is equal to the makespan of the schedule, which means the index task is the last task of the schedule. The refine algorithm finishes and returns a new task schedule.
Algorithm 1: Refine Algorithm

1. **input**: Task schedule
2. **output**: New schedule result
3. \(S_{CPU_i}: \) CPU schedule, \(S_{CPU_i} = \{S_{CPU_1}, S_{CPU_2}, \ldots, S_{CPU_n}\}\)
4. \(S_{GPU_i}: \) GPU schedule, \(S_{GPU_i} = \{S_{GPU_1}, S_{GPU_2}, \ldots, S_{GPU_m}\}\)
5. \(t_{Ci}: \) Current task in \(S_{CPU_i}, \) initialized as the first task of the \(S_{CPU_i}\)
6. \(t_{Gi}: \) Current task in \(S_{GPU_i}, \) initialized as the first task of the \(S_{GPU_i}\)
7. \(T_{cand}: \) Set of candidate tasks corresponding to the currently considered hole
8. \(Chosen: \) The task chosen to be partitioned
9. \(Gain(t): \) Gain of the task \(t\)
10. \(Flag(t): \) Record whether \(t\) is partitioned or not, the default of \(Flag(t)\) is false
11. \(WPM(t): \) The workload partitioning method considered for partitioning task \(t\)

while \((\max(\text{Endtime}(t_{Ci}), \text{Endtime}(t_{Gi})) < \text{makespan})\) do

1. \(h = \text{search hole}(t_{Ci}, t_{Gi}) \) // Algorithm 2
2. if \((\left(\text{Endtime}(h) – \text{Starttime}(h)) \neq 0\right)\) then
3. \(T_{cand} = \text{search cand}(h) \) // Algorithm 3
4. \(Gain(T_{cand}) = \text{get gain}(\text{get proportion}(T_{cand}, h))\)
5. \(Gain(Chosen) \leftarrow \max(Gain(T_{cand}))\)
6. while \((Gain(Chosen) < 0 \land WPM(Chosen) \neq \text{OtfPart})\) do
7. \(Gain(Chosen) = \text{get gain}(\text{get proportion}(Chosen, h))\)
8. \(Gain(Chosen) \leftarrow \max(Gain(T_{cand}))\)
9. if \((Gain(Chosen) \geq 0)\) then
10. \(\text{re schedule}(S_{CPU_i}, S_{GPU_i}, h, Chosen)\)
11. \(\text{Flag}(Chosen) = \text{true}\)
12. if \(h \in S_{CPU_i}\) then
13. \(t_{Ci} + 1 \) // boundary condition issue resolved in program
14. else
15. \(t_{Gi} + 1\)

return \(S_{GPU_i}, S_{CPU_i}\)
In our refine method, we deal with only one hole at a time from the task schedule. The algorithm for searching hole is given as Algorithm 2. The input of the algorithm is the set of current tasks on each device and our goal is to find the earliest hole. We choose the earliest task between every current task and record the start time of the hole and the end time of the hole. The hole returns to the Algorithm 1.

**Algorithm 2: Search Hole Algorithm**

<table>
<thead>
<tr>
<th>input</th>
<th>$t_{Ci}, t_{Gi}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>output</td>
<td>$h$</td>
</tr>
<tr>
<td>1 $t = \text{Argmin}(\text{Endtime}(t_{Ci}), \text{Endtime}(t_{Gi}))$</td>
<td></td>
</tr>
<tr>
<td>2 $\text{Starttime}(h) = \text{Endtime}(t)$</td>
<td></td>
</tr>
<tr>
<td>3 $\text{Endtime}(h) = \text{Starttime}(t + 1)$</td>
<td></td>
</tr>
<tr>
<td>4 return $h$</td>
<td></td>
</tr>
</tbody>
</table>

