ADVANCES IN PARALLEL PROGRAMMING FOR ELECTRONIC DESIGN AUTOMATION

BY

CHUN-XUN LIN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor Martin D. F. Wong, Chair
Professor Wen-Mei Hwu
Professor Deming Chen
Dr. Jinjun Xiong, IBM T. J. Watson Research Center

ABSTRACT

The continued miniaturization of the technology node increases not only the chip capacity but also the circuit design complexity. How does one efficiently design a chip with millions or billions transistors? This has become a challenging problem in the integrated circuit (IC) design industry, especially for the developers of electronic design automation (EDA) tools. To boost the performance of EDA tools, one promising direction is via parallel computing. In this dissertation, we explore different parallel computing approaches, from CPU to GPU to distributed computing, for EDA applications.

Nowadays multi-core processors are prevalent from mobile devices to laptops to desktop, and it is natural for software developers to utilize the available cores to maximize the performance of their applications. Therefore, in this dissertation we first focus on multi-threaded programming. We begin by reviewing a C++ parallel programming library called Cpp-Taskflow. Cpp-Taskflow is designed to facilitate programming parallel applications, and has been successfully applied to an EDA timing analysis tool. We will demonstrate Cpp-Taskflow's programming model and interface, software architecture and execution flow. Then, we improve Cpp-Taskflow in several aspects. First, we enhance Cpp-Taskflow's usability through restructuring the software architecture. Second, we introduce task graph composition to support composability and modularity, which makes it easier for users to construct large and complex parallel patterns. Third, we add a new task type in Cpp-Taskflow to let users control the graph execution flow. This feature empowers the graph model with the ability to describe complex control flow. Aside from the above enhancements, we have designed a new scheduler to adaptively manage the threads based on available parallelism. The new scheduler uses a simple and effective strategy which can not only prevent resource from being underutilized, but also mitigate resource over-subscription. We have evaluated the new scheduler on both micro-benchmarks and a very-large-scale

integration (VLSI) application, and the results show that the new scheduler can achieve good performance and is very energy-efficient.

Next we study the applicability of heterogeneous computing, specifically the graphics processing unit (GPU), to EDA. We demonstrate how to use GPU to accelerate VLSI placement, and we show that GPU can bring substantial performance gain to VLSI placement. Finally, as the design size keeps increasing, a more scalable solution will be distributed computing. We introduce a distributed power grid analysis framework built on top of DtCraft. This framework allows users to flexibly partition the design and automatically deploy the computations across several machines. In addition, we propose a job scheduler that can efficiently utilize cluster resource to improve the framework's performance.

To my parents, for their love and support.

ACKNOWLEDGMENTS

First I would like to express sincere gratitude to my advisor, Prof. Martin D. F. Wong, who has patiently guided me through my doctoral study. I want to thank him for giving invaluable advice on research direction and helping me develop research skills. Specifically, I am grateful to him for all our meetings where I can always gain new insights from the discussion. I am also grateful to my doctoral committee, Prof. Wen-Mei Hwu, Prof. Deming Chen and Dr. Jinjun Xiong. I want to thank them for listening to my presentation, and providing many useful comments and suggestions to this dissertation.

I want to thank Dr. Chih-Hung Liu for encouraging me to study abroad and helping me greatly with my PhD application. Many thanks to Dr. Tsung-Wei Huang for teaching me many research techniques and much programming knowledge, and I am fortunate to participate in those interesting projects with him. I want to thank my lab-mates Zigang Xiao, Leslie Hwang, Daifeng Guo, Haitong Tian, Tin-Yin Lai, Guannan Guo and Chih-Shin Wang for their assistance in my research, and for helping me clarify my thoughts via numerous stimulating discussions. I am also grateful to my friends Jhih-Chian Wu, Iou-Jen Liu, Hsiao-Lun Wang, Hsi-Ping Chu, Pao-Yi Tang, Chen-Hsuan Lin and Sitao Huang for sharing their life stories and work experience with me and bringing so much joy and fun to my life. I want to thank two visiting scholars: Prof. Fan Zhang for showing me around the office when I joined the lab, and Prof. Hung-Ming Chen from NCTU for sharing his study and career experience with me.

Last but not least, I want to express my deepest gratitude to my parents, sister and brother. With their endless love and support, I am able to get over all the difficulties and pursue my goal wholeheartedly.

TABLE OF CONTENTS

CHAPTER 1 DISSERTATION OVERVIEW 1

CHAPTER 2 CPP-TASKFLOW PROGRAMMING SYSTEM 4
2.1 Task Dependency Graph 4
2.2 Software Architecture and Execution Flow 6

CHAPTER 3 ADAPTIVE WORK-STEALING SCHEDULER 11
3.1 Introduction 11
3.2 Adaptive Work-Stealing Scheduler 13
3.3 Evaluation 23
3.4 Conclusion 36

CHAPTER 4 TASK GRAPH COMPOSITION AND CONDITIONALS 37
4.1 Introduction 37
4.2 New Task Dependency Graph 40
4.3 Composable Tasking 42
4.4 Conditional Tasking 47
4.5 Conclusion 56

CHAPTER 5 ANALYTICAL PLACEMENT WITH GPU 57
5.1 Introduction 57
5.2 Wirelength Computation 58
5.3 Density Computation 64
5.4 Experimental Results 69
5.5 Conclusion 70

CHAPTER 6 A DISTRIBUTED POWER GRID ANALYSIS FRAMEWORK 73
6.1 Introduction 73
6.2 Distributed Power Grid Analysis 75
6.3 Distributed Power Grid Analysis based on Stream Graph . . . 78
6.4 Application-specific Resource Control Plug-in 83
6.5 Experimental Results 85
6.6 Conclusion 90

REFERENCES 91

CHAPTER 1

DISSERTATION OVERVIEW

The dissertation can be divided into three parts: Chapters 2, 3 and 4 are dedicated to multi-threaded programming, and especially we will focus on a parallel programming library called Cpp-Taskflow. In Chapter 5 we demonstrate how the performance of VLSI placement can benefit from GPU. Chapter 6 presents a distributed computing framework for power grid analysis. We give a brief overview of subsequent chapters below.

Chapter 2 reviews Cpp-Taskflow which is a C++ parallel programming library developed by our group [1]. Cpp-Taskflow arises from the need of an efficient approach to parallelize an EDA application with complex parallel patterns. The goal of Cpp-Taskflow is to enable programmers to quickly parallelize their applications with task-based programming model. For this purpose, Cpp-Taskflow adopts task dependency graph as the programming model, and provides intuitive tasking interface for ease of programming. Cpp-Taskflow is open-source [2] and has been used in several applications. In this chapter we will go over Cpp-Taskflow's programming interface, software architecture and internal execution flow, then we introduce the new elements that we add to Cpp-Taskflow in Chapters 3 and 4, respectively.

In Chapter 3, we present an efficient work-stealing scheduler to execute the task dependency graph of Cpp-Taskflow. Maintaining a scheduler to manage a pool of threads is a frequently used method in parallel programming libraries, as this method can prevent the overhead of repeatedly spawning threads. The scheduler has a great impact on the overall library's performance as it controls the thread activities and coordinates task execution. It employs a simple and efficient strategy to adapt the number of active threads to available parallelism. This strategy not only can prevent resource underutilization but can also minimize resource waste to achieve substantial energy saving. In addition, the scheduler can maintain decent throughput in a shared environment where multiple parallel processes are running con-

currently. We provide an analysis on the scheduler's thread management to prove the effectiveness of our scheduler. The experimental results show that our scheduler can deliver good performance, energy efficiency and throughput on a VLSI timing analysis tool.

In Chapter 4[1], we propose several enhancements to Cpp-Taskflow's programming model. The first and foremost enhancement is to separate the task graph and executor. This allows users to create multiple task graphs and graphs can be executed multiple times and in arbitrary order. The second enhancement is the graph composition which allows users to compose small and simple graphs into a large and complex graph. Last but not least, we add a new tasking type: conditional tasking to enable users to control the graph execution flow at runtime. The conditional tasking is a revolutionary breakthrough as it removes the restrictions that a task graph must be acyclic, and each task must be executed exactly once. Users can use conditional tasking to iterate parts of a graph multiple times, or conditionally bypass the execution of some tasks. These new capabilities make it very easy to build task graphs for applications with complex control flow.

In Chapter 5[2], we develop GPU techniques to accelerate VLSI placement. VLSI placement decides the positions of cells on the chip, and the placement result will have a significant impact on subsequent steps in the physical design flow. To derive a good placement, state-of-the-art VLSI placement methods adopt an analytic approach which typically involves a huge amount of computation and is therefore very time-consuming. In this chapter, we propose to use GPU to accelerate the wirelength and density computations in VLSI placement. We utilize the sparse graph property to speed up the wirelength computation via sparse matrix multiplication, For density computation, we come up with a computation flattening technique to mitigate the load balancing issue, and we take advantage of the CUDA stream to overlap the data transfer with computation to further reduce the overhead. The experiment results show GPU can bring considerable performance gain to VLSI placement.

[1]Part of the content in this chapter was published in IEEE High Performance Extreme Computing Conference, 2019 [3], and is used here with permission.

[2]The content of this chapter was previously published in Design, Automation and Test in Europe Conference, 2018 [4], and is used here with permission.

Finally, in Chapter 6[3], we demonstrate a distributed power grid analysis framework using DtCraft [6]. DtCraft is a distributed execution engine that takes a stream graph and automatically deploys the computations on the machines in a cluster. In the stream graph model, users encapsulate the computations in nodes which will be invoked when data arrive, and the directed edges specify the data flow between nodes. With the stream graph abstraction, the proposed framework can perform flexible domain decomposition regardless the available hardware resources. We have conducted experiments to show the framework's flexibility. In addition, we also propose a new scheduler that can better utilize the cluster resource to improve the performance.

[3]The content of this chapter was previously published in Great Lakes Symposium on VLSI, 2018 [5], and is used here with permission.

CHAPTER 2

CPP-TASKFLOW PROGRAMMING SYSTEM

In this chapter, we review a C++ parallel programming library: Cpp-Taskflow proposed by Huang et al. [1]. Cpp-Taskflow is motivated by an EDA application: OpenTimer [7], which is a circuit timing analysis tool. Timing analysis is a crucial part of the physical design flow and is very time-consuming. To speed up OpenTimer, the authors need an efficient way to program the parallel patterns which are highly irregular. As a result, they develop Cpp-Taskflow to serve for this purpose. Compared to existing parallel programming libraries such as OpenMP [8] and Intel Threading Building Blocks (TBB) [9], writing parallel code with Cpp-Taskflow is relatively easy, especially for complex parallel patterns. The evaluation of OpenTimer has shown that Cpp-Taskflow can achieve comparable performance to OpenMP with taking much less coding effort [1]. We will go over Cpp-Taskflow's programming model, user interface, software architecture and execution flow in the following sections.