While searching for candidate tasks, we should not choose a task that starts much later than the hole. Instead, we should choose a candidate task that starts near the beginning time of the hole. This is because choosing a task scheduled much later than the hole, for partitioning result in the need to alter a long of dependent tasks. On one hand, if we choose a candidate task that starts earlier than the beginning time of the hole, the task may be a predecessor of tasks scheduled before the hole. If we move a part of such a candidate task to the hole, the task may finish later causing the tasks scheduled behind the candidate task may also be forced to start later. On the other hand, if we choose a task that starts much later than the beginning time of the hole as the candidate task, the partition gain of the task will probably be negative. The reason for this negative gain is the candidate task may be a successor of the task after the hole. Therefore, partitioning this task may be invalid because
such a movement could violate the dependency between the tasks. Therefore, we choose candidate tasks that have their execution times overlapping with the time of the hole. The algorithm is described from line 1 to line 7 in Algorithm 3. If we do not find any candidate task by this strategy, we search for tasks that begin after the hole, except the tasks on the same device with the hole. We choose the earliest tasks on each device as the candidate tasks. We describe the strategy from line 8 to line 10. We check whether the candidate task is partitioned in line 11.

<table>
<thead>
<tr>
<th>Algorithm 3: Search Candidate Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong> : h</td>
</tr>
<tr>
<td><strong>output</strong>: $T_{cand}$</td>
</tr>
<tr>
<td>1 foreach task $t$ in the schedule do</td>
</tr>
<tr>
<td>2         if $(\text{Endtime}(t) &gt; \text{Starttime}(h)) \land (\text{Endtime}(t) \leq \text{Endtime}(h))$ then</td>
</tr>
<tr>
<td>3         $T_{cand} = T_{cand} \cup {t}$</td>
</tr>
<tr>
<td>4 else if $(\text{Starttime}(t) \geq \text{Starttime}(h)) \land (\text{Starttime}(t) &lt; \text{Endtime}(h))$ then</td>
</tr>
<tr>
<td>5         $T_{cand} = T_{cand} \cup {t}$</td>
</tr>
<tr>
<td>6 else if $(\text{Starttime}(t) \leq \text{Starttime}(h)) \land (\text{Endtime}(t) \geq \text{Endtime}(h))$ then</td>
</tr>
<tr>
<td>7         $T_{cand} = T_{cand} \cup {t}$</td>
</tr>
<tr>
<td>8 if $(T_{cand} = \emptyset)$ then</td>
</tr>
<tr>
<td>9         foreach $d \in D \land d \neq d_h$ do</td>
</tr>
<tr>
<td>10        $T_{cand} = T_{cand} \cup {\text{Argmin}(t)</td>
</tr>
<tr>
<td>11        $T_{cand} = T_{cand} \cap {t \in T_{cand}</td>
</tr>
<tr>
<td>12        return $T_{cand}$</td>
</tr>
</tbody>
</table>
4.2 Workload Partitioning Method

A workload partitioning method divides the data workload of a task into two proportions such that a part of them can be processed concurrently on two different devices. The workload partitioning results determine how much of a schedule hole on a device can be filled and by how much the makespan of the task schedule can be reduced. A poor workload partition may even cause more holes to be created and this increase the makespan. To effectively fill a schedule hole, we use different strategies to deal with different kinds of holes. To fill a variable hole, we employ a profiling-based partitioning method as described in Section 4.2.1. To fill a constant hole, we propose an on-the-fly partitioning method as described in Section 4.2.2.

4.2.1 Profiling-based Partitioning

Since the location and size a variable hole depends on the task being partitioned and used to fill the hole, it becomes very complex to decide on an appropriate partition proportion based on the current schedule hole size. Thus, instead we make use of a task partition principle database that contains previously recorded best partition proportions for each kernel execution on each pair of compute devices. The database is constructed by profiling the kernels on different devices and recording the best partition proportion for each kernel and each pair of devices. Although the proportion of the partitioned task used to fill a hole may not fill the hole to the full, the communication time between the partitioned task and the task scheduled after the hole could be reduced by workload partitioning. After re-scheduling, the task
scheduled after the hole can usually start earlier than in the previous schedule and the hole may become smaller. As a result, using the task partition principle database can help fill the variable hole. Not only is the hole filled by the partitioned task, but the size of the hole is also reduced by workload partitioning. For example, there is a hole from 0 to 6 on GPU in Figure 4.3(a) and the hole is a variable hole respect to $t_1$. After partitioning $t_1$, the hole is filled and the size of the hole is also reduced in Figure 4.3(b).

4.2.2 On-the-fly Partitioning

In contrast to the variable holes, a constant hole is not dependent on the candidate task chosen for filling the hole, thus, the finish time of a constant hole is not affected by workload partitioning. We propose an OtfPart method that can try to fill the hole completely, without allowing any postponement of the task scheduled after the hole. The basic rule employed in OtfPart is as follows. Suppose $t_v = (Comp_v)$ is selected for partitioning and suppose $\alpha$ is the proportion of task partitioned to fill a hole, where $0 < \alpha < 1$. Let $\{t_u\}$ be the set of predecessors of $t_v$, $h = (\text{Startime}, \text{Endtime}, d)$ be the hole to be filled.

$$max_{t_u}(\text{Endtime}(t_u)) - \text{Endtime}(h) \geq max_{t_u}(\text{Comm}_{u,v}) + \alpha \times Comp_v(d) \quad (4.1)$$

Depending on whether a candidate task needs communication time with its predecessor, there are two cases:
1. no communication time, and

2. with communication time.

For the first case, Equation 4.2 is used to determine $\alpha$. For the second case, Equation 4.3 is used to determine $\alpha$.

$$\alpha \times Comp_v = Endtime(h) - Starttime(h) \quad (4.2)$$

$$\alpha \times Comp_v + EST_v = Endtime(h) \quad (4.3)$$

An example for OtfPart is in Figure 4.3(b) and in Figure 4.4(a). In Figure 4.3(b), the hole after $t'_1$ on the CPU is a constant hole respect to $t_3$. The partitioning result of $t_3$ is in Figure 4.4(a), where the hole is filled completely. The gain computed for an instance of successful on-the-fly partitioning is zero because the hole is constant. However, this does not mean OtfPart is ineffective in reducing the overall makespan. By partitioning a task on the critical path of a schedule, OtfPart can still reduce the overall makespan. The ending time of the hole is later unless the candidate task is an ancestor of the task after the hole. In this condition, workload partitioning may cause an unpredictable result, that is, the ending time of the hole may be later.
4.3 Example of Workload Partitioning and Scheduling

Let us use the example task graph given in Figure 4.2 to illustrate why existing algorithms can be improved. There are four tasks, namely, $t_1$, $t_2$, $t_3$, $t_4$. The computation time of these tasks on the CPU and on the GPU are given in Table 4.1. In the task graph, each node represents a task and each edge represents the communication between two tasks. The label on an edge ($t_i$, $t_j$) represents the amount of

![Figure 4.2: Task graph of example](image)
communication time required for transmitting all the data required by the task \( t_j \).

For illustration, we use the heterogeneous earliest-finish-time (HEFT) to schedule the task graph. The scheduling results are given in Figure 4.3(a). The makespan of the HEFT schedule is 29 time units. In this schedule, we can observe that there are three holes. If workload partitioning is used, as proposed in the Thesis, then the schedule will have a smaller makespan.