2.1 Task Dependency Graph

The programming model of Cpp-Taskflow is *task dependency graph* which is a directed acyclic graph. In task dependency graph, a node is a task which encapsulates a computation to be executed on a thread, and the directed edges describe the dependency between nodes. To use Cpp-Taskflow, users have to first decompose the application into dependent tasks, and then specify the task dependency by adding directed edges between tasks. Listing 2.1 is an example of using Cpp-Taskflow to construct a task dependency graph. In this example, we first create an object of type `tf::Taskflow`, and then use the `emplace` method to add four lambda objects to create four tasks. The `emplace` method returns a `tf::Task` object, and we can use the `precede`

method to specify the dependency between those task objects. The lambda objects will be stored in the tasks and invoked during runtime. For the tasks which do not take any input argument, we call them *static tasking*. Static tasking means those tasks will not make any change to the task dependency graph at runtime.

In contrast to static tasking, *dynamic tasking* allows a task to spawn new task dependency graph during graph execution. Both static and dynamic taskings are constructed by the emplace method, and the major difference is that dynamic tasking has to take an input argument of type *tf::SubflowBuilder*. Listing 2.2 shows an example of dynamic tasking. The subflow object can be used to create a task dependency graph via the aforementioned graph construction methods. The task graph in the subflow object will be scheduled to execution after the parent task ends. There are two modes that users can select to schedule the subflow graph: joined and detached modes. The joined mode guarantees the subflow graph will finish before scheduling the successor tasks of its parent task, while in detached mode there is no restriction on the execution order between the subflow graph and the parent graph. Dynamic tasking allows users to flexibly create task dependency graphs during runtime to generate more parallelism. Another benefit of dynamic tasking is to enable users to implement common computing patterns such as recursion, where the number of tasks in those patterns cannot be known before execution

When the task dependency graph is constructed, the graph can be dispatched to execution via either the taskflow object's `wait_for_all`, `dispatch` or `silent_dispatch` method. The `wait_for_all` method will block the caller until the task graph finishes execution. In contrast, the `dispatch` method returns a `std::shared_future` object to let the caller query the execution status asynchronously, while the `silent_dispatch` method does not return anything. Once a task dependency graph is dispatched to execution, the task graph will be destroyed at the end of execution.

```
tf::Taskflow flow;

int a,b,c,d;

// Create tasks
```

```
auto A = flow.emplace([&](){ a = 1; });
auto B = flow.emplace([&](){ b = a + 1; });
auto C = flow.emplace([&](){ c = a + 1; });
auto D = flow.emplace([&](){ d = b + c; });

// Specify dependency
A.precede(B, C);
B.precede(D);
C.precede(D);
```

Listing 2.1: Create a task dependency graph using Cpp-Taskflow.

```
tf::Taskflow flow;

// Dynamic tasking
auto S = flow.emplace([](auto &subflow){
  // Use subflow to construct a task dependency graph
  auto S1 = subflow.emplace([](){ printf("S1\n"); });
  auto S2 = subflow.emplace([](){ printf("S2\n"); });
  auto S3 = subflow.emplace([](){ printf("S3\n"); });
  auto S4 = subflow.emplace([](){ printf("S4\n"); });

  S1.precede(S2, S3, S4);
});
```

Listing 2.2: Dynamic tasking in Cpp-Taskflow.

2.2 Software Architecture and Execution Flow

In this section, we will go over Cpp-Taskflow's main data structures and describe the task graph execution flow. A taskflow object internally stores a task graph (`tf::Graph`) and an executor(`tf::Executor`). A task graph is a list of nodes (`tf::Node`) where each node stores a `callable` object [10], and other graph related data such as pointers to successors and a dependency counter (number of predecessors). Whenever the `emplace` method is called, the taskflow object creates a node and forwards the given task to the node's

callable object. The callable object will invoke the task and schedule its successor tasks at runtime. Algorithm 1 is the content of the callable object. In the `invoke_task` function, the captured task is invoked first based on its tasking type (line 1-17). For static tasking, we simply invoke the task without giving any input argument (line 1-4). If the task is dynamic tasking, we invoke the task with a subflow object (line 6-9). Next, if the dynamic tasking is in joined mode, we let the sink tasks of the subflow object precede parent task (line 10-12) and schedule the source tasks (line 13), and then terminate the `invoke_task` function (line 15). The direct termination is to ensure correct execution order in joined mode, where the new spawned task graph needs to finish before scheduling the parent task's successors. Otherwise, if the mode is detached we directly schedule the source tasks in the subflow object (line 13) and proceed to schedule the successor tasks. After the task has been invoked, we decrement the dependency of its successor tasks (line 19-27). The successor tasks whose dependency is met will be immediately dispatched to execution (line 21-25).

An executor is a thread pool that maintains a set of threads to carry out dispatched tasks, and Cpp-Taskflow adopts a work-stealing method to balance the workload between threads [11]. The executor spawns a set of threads (denoted as workers) on initialization. Each worker has a local queue and a cache to store the tasks ready for execution. The local queue is a double-ended queue which allows the owner to add and pop tasks from the bottom, while others can only steal tasks from the top [12]. The cache is a task holder that enables a worker to reserve a ready task for continuous execution. In addition to the local queues, the executor maintains a master queue for non-worker threads to add tasks. A worker will first carry out all tasks in local queue and cache. Then, the worker tries to randomly steal tasks from other workers and the master queue. Once the steal succeeds, the worker will execute the task and repeat the whole procedure. If the worker fails to obtain any task after a fixed number of steals, the executor suspends the worker by adding the worker into a idler list. To improve both load balancing and performance, a worker will attempt to wake up a suspended worker based on a probability.

Here we use Figure 2.1 as an example to illustrate task dependency graph execution with two workers: W1 and W2. For simplicity, we assume workers will not be suspended before the graph finishes execution, and workers can

Algorithm 1: The `invoke_task` function.

```
Input: node
Input: task: the given task stored in node
Input: executor
Input: w: the worker
if t == static tasking then
    /* Static tasking                                          */
    invoke(t);
else
    /* Dynamic tasking                                         */
    subflow ← node.subgraph();
    if t has never been invoked then
        invoke(t, subflow);
    end
    if subflow.joined() then
        subflow.sink_tasks().precede(node);
    end
    schedule(subflow.sources());
    if subflow.joined() then
        return ;
    end
end
/* Update the dependency of successor tasks                    */
for s ∈ node.successors() do
    if AtomDec(s.dependencies) == 0 then
        /* If the dependency is met, dispatch the successor
           task to execution                                   */
        if w.cache ≠ NIL then
            executor.schedule(w.cache);
        end
        w.cache ← s;
    end
end
```

immediately steal the task if there exists one. Figure 2.1 is a task dependency graph with eight tasks (not including the graph spawn by dynamic tasking). Whenever a graph is dispatched to execution, Cpp-Taskflow first creates an object of type `tf::Topology` to record the runtime data of the graph. The topology object collects the tasks without predecessors (denoted as source tasks, e.g. task A and B) in the task graph, and lets the tasks without successors (denoted as sink tasks, e.g. task G and H) precede a node which

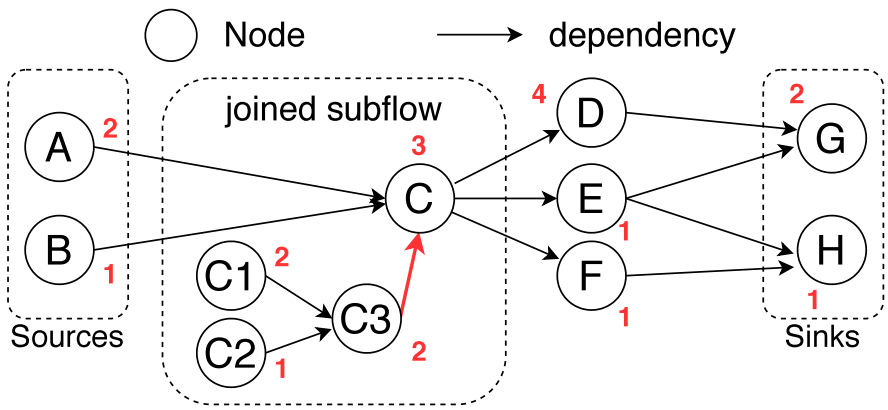


Figure 2.1: An example to illustrate executing a task dependency graph. The numbers in red are the required execution time of tasks. The red edge is added deliberately by executor to respect the execution order.

will set up the future object at the end of execution. After the topology object has built up, the source tasks are added into the executor's master queue to initiate the execution. In this example we assume W1 gets task A and W2 gets task B in the beginning. Next we illustrate the task execution along the timeline below:

1. **T=1**, task B finishes and W2 decrements the dependency of task C. Since W2 has no remaining tasks, W2 will start stealing tasks randomly.

2. **T=2**, task A finishes and W1 decrements the dependency of task C. Since task C's dependency is met, W1 will continue executing task C.

3. **T=5**, task C finishes and a new task graph is spawned. W1 lets task C3 precede its parent task C. Then, W1 will cache task C1 and add C2 to W1's local queue. W2 subsequently steals task C2 from W1.

4. **T=6**, task C2 finishes and W2 decrements the dependency of C3.

5. **T=7**, task C1 finishes and W1 continues execution on task C3.

6. **T=9**, task C3 finishes. W1 will revisit task C and decrement the dependency of task D, E and F. Task D will be cached by W1 and tasks E and F will be added to W1's local queue. Then, W2 steals task E from W1's queue.

7. **T=10**, task E finishes and W2 decrements the dependency of task G and H. W2 steals task F from W1's queue.

8. **T=11**, task F finishes and W2 decrements the dependency of task H. W2 continues executing task H.

9. **T=12**, task H finishes. W2 has no tasks in local queue and thus starts random stealing.

10. **T=13**, task D finishes and W1 decrements the dependency of task G. W1 continues executing task G.

11. **T=15**, task G finishes. Now all sink nodes are executed and W1 will set up the future object and mark the task graph as finished.

CHAPTER 3

ADAPTIVE WORK-STEALING SCHEDULER

3.1 Introduction

Work stealing has been proved to be an efficient approach for parallel task scheduling on multi-core systems and has received wide research interest over the past two decades [11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]. Several task-based parallel programming libraries and language have adopted work-stealing scheduler as the runtime for thread management and task dispatch such as Intel Threading Building Blocks (TBB) [9, 24], Cilk [13, 25], X10 [19, 26], Nabbit [27], Microsoft Task Parallel Library (TPL) [28], and Golang [29]. The efficiency of the work-stealing scheduler can be attributed to the way it manages the threads: The scheduler spawns multiple threads (denoted as workers) on initialization. Each worker has a double-ended queue storing the tasks ready for execution, and a worker can only add newly spawned tasks into its queue. A worker first carries out all tasks in its queue, and then becomes a thief to randomly steal tasks from others. When a thief has successfully stolen a task, it restores to a normal worker and commences executing the task. The key is to have thieves actively steal tasks. By doing this the scheduler is able to balance the workload and maximize the performance.

However, implementing an efficient work-stealing scheduler is not an easy job, especially when dealing with a task dependency graph where the parallelism could be very irregular and unstructured. Due to the decentralized architecture, developers have to sort out many implementation details to efficiently manage workers such as deciding the number of steals attempted by a thief and how to mitigate the resource contention between workers. The problem is even more challenging when considering throughput and energy efficiency, which have emerged as critical issues in modern scheduler

designs [15] [30]. The worker management can have a huge impact on these issues if it is not designed properly. For example, a straightforward method is to keep workers busy in waiting for tasks. Apparently, this method consumes too much resource and can result in a number of problems, such as decreasing the throughput of co-running multithreaded applications and low energy efficiency [15] [30]. Several methods have been proposed to remedy this deficiency, e.g., making thieves relinquish their cores before stealing [16] or backing off a certain time [17] [24], or modifying OS kernel to directly control CPU cores [15]. Nevertheless, these approaches still have drawbacks, especially from the standpoints of solution generality and performance scalability.

In this chapter, we propose a work-stealing scheduler with provably good worker management for executing task dependency graphs. Our scheduler employs a simple yet effective strategy that adaptively adjusts the number of thieves by tracking the number of workers that are executing tasks. This strategy has three advantages: First, it ensures one thief will keep looking for tasks when any worker is executing tasks, which can prevent resource from being underutilized. Second, our strategy only uses a reasonable number of workers to meet the parallelism at any time, which can minimize the resource waste without compromising performance. Lastly, this strategy has very little overhead. Workers can quickly carry out their tasks without being slowed down by the extra management work. With this strategy, our scheduler can make efficient use of workers to achieve good performance under different parallelism. Meanwhile, this strategy effectively mitigates the resource waste by reducing unnecessary steals, and therefore our scheduler is energy-efficient and can maintain good throughput when co-running multithreaded processes. We summarize the contributions of our scheduler below:

- **An adaptive scheduling strategy**: We develop an adaptive scheduling strategy for executing task dependency graph. The strategy is simple; no sophisticated data structures or complex algorithms are required and thus the overhead is small. The experimental results show our scheduler can efficiently utilize CPU resource to achieve good performance.
- **Provably good worker management**: We proved our scheduler can prevent the under-subscription problem and effectively mitigate the over-subscription problem. Our scheduling algorithm is efficient in balancing

working threads on top of available task parallelism.

- **Energy efficiency**: Our scheduler is very energy-efficient in that it reserves thieves to steal only when there exists a worker executing tasks. We also show that our scheduler will put most thieves put into sleep when tasks are scarce, which effectively reduces resource waste and saves energy.

We evaluated the proposed scheduler on two benchmark sets: a set of micro-benchmarks and a very-large-scale integration (VLSI) timing analyzer. We use the Linux utility `perf` to measure the CPU utilization, runtime and energy usage of ours and the scheduling approach proposed by Aurora et al. [16] (denoted as ABP) and a modified approach from Ding et al. [15]. The micro-benchmarks show our scheduler can utilize the computing resources effectively to accommodate different degrees of parallelism. Specifically, in an extreme case with linear task graph, the CPU utilization of our scheduler is 1.2 while ABP is 31.9, which highlights the effectiveness of our scheduler's worker management. The second experiment is a real workload: VLSI static timing analysis. This experiment demonstrates that the scheduler not only achieves scalable performance but is also energy-efficient. On the largest circuit, our scheduler achieves 15% less runtime and 36% less energy consumption than ABP. Finally, we show the scheduler can maintain good throughput when co-running multithreaded applications.

3.2 Adaptive Work-Stealing Scheduler

In this section we present the details of the proposed work-stealing scheduler. We first outline our scheduler's architecture and associated data structures. Next we describe the proposed worker management approach and its implementation with pseudo code. Lastly, we provide an analysis on our worker management to show its efficiency.

3.2.1 Scheduler Overview

Figure 3.1 shows a task dependency graph (left) and the architecture of the proposed scheduler (right). Our scheduler consists of a set of workers, a master queue, a lock and a notifier. On initialization our scheduler spawns

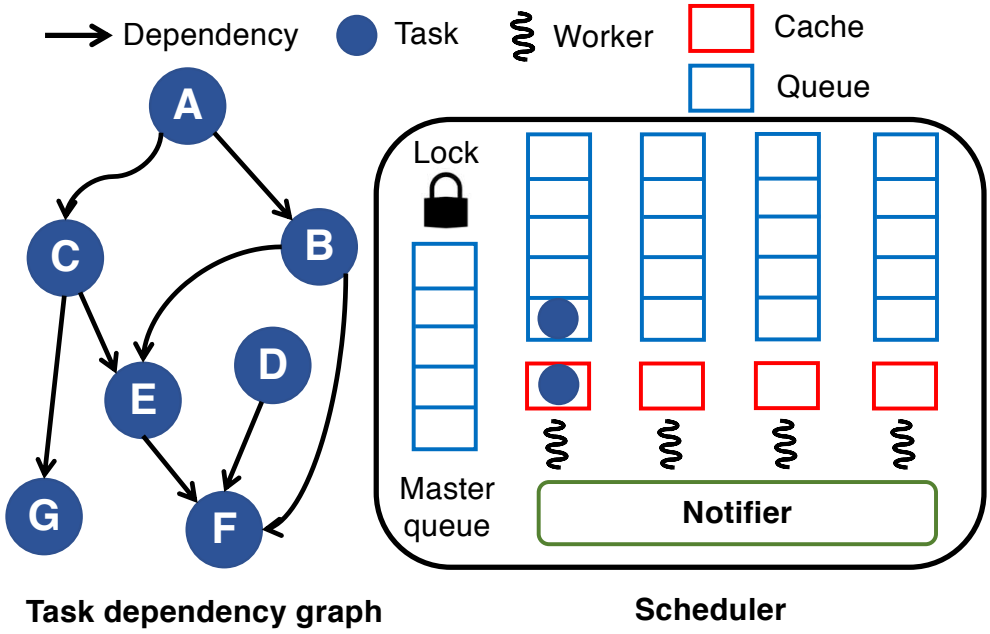


Figure 3.1: A task dependency graph and the architecture of our scheduler.

workers waiting for tasks. Each worker is equipped with a queue and a cache to store tasks ready for execution. After users create a task graph, they add the source nodes to the master queue and notify workers via the notifier to start execution. Algorithm 2 is the pseudo code of task insertion from users. In Algorithm 2, users first acquire the lock (Algorithm 2 line 1) which prevents concurrent insertion to the master queue. Then users add tasks into the master queue (Algorithm 2 line 2:4) and notify waiting workers (Algorithm 2 line 6). A worker will continue pulling tasks from its queue or others (including the master queue) for execution. When a worker finishes a task, it automatically adds successive tasks ready for execution to its own queue or cache. A worker's queue allows only the owner to add tasks, while only non-worker threads (such as the main thread controlled by users) can add tasks into the master queue. A cache is simply a task holder and only the owner can access its cache. The cache enables the worker to prefetch a task when the worker adds tasks. For example, in Figure 3.1 a worker can add the task B into its cache and C to its queue after it finishes the task A. With the cache, we can facilitate task retrieval by reducing the queue access.

Algorithm 2: Task insertion from users

```
Input: tasks: a set of ready tasks
lock();
for t in tasks do
    master_queue.push(t);
end
unlock();
notifier.notify_one();
```

3.2.2 Data Structures

The queue and the notifier are two important data structures in our scheduler. We implement the queues (both the worker's queue and master queue) based on the Chase-Lev algorithm [12] [31]. Access to the queue is non-blocking and the queue capacity can grow if more space is needed. The queue provides three operations:

1. Push: Add a task into the bottom of the queue. Only the queue's owner can use this operation.

2. Pop: Retrieve a task from the bottom of the queue. Only the queue's owner can use this operation.

3. Steal: Retrieve a task from the top of the queue. Any worker can use this operation.

The `notifier` is a synchronization component that is capable of (1) putting workers that are waiting for tasks into sleep and (2) notifying one or all waiting workers when new tasks are present or the scheduler terminates. In our scheduler, we use the `EventCount` struct from the Eigen library [32] as the `notifier`. The usage of `EventCount` is similar to a condition variable. The notifying thread sets a condition to true and then signals waiting workers via `EventCount`. On the other side, a worker first checks the condition and returns to work if the condition is true. Otherwise, the worker updates the `EventCount` to indicate it is waiting and checks the condition again. If the condition is still false, the worker is put into sleep via the `EventCount`.

3.2.3 Worker Management

Recall that our scheduler spawns workers on initialization and Algorithm 3 contains the pseudo code for spawning workers and a worker's control flow. In the `spawn` function, the scheduler spawns N workers where N is specified by users. The scheduler makes workers execute the `worker_loop` function after they are spawned. The `worker_loop` function consists of two steps. In the first step `exploit_task`, a worker executes a ready task and the tasks in its queue (Algorithm 3: line 10) until its queue becomes empty. Next, the worker leaves the `exploit_task` and calls the `wait_for_task` (Algorithm 3: line 11) to start stealing tasks. If the worker successfully steals a task, it returns from the `wait_for_task` and repeats the first step. Otherwise, the worker is put into sleep via the notifier to wait for task notification. When the scheduler terminates, the `wait_for_task` returns `false` and the worker exits the while-loop.

Algorithm 3: spawn

```
   Input: N: number of workers
 1 Function spawn(N):
 2     for i ← 0 to N do
 3         workers.emplace_back(worker_loop(i));
 4     end
 5 ;
 6 Function worker_loop(id):
 7     w ← workers[id];
 8     t ← NIL;
 9     while true do
10         exploit_task(t, w);
11         if wait_for_task(t, w) == false then
12             break;
13         end
14     end
15 return
```

One major contribution of this work is the adaptive scheduling algorithm, which is implemented in the `exploit_task` and `wait_for_task` functions. The main idea of this algorithm is to maintain at least one thief (except when all workers are executing tasks) when a worker is executing tasks. This is different from prior research where they unconditionally keep one or more thieves busy in waiting tasks [15][16], whereas we keep thieves only when

there exists potential parallelism. We achieve this by using two counters: `num_actives` and `num_thieves` to adaptively adjust the number of thieves.

Algorithm 4 is the pseudo code of `exploit_task` function. In this function, the worker first increments the `num_actives` (Algorithm 4: line 2) and checks the `num_thieves` (Algorithm 4: line 2). If `num_thieves` is zero and this is the first increment on `num_actives`, then the worker notifies a waiting worker (Algorithm 4: line 3) and proceeds to execute the task. The worker continues fetching and executing tasks from its cache (Algorithm 4: line 8) and queue (Algorithm 4: line 10) until both become empty. Then the worker decrements the `num_actives` (Algorithm 4: line 13) and returns. Obviously, the `num_actives` records the number of workers that are executing tasks, and a non-zero `num_actives` implies there could be tasks in a queue.