As shown in Figure 4.3(a), we find that the first hole occurs in the GPU device schedule from time 0 to time 6. Tasks \( t_1 \) and \( t_2 \) are the candidate tasks, that is,
Figure 4.4: (a) Scheduling result of example after second partition (b) Scheduling result of example after third partition
the tasks with executions overlapping with the hole. For \( t_1 \), it is a variable hole and for \( t_2 \) it is a constant hole. We manually tune the partition proportion, such that task \( t_1 \) is partitioned into two parts \( t'_1 \) and \( t''_1 \), where \( t'_1 \) executes on CPU and \( t''_1 \) on GPU. The partition proportion unit is 10 percent. The execution time of the task partitions will be almost the same on the two different devices after partitioning. We compute the execution times and communication times of the partitions of candidate tasks. Their execution times and communication times are the proportions of the original ones. After that, we compare the beginning time of the task after the hole between the original schedule and the partitioned schedule. If the beginning time of the task is earlier, this means the hole is smaller. We also call this improvement as gain. In contrast, if the beginning time of the task is later, the gain of the partition is negative. After comparison, the gain of \( t_1 \) is greater than the gain of \( t_2 \) and the gain is positive. The partitioning result is presented in Figure 4.3(b). We go on to find the next hole, which occurs after \( t''_1 \) in the GPU schedule. Since \( t'_1 \) is a partitioned task, it should not be considered as a candidate for partitioning again. Thus, there is nothing to do for this hole. The third hole is found after the task \( t'_1 \) in the CPU schedule. We manually tune the partition percentage of the candidate task \( t_3 \). We found that the hole becomes bigger after partitioning. This means the gain is negative and we try to get a better partition percentage of the candidate task \( t_3 \). We use OtfPart to get a new partition percentage and the new gain corresponding to task \( t_3 \) is now zero. Therefore, we partition \( t_3 \) into two parts \( t'_3 \) and \( t''_3 \). The next hole is in the GPU schedule after \( t'_{3'} \) completes execution. Considering candidate task \( t_2 \), it can be
partitioned to fill this hole. Note that if the beginning time of the candidate task is earlier than the beginning time of the hole, we only consider partitioning the part of the candidate task starting from the beginning time of the hole to the ending time of the task. The partition result is shown in Figure 4.4(b). A new hole appears in the CPU schedule after $t'_2$ completes execution. Now, there are two candidate tasks, namely $t''_2$, and $t_4$. After comparison, it is found that partitioning task $t_4$ is more preferable because it can result in a smaller makespan. The result of partitioning $t_4$ is shown in Figure 4.5. The makespan now becomes 17.08. The reduction is almost half of the makespan of the original HEFT schedule. This example demonstrates that the result of workload partitioning efficiently improves current scheduling algorithms.
Figure 4.5: Scheduling result of example after fourth partition
Chapter 5

Experiment Results

We certify the benefit of our algorithm here. We introduce the experiment setup first in this Chapter and implement our re-scheduling method with several task scheduling algorithms. Besides, we compare our re-scheduling method to other methods that can also re-schedule the tasks.

5.1 Experiment Setup

In this Chapter, we describe the architecture of our experiment device. The simulator will be introduced in Chapter 5.1.2.

5.1.1 Architecture Setup

We use Intel(R) Core(TM) i7-2600 CPU 4 cores @ 3.40GHz as the experiment device and the computer equips DDR3 16 GB Memory. We use the linux system to program and the OS is Ubuntu 11.10 (GNU/Linux 3.0.0-25-generic x86_64). Our program uses gcc 4.4.6 compiler.
5.1.2 Simulation Framework

Our simulation framework is in Figure 5.1. We use the task generator to generate the task graph and put it into the scheduling simulator. Moreover, the schedule that the scheduling simulator generates is put into the schedule analyzer to analyze the performance of our refine method. Each work will be described in detail later.

Figure 5.1: Simulation framework

The input of our re-scheduling method is a task graph. We use Task Graph For Free (TGFF) [26] to generate the task graph randomly. TGFF has many parameters about the task graph and the parameters control the characteristics of the graphs, the characteristics of the tasks, the periods of the tasks, the deadlines of the tasks, and the tables of the parameters. We choose some parameters and adjust these parameters to generate the suitable task graph for our scheduling method. These parameters are:

- **tg_cnt**: This parameter determines how many task graphs are generated once. If tg_cnt is greater than one, each task graph adopts the same parameters but their graph tables are different.

- **task_cnt**: This parameter determines how many tasks are generated in a task graph. We can set an error value about this parameter. Therefore, the number
of the tasks may not the same in each task graph. We can only confirm the minimum number of the tasks in the task graph.

- **task_degree**: This parameter determines the complexity of a task. We can decide the maximum number about how many tasks can transmit in from other tasks and transmit out to other tasks.

- **table_cnt**: This is the number of the task type tables and transmit type tables. Each task graph can adopt different task type table and transmit type table.

- **task_type_cnt**: This parameter determines the number of the task types. That is, how many kinds of the tasks we have in a task type table.

- **trans_type_cnt**: This parameter determines the number of the transmit types. That is, how many kinds of the transmit lines we have in a transmit type table.

- **type_attrib**: This parameter is the characteristic of the task types and the transmit types. A task usually has not only one characteristic in a task graph.

We implement four scheduling algorithms in the scheduling simulator and implement our workload partitioning and refine method. After simulating, both of the schedule after scheduling and the schedule after refine are analyzed. The metrics we analyze are makespan, turnaround time, number of the partitioned task, number of the hole, the time of the maximum hole, and the total amount of the hole. We also measure the time the scheduling algorithms and the refine algorithm spend. The performance improvement can be seen in next Section.
5.2 Performance Evaluation

We illustrate the information of the task graph and the algorithms we adopted in the experiment in detail. The performance results are also performed in this Chapter and they will be analyzed in detail.