After a worker returns from `exploit_task`, the worker starts stealing tasks by invoking the `wait_for_task` function. Algorithm 5 is the pseudo code of the `wait_for_task` function. In the `wait_for_task`, the thief first increments the `num_thieves` (Algorithm 5: line 2) and conducts random stealing by invoking the `explore_task` function (Algorithm 5: line 4). Algorithm 6 is the pseudo code of the `explore_task` function. In the `explore_task`, the thief first randomly selects a victim (Algorithm 6: line 4) which could be other workers or the master queue. Then it tries to steal a task from the victim (Algorithm 6: line 5:9). If the steal fails, the thief will attempt to steal for a certain number of times. When the number of failed steals is greater than a pre-defined threshold, `steal_bound` (Algorithm 6: line 14), the thief invokes a `yield` system call every time after each failed steal. A thief stops stealing if it still cannot obtain any task after yielding `yield_bound` times (Algorithm 6: line 17). The thief returns from `explore_task` in either one of the three conditions:

- The thief successfully steals a task (Algorithm 6: line 11).
- The scheduler terminates (Algorithm 6: line 3),
- The thief fails to obtain any task after a fixed number of attempts (Algorithm 6: line 18).

In the first case, the thief decrements `num_thieves` (Algorithm 5: line 5) and notifies a waiting worker if it is the last thief (Algorithm 5: line 6). For the other two cases, the thief first updates the notifier (Algorithm 5: line

10) to indicate it is waiting. Next the thief checks the master queue and tries to steal a task if the queue is non-empty (Algorithm 5: line 11:22). The thief returns (Algorithm 5: line 18) if it successfully steals a task from the master queue or goes back to steal tasks if it failed (Algorithm 5: line 20). Otherwise, the thief proceeds to check the scheduler's status. If the scheduler shuts downs (Algorithm 5: line 23), the thief notifies all waiting workers (Algorithm 5: line 25) and then decrements the `num_thieves` and returns. If all preceding conditions do not hold, the thief decrements the `num_thieves` and then either continues to steal if `num_actives` is non-zero (Algorithm 5: line 31) and it is the last thief, or goes into sleep (Algorithm 5: line 33).

Algorithm 4: exploit_task

```
Input: t: a task holder, w: the worker's data structure
1  if t ≠ NIL then
2      if AtomInc(num_actives) == 1 and num_thieves == 0 then
3          notifier.notify_one();
4      end
5      do
6          execute(t);
7          if w.cache ≠ NIL then
8              t ← w.cache;
9          else
10             t ← pop(w.queue);
11         end
12     while t ≠ NIL;
13     AtomDec(num_actives);
14 end
```

3.2.4 Analysis

We show the scheduler's worker management is very efficient in two fronts: (1) At least one thief exists when there is a worker executing tasks. (2) It mitigates the thieves over-subscription problem by putting most thieves into sleep after they failed to steal. We first define the states of a worker:

Definition A worker is **active** if it is exploiting tasks (Algorithm 4: line 2:13), otherwise, the worker is **inactive**.

Algorithm 5: wait_for_task

```
Input: t: a task, w: the worker's data structure
Output: A Boolean value to indicate continuation of worker-loop
1  wait_for_task:
2  AtomInc(num_thieves);
3  explore_task:
4  if explore_task(t, w) and t ≠ NIL then
5  |  if AtomDec(num_thieves) == 0 then
6  |  |  notifier.notify_one();
7  |  end
8  |  return true;
9  end
10 notifier.prepare_wait(w);
11 if master_queue is not empty then
12 |  notifier.cancel_wait(w);
13 |  t ← steal(master_queue);
14 |  if t ≠ NIL then
15 |  |  if AtomDec(num_thieves) == 0 then
16 |  |  |  notifier.notify_one();
17 |  |  end
18 |  |  return true;
19 |  else
20 |  |  go to explore_task;
21 |  end
22 end
23 if scheduler stops then
24 |  notifier.cancel_wait(w);
25 |  notifier.notify_all();
26 |  AtomDec(num_thieves);
27 |  return false;
28 end
29 if AtomDec(num_thieves) == 0 and num_actives > 0 then
30 |  notifier.cancel_wait(w);
31 |  go to wait_for_task;
32 end
33 notifier.commit_wait(w);
34 return true;
```

Definition An inactive worker is **sleeping** if it has been suspended by the notifier (Algorithm 5: line 33).

Definition An inactive worker is a **thief** if it is not exploiting tasks (Algo-

Algorithm 6: explore_task

```
Input: t: a task holder, w: the worker's data structure
Output: t
num_failed_steals ← 0;
num_yields ← 0;
while scheduler not stops do
    victim ← random();
    if victim == w then
        t ← steal(master_queue);
    else
        t ← steal_task_from(victim);
    end
    if t ≠ NIL then
        break;
    else
        num_failed_steals ← num_failed_steals + 1;
        if num_failed_steals ≥ steal_bound then
            yield();
            num_yields ← num_yields + 1;
            if num_yields == yield_bound then
                break;
            end
        end
    end
end
```

rithm 4: line 2:13) nor sleeping.

Lemma 1. *When a worker is active and at least one worker is inactive, one thief always exists.*

Proof. Assume there exists one active worker and one inactive worker. The inactive worker is either awake (Algorithm 5: line 1:28) or sleeping (Algorithm 5: line 33). If the inactive worker is awake, then it is a thief and the lemma holds. Otherwise the inactive worker is sleeping and it must have decremented the `num_thieves` without seeing any active worker (Algorithm 5: line 29). This happens only when the active worker just enters the `exploit_task` function and is about to increment `num_actives` (Algorithm 4: line 2). Subsequently the active worker shall wake up a thief (Algorithm 4: line 3) and the lemma holds.

□

Lemma 1 is important to our scheduler as it prevents the *under-subscription* problem.

Definition An under-subscription problem means:

$$T = 0 \quad \text{and} \quad 0 < Q < W$$

where

$$\begin{aligned} Q &: \text{ number of non-empty queues} \\ &\quad \text{(excluding the master queue)} \\ T &: \text{number of thieves} \\ W &: \text{number of total workers} \end{aligned}$$

In the following discussion we exclude two special conditions where all workers are active and all workers are inactive. An under-subscription problem occurs when all thieves go into sleep (i.e. $T = 0$) while at least one queue is non-empty. The under-subscription problem degrades the scheduler's performance since the scheduler does not fully exploit the available parallelism. With Lemma 1, we show that our scheduler does not have the under-subscription problem:

Lemma 2. *Our work-stealing scheduler always has*

$$0 < T \quad \textit{if} \quad 0 < Q < W$$

Proof. In our scheduler a worker is active if its queue is non-empty (Algorithm 4):

$$Q \leq A$$

where A is the number of active workers, and by Lemma 1:

$$T \geq 1 \quad \text{if} \quad 0 < A < W$$

Combining these two inequalities, we have:

$$0 < T \quad \text{if} \quad 0 < Q \leq A < W$$

□

Lemma 2 guarantees at least one thief exists when there is a non-empty queue, which prevents the under-subscription problem. Lemma 2 also enables us to offload the task notification from active workers to thieves. In our scheduler workers do not need to notify waiting workers when spawning new tasks. Instead, a thief will notify a waiting worker when the steal succeeds and it is the last thief that decrements the `num_thieves` (Algorithm 5: line 5:6). This allows active workers to quickly add new tasks without being stalled by the notification.

Our scheduling method not only prevents the under-subscription problem but can also mitigate the over-subscription problem. The over-subscription problem means the number of thieves is greater than the number of available tasks. Mitigating the over-subscription problem is very important for two reasons: First, excessive thieves will cause substantial resource wasted on failed steals if they persist for a long time. Second, excessive thieves and active workers might contend for resource, which can result in inefficient resource utilization. We now show the scheduler will put most thieves into sleep within a time bound if they fail to steal any task.

Definition Assume there exists more than one thief. We call these thieves a **group** and a thief leaves the group if it goes into sleep (Algorithm 5: line 33) or successfully steals a task (Algorithm 6: line 10).

Given a group, we prove that only one thief exists in the group after a certain time. This implies most thieves will go into sleep when there are no sufficient tasks. In the following proof, we assume the master queue is empty since thieves will check the master queue before going to sleep. For better description, we denote the constants `steal_bound` and `yield_bound` in Algorithm 6 as α and β, respectively.

Lemma 3. *Given a group of thieves, only one thief in the group exists after $O((\alpha + \beta) * S + C)$ time, where S is the time to perform a steal and C is a constant.*

Proof. Given a group of thieves, we call the thief that lastly decrements the `num_thieves` (Algorithm 5: line 5 and 29) in this group as the *last thief*. Thieves in a group except the last thief must either (1) become active workers if they successfully steal tasks (Algorithm 5: line 4) or (2) go into sleep (Algorithm 5: line 33) after they decrement the `num_thieves`. Therefore,

eventually only one thief stays in the group when the last thief performs the decrement. Next we analyze the runtime taken by the last thief to do the decrement. There are two cases: the last thief either successfully steals a task (Algorithm 5: line 4) or fails to steal any task (Algorithm 5: line 29). For the first case, the runtime is bounded by $O((\alpha + \beta) * S)$ where S is the time of conducting one steal and $(\alpha + \beta)$ is the maximum number of steals that can be attempted. For the second case, the last thief will go through following steps:

1. Perform $(\alpha + \beta)$ steals (Algorithm 5: line 4).
2. Prepare for sleep (Algorithm 5: line 10).
3. Check the master queue (Algorithm 5: line 11) and the scheduler status (Algorithm 5: line 23).

Because steps 2 and 3 are simple routines, we use a constant C to denote the maximal total runtime took by these two steps. Then the runtime of the second case is bounded by $O((\alpha+\beta)*S+C)$. Therefore, the runtime for the last thief to perform the decrement will be bounded by $O((\alpha+\beta)*S+C)$. □

To sum up, we proved our scheduler can prevent the under-subscription problem (Lemma 2) and effectively mitigate the over-subscription problem (Lemma 3). Our scheduling algorithm is simple and efficient in balancing working threads on top of available task parallelism. We will demonstrate the practical performance in the experiment results.

3.3 Evaluation

We evaluated our scheduler using a set of micro-benchmarks and a timing analyzer for VLSI systems. We compare our scheduler with two approaches: the ABP method [16], and the MBWS which is modified from the BWS of Ding et al. [15]. For fair comparison, we implement all scheduling methods in Cpp-Taskflow. We briefly summarize our implementation of ABP and MBWS: ABP lets thieves repeatedly steal until they succeed, and thieves will invoke `yield` system call every time before attempting a steal. BWS [15] introduces two methods to enhance ABP's resource utilization: (1) BWS modifies the OS kernel so that workers can query the running status of others

and yield their cores directly to others. (2) BWS uses two counters, a wake-up and a steal counter, to make thieves wake up two sleeping workers for busy workers and limit the number of steals a thief can attempt. We modify BWS as follows: First, we do not modify the OS kernel as we aim for a portable solution that does not introduce system-specific hard code. To compensate for this, we associate each worker with a status flag which is set by the owner to inform its current status, and thieves do not yield their cores. Second, we implement the modified counter-based approach in the `explore_task` function. As multiple thieves can concurrently modify the wake-up counter, we use atomic compare-and-swap operation to decrement the wake-up counter. A deficiency of BWS is that all thieves could be sleeping while the parallelism changes. To resolve this problem, BWS has to keep one watchdog worker which never goes into sleep to prevent missing parallelism. We also implemented this mechanism in MBWS by having a thief continue to steal if it is the last one that decrements the `num_thieves`. Notice that the modified BWS may not be reflective of the true implementation but it provides a good reference to implement the wake-up-two heuristic.

We conducted all experiments on a machine with two Intel Xeon Gold 6138 processors (2 NUMA nodes) and 256 GB memory. Each processor has 20 cores with 2 threads per core. The OS is Ubuntu 19.04 and the compiler is GCC 8.3.0. We compile all source code with the optimization flag `O2` and C++ 17 standard flag (`-std=c++17`). To reduce the impact of thread migration, we use the system command `taskset` to bond the threads to a set of cores, and we split the threads evenly on the two processors. The `steal_bound` is set to $2 * (\text{number of workers} + 1)$ and the `yield_bound` is 100. For MBWS we adopt 64 as the `SleepThreshold`, which is the same as the experiment setting in [15]. We report the results measured by Linux profiling utility `perf`.

3.3.1 Micro-benchmarks

We select four micro-benchmarks with different kinds of parallelism. This experiment is to provide insight into the schedulers' CPU utilization under various task dependency graphs.

- Linear chain: The task graph is a singly connected list, i.e. each task has

one successor and one predecessor except the first and last tasks. Each task increments a counter by 1. The size of the graph is 8388608.

- Binary tree: The task graph is a binary tree, i.e. each task has one predecessor and two successors except the root and leaf tasks. The task has no computation. The size of the graph is 8388607.
- Graph traversal: We generate a task graph where the dependency is randomly determined, and the number of successors of a node is bounded by a given value. Each task sets a Boolean variable to true to indicate the associated node is visited. The size of the graph is 4000000.
- Matrix multiplication: Given three matrices (2-D array with size 2048x2048), we create a task graph to perform the matrix multiplication. The task graph has two levels: (1) In the first level each task initializes the elements in a row of a matrix. (2) In the second level each task computes a row in the resulting matrix. Tasks in the same level are independent of each other and we create an empty task to synchronize the first level before starting the second level.

In this experiment, we vary the number of cores among 1, 4, 8, 12, 16, 20, 24, 28, 32, 36 and 40. We first compare the schedulers' performance and CPU utilization in all four cases. Then we vary the task granularity of two benchmarks with irregular and regular dependency to observe their performance, CPU utilization and energy consumptions. For each scheduler we report the average value of ten runs on each benchmark (the command is `perf stat -r`).

Figures 3.2 and 3.3 show the runtime and CPU utilization of each benchmark, respectively. For the linear chain, the runtime does not decrease when adding more cores. This is expected as the linear chain has no parallelism at all and one core should suffice for the execution. The CPU utilization of ABP increases along with the number of cores while both MBWS and ours remain nearly uninfluenced. In fact, the CPU utilizations of MBWS and ours stay around 2.0 and 1.2 from 4 to 40 cores respectively.

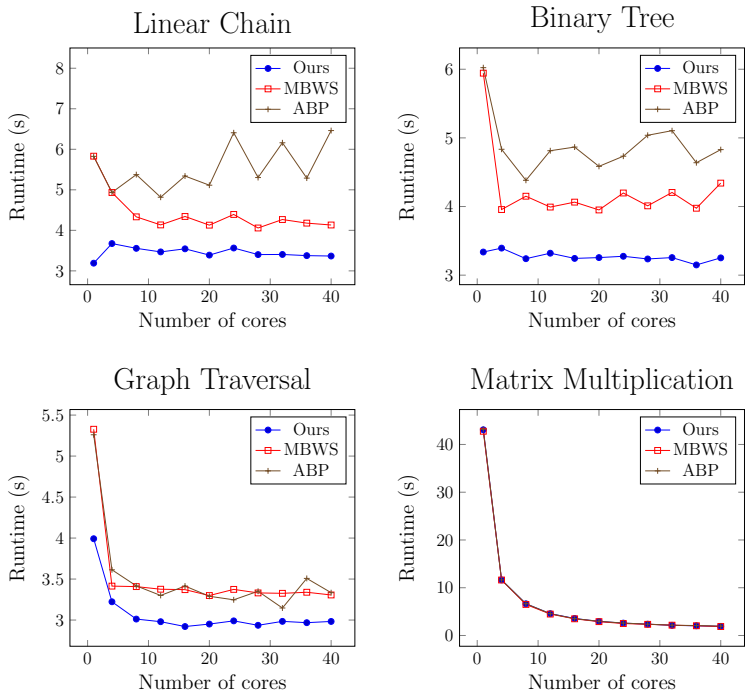


Figure 3.2: Runtime comparisons between ours, MBWS, and ABP on micro-benchmarks.

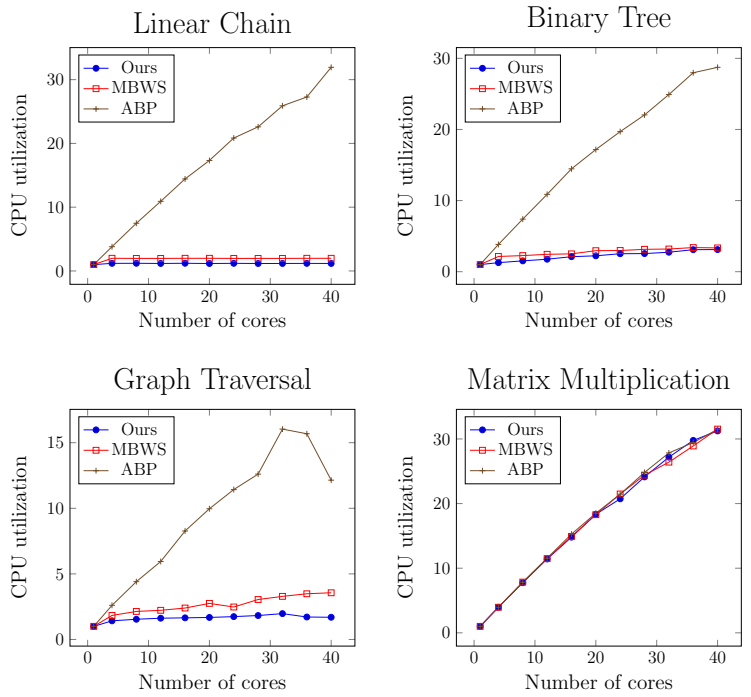


Figure 3.3: CPU utilization comparisons between ours, MBWS, and ABP on micro-benchmarks.

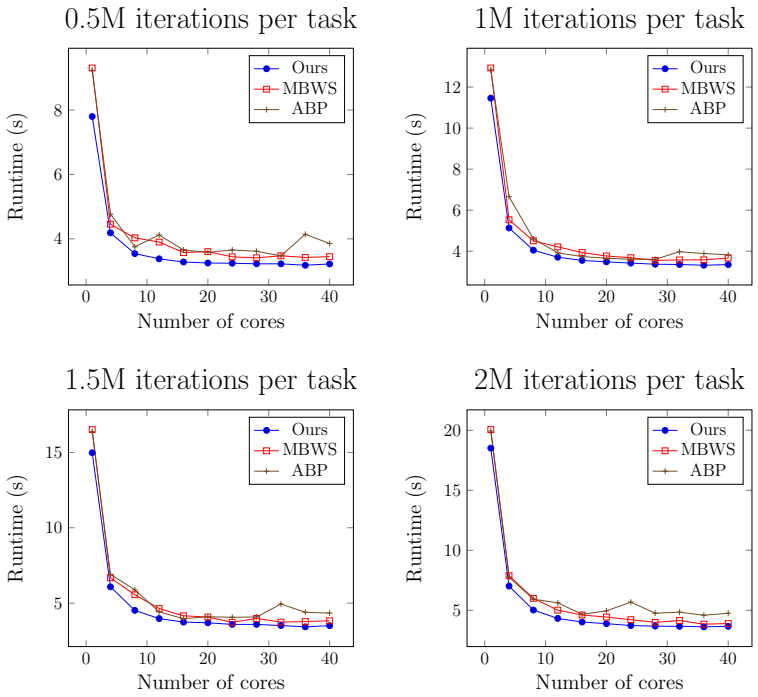


Figure 3.4: Runtime comparisons between different task granularities (number of iterations).

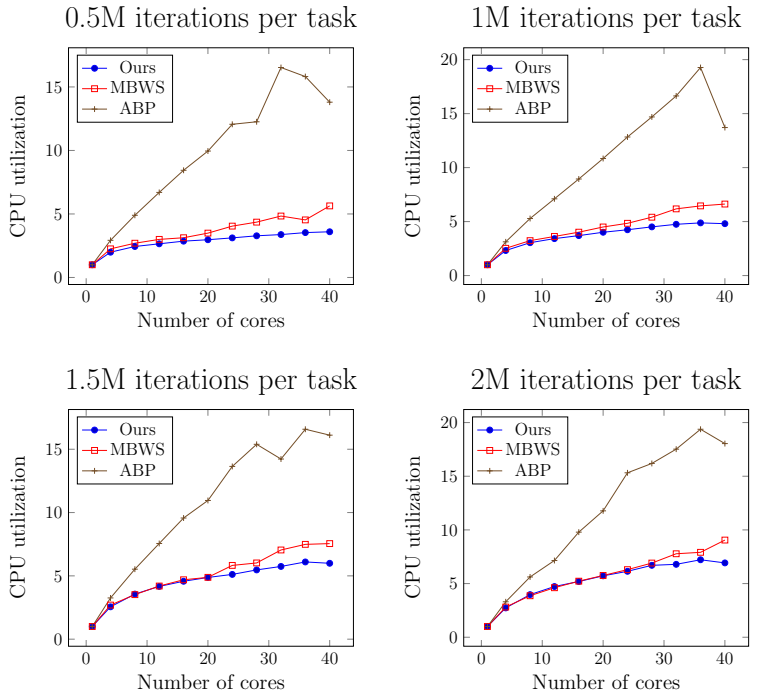


Figure 3.5: CPU utilization comparisons between different task granularities (number of iterations).

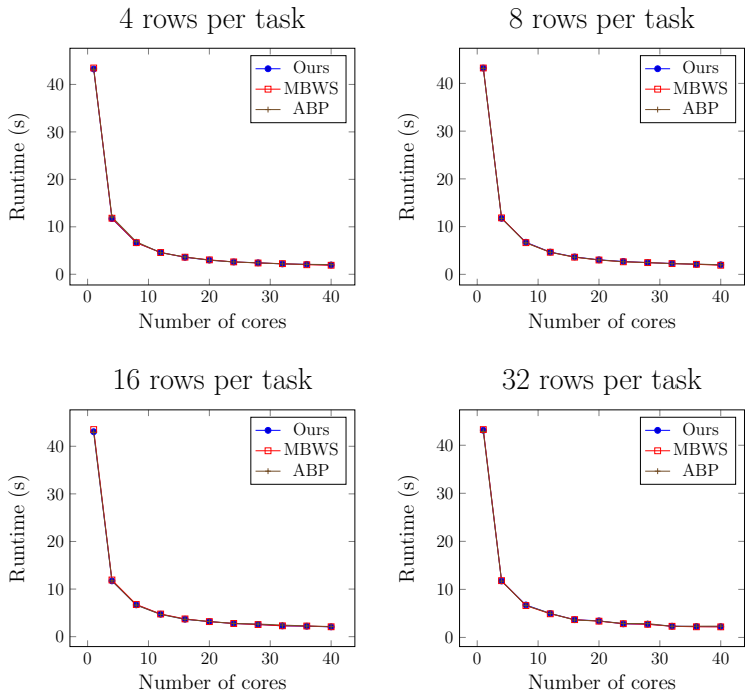


Figure 3.6: Runtime comparisons between different task granularities (number of rows per task).