5.2.1 General Cases

In our experiment, we set the parameters of our experiment.

- The number of the devices is 2.
- The average number of tasks is 20.
- Both the degree in and out a task is 2.
- The number of the task graphs is 50.

5.2.2 Adopted Algorithms

We adopt the scheduling algorithms Min-min, Max-min, and HEFT. These scheduling methods are introduced in the Chapter 2.1.1 that have no workload partitioning concept in the algorithms. We use the schedule of these algorithms as the input of our refine method.
5.2.3 Performance

We evaluate the makespan of these methods we adopt. In Figure 5.2, the makespan of all refine algorithms are decreased. The makespan decreases most in the refine method using Max-min. The average decreased makespan of the refine method using Max-min is 14%. For each task graph, the one decrease the makespan the most is in the refine method using HEFT. It can decrease the makespan by 50% once.

![Figure 5.2: Makespan of scheduling methods](image)

Then, we evaluate the turnaround time of the algorithms. The result of turnaround time is in Figure 5.3. Max-min performs the best about the decrease of turnaround time, it decrease 13.4% of the total turnaround time. The decrease about turnaround time is not much of Min-min. Because of workload partitioning may not move part of the task to a preferred device and the start time of two separate tasks may not be centralized, the workload partitioning does not have obvious decrease of the turnaround
Figure 5.3: Turnaround time of scheduling methods

Figure 5.4 represents the reduction of hole times. Our method tries to reduce the time of the holes to decrease the makespan. In Figure 5.4, the time of the holes decrease a lot and each task decrease almost half of the hole time. HEFT performs the best in decreasing hole time, it decreases 50.3% in average.

Except the total time of hole, we also evaluate the number of the hole which results in Figure 5.5. The time of the hole decreased a lot but the number of the hole is not always decreased. A hole that is filled may become to one or more smaller holes. Therefore, the number of the holes may not be decreased and possibly be increased.
Figure 5.4: Total amount of hole of scheduling methods

<table>
<thead>
<tr>
<th></th>
<th>HEFT</th>
<th>Min-min</th>
<th>Max-min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule</td>
<td>176.3</td>
<td>142.4</td>
<td>264.56</td>
</tr>
<tr>
<td>Re-schedule</td>
<td>85.95</td>
<td>78.3</td>
<td>148.24</td>
</tr>
</tbody>
</table>

Figure 5.5: Number of hole of scheduling methods

<table>
<thead>
<tr>
<th></th>
<th>HEFT</th>
<th>Min-min</th>
<th>Max-min</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule</td>
<td>4.8</td>
<td>4.04</td>
<td>4.76</td>
</tr>
<tr>
<td>Re-schedule</td>
<td>4.08</td>
<td>5.48</td>
<td>6.96</td>
</tr>
</tbody>
</table>
In spite of the metrics above, we evaluate the run time while scheduling. The time spend on scheduling is almost the same to the time spend on re-scheduling. The scheduling result of runtime is in Figure 5.6.

![Runtime](image)

**Figure 5.6: Runtime of scheduling methods**

We also calculate the number of the partitioned tasks. The result of the percentage of partitioned tasks in a task graph is in Figure 5.7. We calculate the number of the partitioned tasks because workload partitioning may cause a few overhead to the schedule.

In spite of the number of the partitioned tasks, we analyze the distribution of partitioned tasks on task graphs in Figure 5.8. The number of partitioned task is centralized in 1 to 11. Only Max-min has more task graphs that the number of partitioned tasks is greater than 11. Max-min performs the worst before refining, so refine method do a great job in partitioning tasks.
Figure 5.7: Percentage of partitioned tasks in a task graph

Figure 5.8: Distribution of partitioned tasks on task graphs
5.3 Comparison

In Chapter 5.2.3, we find that Max-min performs the best in decreasing the makespan in our refine algorithm. Therefore, we use Max-min in our refine method to compare with the other methods that can also refine the schedule. Our refine method use both OtfPart and ProfPart in workload partitioning. To confirm the advantage of the classification of the holes, we compare our refine method to the other two methods, one is the OtfPart only and another one is ProfPart only. The parameters of the experiment are as follows.