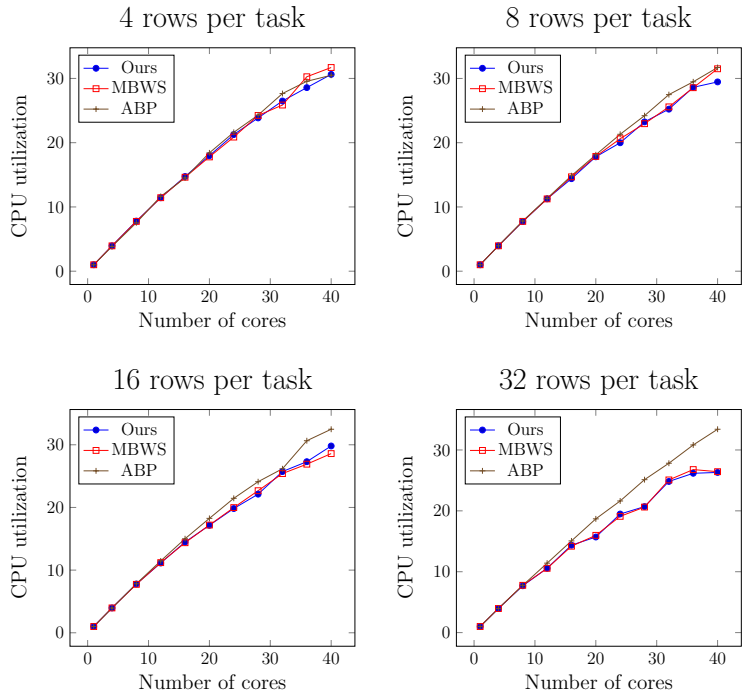


Figure 3.7: CPU utilization comparisons between different task granularities (number of rows per task).

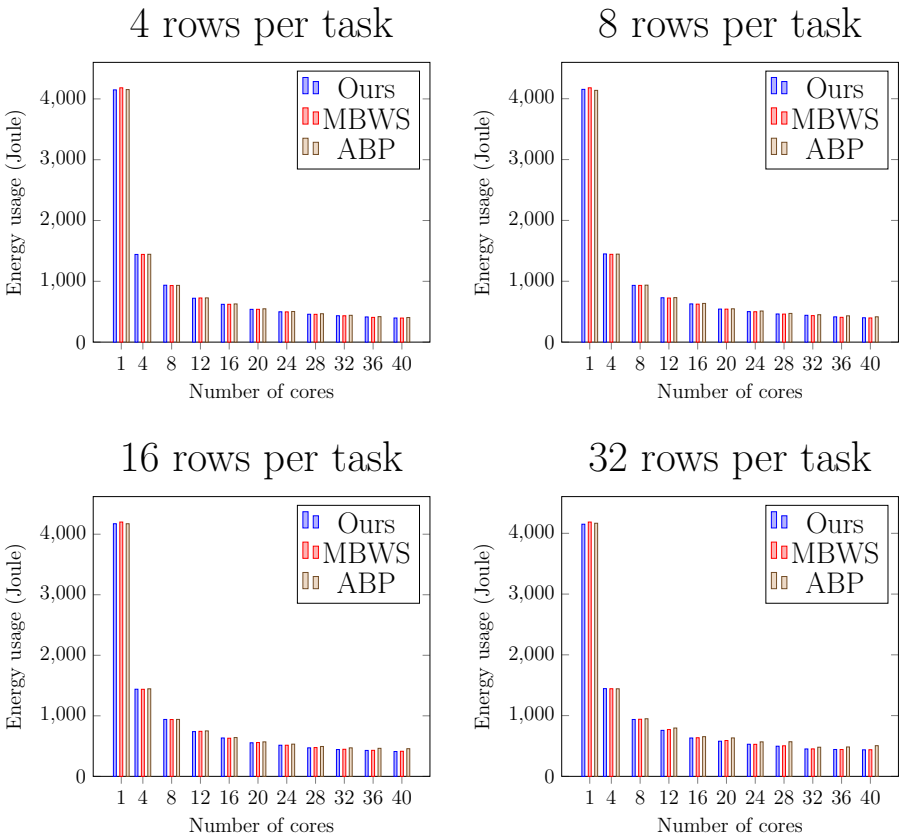


Figure 3.8: Energy usage of ours, MBWS, and ABP on matrix multiplication with different numbers of rows per task.

This example shows that although a thief can possibly yield its core to other workers before stealing, keeping thieves awake can still incur high CPU utilization. For the binary tree and graph traversal, the runtimes of all schedulers drop to a stable point after 4 cores. Adding more cores does not improve the performance as the workload of their tasks is very small and a worker might quickly carry out all tasks in its queue before thieves discover them. ABP has the highest CPU utilization among all schedulers and ours is the lowest in both cases. The CPU utilization of ABP also grows more rapidly than others in these two cases.

For the matrix multiplication, which has better scalability than the previous three cases, the runtimes of all schedulers are very close and their CPU utilizations exhibit similar growth trends. There are two main reasons accounting for this: (1) In the matrix multiplication, intra-level tasks are independent of each other and those tasks have nearly equal workload. (2) The multiplication is compute-intensive and thus the runtime is dominated

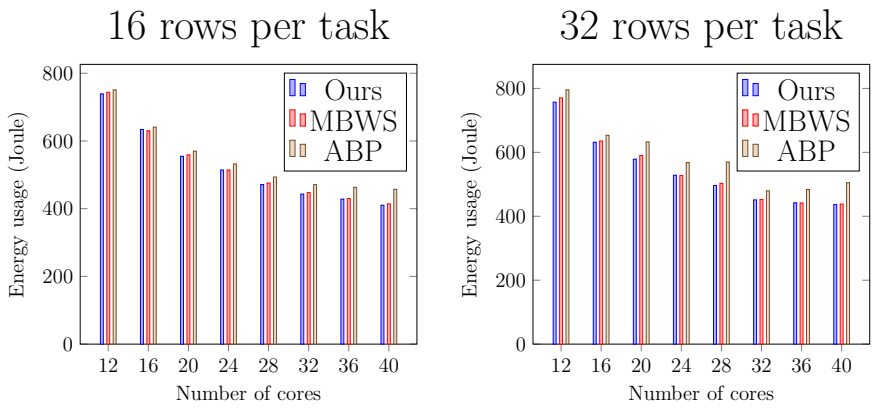


Figure 3.9: Energy usage of ours, MBWS, and ABP on matrix multiplication with 16 and 32 rows per task.

by the computation rather than the scheduling overhead.

Next, we further evaluate these schedulers by tuning the task granularity. In this experiment we select two benchmarks: graph traversal and matrix multiplication, The former has irregular dependency between tasks, while in the latter tasks in the same layer are independent. For each task in the graph traversal benchmark, we deliberately add a for-loop which iteratively performs division, and we change the loop's number of iterations to adjust the tasks' workload. The numbers of iterations tested in this experiment are 5×10^5, 1×10^6, 1.5×10^6 and 2×10^6. Figure 3.4 and 3.5 are the runtime and CPU utilization of all schedulers under different numbers of iterations, respectively. In general, in all scenarios adding more cores can improve all schedulers' performance. The CPU utilizations of all schedulers increase along with the number of iterations. The reason for this could be that with more iterations a worker will take longer to execute a single task, and this will give more time for the idle workers to steal tasks. We observe that ABP has a noticeable fluctuation in CPU utilization, while the CPU utilizations of ours and MBWS increase steadily. Under the same number of iterations, ABP has the highest CPU utilization; nevertheless the runtime of ABP is higher than others. On the contrary, our scheduler can get better performance than both ABP and MBWS with less CPU utilization.

Finally for the matrix multiplication benchmark, we delegate the computations of multiple rows in the resulting matrix to each task to vary the workload. Figure 3.6 and 3.7 are the runtime and CPU utilization of all schedulers under different numbers of rows, respectively. Regarding the performance, all

schedulers have very similar runtime and scalability in all workloads. For the CPU utilization, we found that ABP has higher CPU utilization when a task is given more rows, especially when the number of available cores increases. Because the total number of tasks is inversely proportional to the number of rows in a task, this result shows that ABP can cause excessive CPU usage when there is no sufficient task. Over-subscription of the CPU resource can lead to inefficient energy use which is shown in Figure 3.8. To clearly demonstrate the difference, we specifically single out the energy consumptions of 16 and 32 rows with using more than 8 cores in Figure 3.9, which shows that ABP consumes more energy than the others. For instance, ABP consumes 13.2% and 13.5% more energy than MBWS and ours respectively, when there are 40 cores and a task is assigned 32 rows.

To conclude, our scheduler can deliver comparable performance to others under various task dependency graphs and is more efficient in CPU utilization. The latter can contribute a lot to the energy efficiency and throughput of co-running multithreaded applications, which will be demonstrated later in a large-scale workload.

3.3.2 VLSI Timing Analysis

Next we evaluate the schedulers on a real-world application: VLSI timing analyzer. Static timing analysis (STA) plays a critical role in the circuit design flow. For a circuit to function correctly, its timing behavior must meet all requirements under different design constraints and environment settings. Thus, circuit designers have to apply STA to verify the circuit's timing behavior during different stages in the design flow. STA calculates the timing-related information by propagating through the gates in a circuit, and this workload can be naturally described using a task dependency graph. In this experiment, we use these schedulers to execute the task graph built in OpenTimer [7], an open-source VLSI timer. We randomly generate a set of operations which incrementally modify a given circuit and then perform STA to update the timing. We use the circuits from TAU 2015 timing contest [33] and the statistics of the circuits are listed in Table 3.1. For each circuit we ran OpenTimer five times and report the average runtime and CPU utilization recorded by `perf`. Figure 3.10 and 3.11 show the runtime and CPU utilization

Table 3.1: Statistics of circuits

Circuit	# of gates (K)	# of nets (K)	# of operations
c6288	1.7	1.7	80800
c7552	1.1	1.4	80800
tv80	5.3	5.3	51000
mgc_matrix	171.3	174.5	10100
b19	255.3	255.3	10100
vga_lcd	139.5	139.6	30300

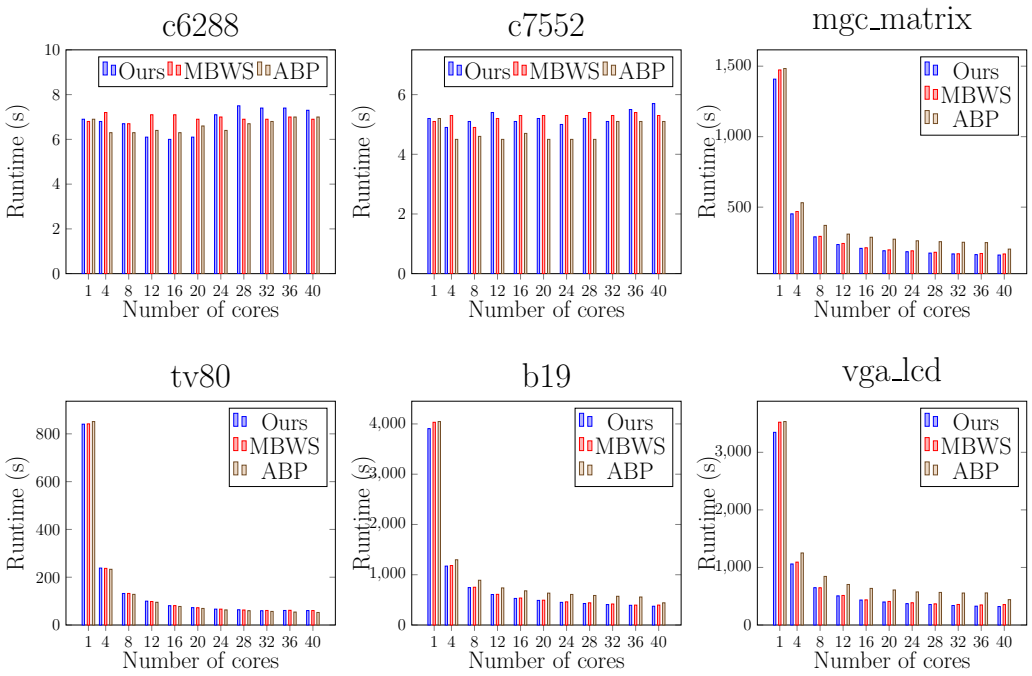


Figure 3.10: Runtime comparisons between ours, MBWS, and ABP on OpenTimer.

of each circuit respectively.

We categorize the circuits into different groups based on their sizes and discuss the results. For those small circuits c6288 and c7552, their runtimes do not scale with the number of cores. The CPU utilizations of all schedulers on these two circuits increase along with the number of cores, and ABP has the highest CPU utilization followed by the MBWS and ours is the smallest. Next for the medium size circuit tv80, the runtimes of all schedulers decrease after adding more cores. ABP is faster than others except at single core and the runtimes at 40 cores are 60.4 (ours), 60.8 (MBWS) and 52.5 (ABP), respectively. We attribute this to the overhead of notifying workers. Both

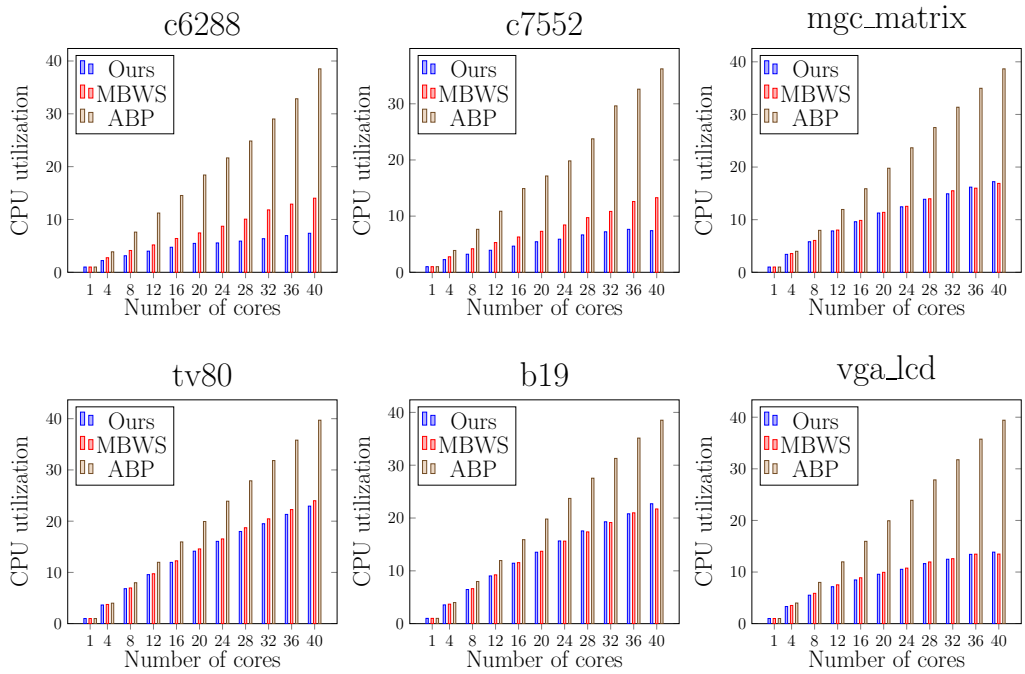


Figure 3.11: CPU utilization comparisons between ours, MBWS, and ABP on OpenTimer.

ours and MBWS will put thieves into sleep and notify them when tasks present, while in ABP all thieves are kept busy in waiting for tasks. In terms of the CPU utilization, ABP is still the highest and ours and MBWS are very close. Lastly, for those large circuits with over 100,000 gates: mgc_matrix, b19, and vga_lcd, the performance scales with the number of cores in all schedulers. When using multiple cores, ABP is slower than others even though ABP's CPU utilization remains the highest. Take the largest circuit b19 with 40 cores as an example; the runtime of ours is 5% and 15% less than MBWS and ABP, respectively, and the CPU utilizations are 22.7 (ours), 21.7 (MBWS) and 38.5 (ABP). This experiment shows that our scheduler has competitive performance, and can utilize the CPU resource in a reasonable way under a large-scale workload.

Next we demonstrate the energy usage and power consumption of each scheduler with OpenTimer. Intel has provided users the Running Average Power Limit (RAPL) [34] interface for power management on recent processors. We use `perf`, which can access the RAPL interface, to measure the energy consumed by two packages (2 NUMA nodes) during the execution (the command is `perf stat -e power/energy-pkg/ -a`), and we let `perf`

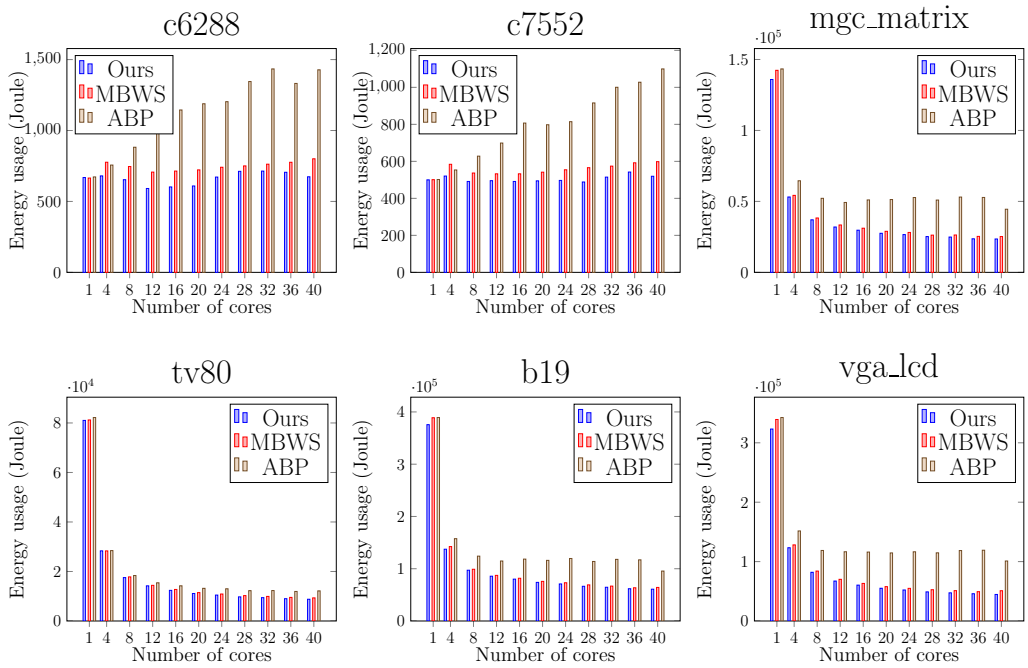


Figure 3.12: Energy usage of ours, MBWS, and ABP on OpenTimer.

report the average value of five runs.

Figure 3.12 is the average energy usage reported by `perf`, and we divide the energy usage by the runtime to derive the power consumption which is shown in Figure 3.13. For energy usage, ABP is the highest in all cases. MBWS is very close to ours with ours performing slightly better in most cases. For small circuits like c6288 and c7552, the energy usage of ABP increases along with the number of cores even the performance does not scale. For example, in c6288 the energy usage of ABP is 2x of ours at 40 cores but the runtime of ABP is only 8% less than ours. For other circuits, the energy usage of all schedulers decreases after adding more cores as those circuits have good scalability. However, ABP's energy usage is still much higher than others. For example, in the largest circuit b19 ABP's energy usage is 1.57x of ours and 1.48x of MBWS when using 40 cores. Next for the power consumption, ABP's power consumption increases along with the core numbers in all cases. The result shows that thieves can still consume substantial power even making them yield frequently. This is especially evident in small circuits c6288 and c7552 where ABP's power consumption doubles when the number of cores increases from 1 to 40. In contrast, the power consumptions of ours and MBWS do not show substantial growth after adding more cores in these

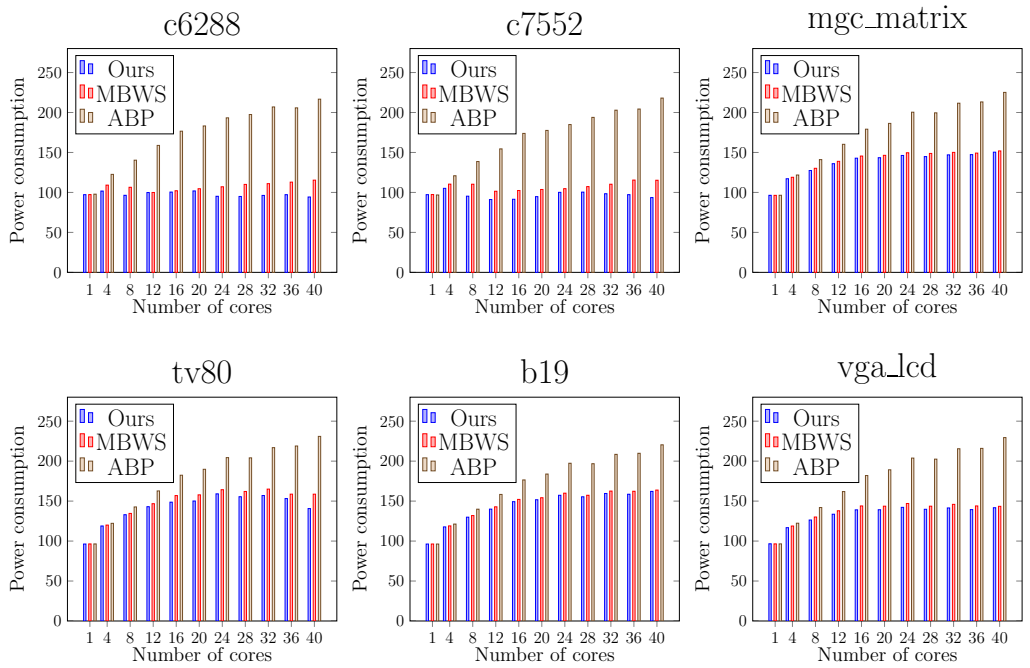


Figure 3.13: Power consumption (energy/runtime) of ours, MBWS, and ABP on OpenTimer.

two circuits. For larger circuits, the power consumption of all schedulers increases along with the core numbers, and again ABP's grows faster than others. In the largest circuit b19 with 40 cores, the power consumptions of ours and MBWS are 26% and 25% less than ABP, respectively.