- The number of the devices is 2.
- The average number of tasks is 20.
- Both the degree in and out a task is 2.
- The number of the task graphs is 50.

5.3.1 Comparative Results

Table 5.1 is the comparison between the original schedule and the schedule after refining. The values in Table 5.1 are the average of one graph. In Table 5.1, all of the refine methods decrease the makespan but our refine method performs the best. The refine method performs the best in makespan, turnaround time, and hole time and the number of partitioned tasks is the highest. The runtime of the three methods are similar. OtfPart has the lowest number of hole, which means OtfPart fills the hole
the best. However, OtfPart does not have the lowest time of the hole confirms that we should use different method in different types of holes.
<table>
<thead>
<tr>
<th>Method</th>
<th>Makespan</th>
<th>Turnaround time</th>
<th>Hole time</th>
<th>Number of hole</th>
<th>Runtime (μs)</th>
<th>Number of partitioned tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schedule</td>
<td>403.92</td>
<td>1178.32</td>
<td>264.56</td>
<td>4.76</td>
<td>1121.18</td>
<td>0</td>
</tr>
<tr>
<td>Refine</td>
<td>347.59</td>
<td>1019.92</td>
<td>148.24</td>
<td>6.96</td>
<td>1331.22</td>
<td>7.78</td>
</tr>
<tr>
<td>OtfPart</td>
<td>361.67</td>
<td>1069.86</td>
<td>183.65</td>
<td>5.64</td>
<td>1319.56</td>
<td>6.9</td>
</tr>
<tr>
<td>ProfPart</td>
<td>366.06</td>
<td>1041.31</td>
<td>165.48</td>
<td>7.38</td>
<td>1368.16</td>
<td>6.28</td>
</tr>
</tbody>
</table>
5.4 Communication to Computation Ratio (CCR)

The schedule is varied with the variance of the computation time and the communication time of the tasks. We use different CCR to observe the difference of the schedule. We use Max-min method for the CCR experiment. CCR is defined as follows, which $g$ is the average of the communication time and the computation time in a task graph.

$$CCR = \frac{Comm_g}{Comp_g}$$ (5.1)

The parameters of the experiment are listed as follows.

- The number of the devices is 2.
- The average number of tasks is 20.
- Both the degree in and out a task is 2.
- The number of the task graphs is 50.

5.4.1 CCR Results

We set CCR of the experiment as 5, 1, and 0.2. The result of CCR experiment is in Figure 5.9. The larger CCR means the communication time is higher. The schedule cannot be refined much because workload partitioning may cause more communication and the partitioned task cannot fill the hole. On the other hand, the
smaller CCR means the communication time is smaller and the workload partitioning method can be adopted easily. The decrease of makespan can reach to 17.9% when CCR equals to 0.2.

![Figure 5.9: Makespan comparison of CCR](image)

<table>
<thead>
<tr>
<th>Time</th>
<th>Schedule</th>
<th>Re-schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>484.7</td>
<td>470.07</td>
</tr>
<tr>
<td>1</td>
<td>404.46</td>
<td>353.89</td>
</tr>
<tr>
<td>0.2</td>
<td>372.34</td>
<td>305.41</td>
</tr>
</tbody>
</table>
Chapter 6

Conclusions

The problem of idle times in the schedule exists for a long time. We analyze many conditions of the hole in detail to deal with the problem. We use workload partitioning and task refine method to solve the problem. We use HEFT, Min-min, and Max-min as our input schedule method. We adopt the schedule of these methods to improve. While refining the schedule, we try our best to decrease the amount of the hole. Nevertheless, our workload partitioning method combines two workload partitioning strategy and we compare the workload partitioning strategies with each other. As a result, our re-scheduling method performs well in reducing the amount of hole. The reduction percentage of the hole can reach to 50% and the makespan can be decreased by 14%. Although the decrease of the total amount of hole is great, we thought the decrease of makespan could be better. The method can not only used in the device that has one CPU and one GPU. The analyze of different combination of the devices is an important future work.
Bibliography