In the last experiment, we measure the effect of co-running multiple OpenTimers. This experiment is to simulate real working environment which is typically shared by multiple users such as servers or cloud computing platforms. In those environments users can run multithreaded applications concurrently, and applications might request computing resources more than their actual parallelism.

In this experiment, we run multiple OpenTimers simultaneously on the same circuit and every timer can use all the cores (40 on our machine). The number of OpenTimers in the co-runs ranges from 2 to 8 and we use the `time` command to measure the runtime (wall clock time) of each timer. We repeat each co-run five times and use the average as the throughput. For each scheduler, we take the runtime of its solo-run as the baseline and compute the throughput using the weighted-speedup method [15] [35]. The weighted-speedup method sums the speedup of each process in the co-runs, where the

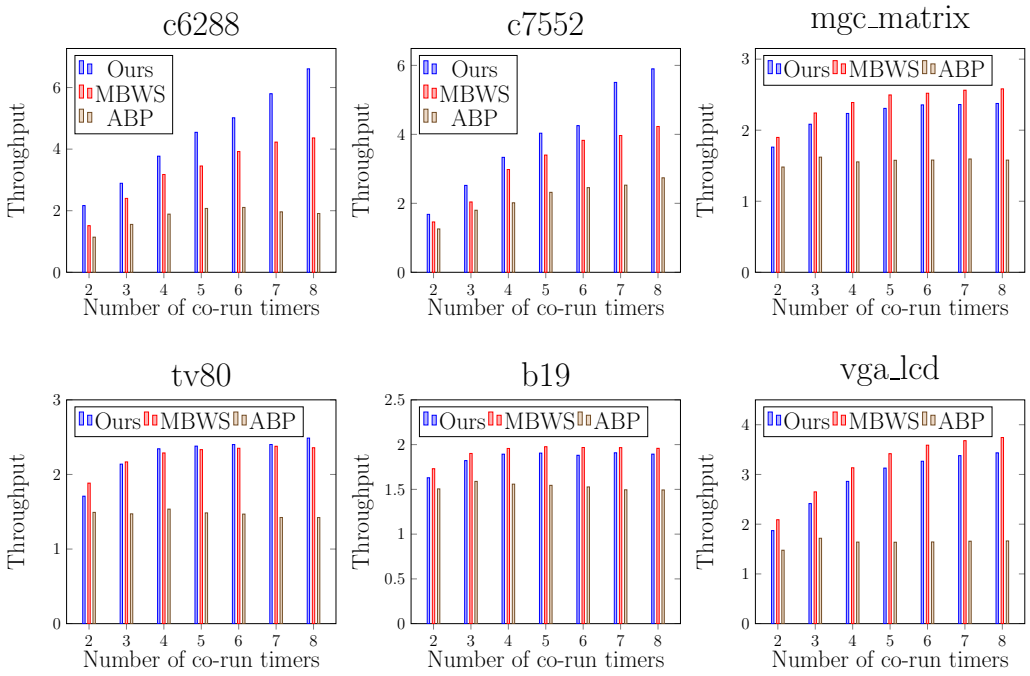


Figure 3.14: Throughput comparisons of co-running multiple OpenTimers.

speedup is defined as $T_{baseline}/T_{per\_proc}$. Figure 3.14 shows the throughputs of all schedulers on different circuits. ABP has the lowest throughput in all co-runs regardless the circuit size. For example, when co-running 8 OpenTimers, the throughputs of ABP are 1.49 and 2.74 on b19 and c7552, respectively, while ours is 1.89 and 5.9 and MBWS is 1.95 and 4.2. MBWS and ours have similar throughput except at c6288 and c7552 where ours is much higher.

3.4 Conclusion

In this chapter, we have introduced a work-stealing scheduler for executing task dependency graph. We have designed an efficient worker management method that adaptively adjusts the number of thieves by tracking the number of workers that are executing tasks. This method not only effectively prevents resource from being underutilized but also mitigates resource waste. We have evaluated the scheduler on a set of micro-benchmarks and a VLSI timing analyzer. The results show our scheduler achieved comparable performance to existing approaches, with effective resource utilization and good energy efficiency.

CHAPTER 4

TASK GRAPH COMPOSITION AND CONDITIONALS

4.1 Introduction

The key to make developers productive in writing software is *composability*. We use libraries written by other developers to compose a large program, or we decompose a job into smaller pieces to tame the complexity in software development. Composability is especially important in developing fast market-expanding applications such as high-performance machine learning, data analytics, and parallel simulation engines [36]. These applications exhibit both *regular* and *irregular* compute patterns, and are often combined with other functions to compose large software that will be deployed on a multicore machine or a distributed cloud [37, 38]. However, *composable parallel processing* is rarely addressed as the first-class concept by existing parallel programming libraries [39]. Many libraries were designed to solve a single hard problem as fast as possible, leaving users to decide composition with their own practice. This can create a lot of pain and data engineering tasks for developers of different teams to collaborate on a large parallel application. Some common problems include confusing API mix-uses, unwanted coupling layers, error-prone dependency wrappers, inconsistent threading models, and suboptimal scheduling results.

The traditional interface for program decomposition is *function call*. Developers break down a large *sequential* program into a specific set of tasks each wrapped in a function call with clear definition of data exchange. These function calls are often *modular* and *reusable* to make the codebase maintainable and readable. However, *composable parallel programming* is much more challenging. Modern parallel workloads typically combine a broad mix of algorithms, functions, and libraries. Each library manages its own threads and task execution, making it difficult to perform optimization across dif-

ferent libraries. When coupling these software pieces together, we need to tackle the *dependencies* both inside and outside the libraries. Some libraries are already parallel and they are being used by other parallel programs and so forth. There are many practical issues to consider such as thread management, resource over-subscription, and concurrency controls. As a result, the lack of a clear and unified interface has a serious impact on performance, even when individual libraries are heavily optimized.

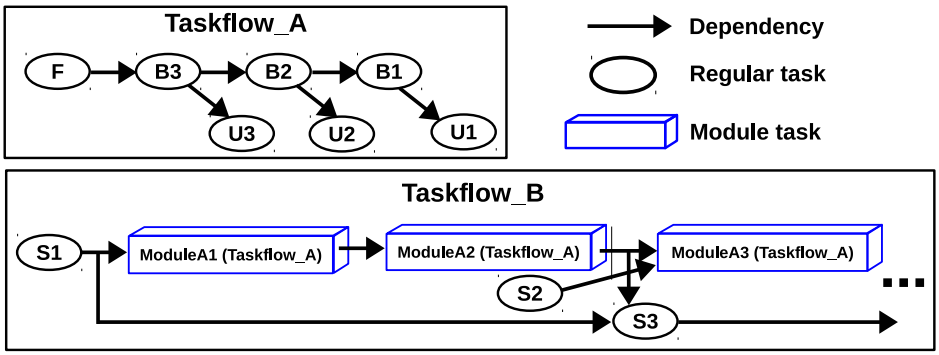


Figure 4.1: Using our composable task dependency graph to describe a parallel neural network training workload. Taskflow object A represents one training iteration and is used to compose taskflow object B for the entire training procedure.

In this chapter, we largely enhanced Cpp-Taskflow's capability with three key design changes: (1) separating the task dependency graph and execution kernel, (2) making the task dependency graph reusable and composable and (3) enabling execution flow control via conditional tasking. In our model, a *taskflow object* consists of a *composable task dependency graph* and user-friendly APIs to facilitate the creation of *modular* and *reusable* parallel compute patterns and libraries. These libraries can recursively compose large and complex parallel computations on a single machine, taking advantage of multicore processing while sharing thread resources to minimize overhead. Figure 4.1 gives an example of using taskflow objects to describe a parallel training algorithm of a deep neural network (DNN). Taskflow object A represents a training pipeline. Taskflow object B is composed of multiple As and other tasks to complete the training procedure. Also, users can easily couple B with other parallel computations. There is no redundancy from the programmability standpoint. And last but not least, we design a new tasking: *conditional tasking* to allow users to control a task graph's execution

flow at runtime. We summarize our contributions as follows:

- **A new composable parallel programming model**. We developed a composable task interface to enable efficient composition of parallel workloads. The composable programming model lets users quickly describe a large parallel program through composition of modular and reusable task graphs that embed both regular and irregular compute patterns. The program runs on a multicore machine with automatic scheduling optimization across different layers of composed tasks.

- **A unified task composition interface**. We developed a unified task graph construction interface that can capture a diverse set of tasks from single sequential functions to large parallel dependent tasks or even out-of-context executions such as third-party calls and process forks. The unified interface empowers developers with both explicit and implicit task graph composition to explore cross-layer optimizations of their parallel workloads.

- **A simple and efficient composition API**. We developed a user-friendly API to describe task dependency graph composition using modern C++17 syntax. Users can fully take advantage of the rich features of our engine together with robust standard C++ libraries to productively compose many parallel applications. Our library effectively separates users from low-level difficult concurrency details and offers *transparent* scaling to many cores and future hardware generation.

- **A new tasking for flexible execution scheme**. We develop conditional tasking to let users control the execution flow at runtime. Different from other taskings that every task must be executed once, conditional tasking enables users to selectively execute part of a task graph, and users can iterate tasks multiple times by embedding cycles in the task graph. This means the task graph does not need to be *acyclic* anymore, and therefore the task dependency graph can formulate generic control flow.

4.2 New Task Dependency Graph

We introduce a new composable task interface, the *tf :: Taskflow* class, which is the main gateway to create a composable task dependency graph. The `tf::Taskflow` class inherits all the task construction methods from original Cpp-Taskflow. Listing 4.1 demonstrates how to create a task dependency graph with two tasks A and B where B runs after A and B spawns a new task B1 during runtime. We enhance Cpp-Taskflow's usability by separating the task dependency graph from executor. Users now have full control over their task dependency graphs but are also responsible for their lifetime.

```
tf::Taskflow taskflow;

// Add a static task
auto taskA = taskflow.emplace([](){
  std::cout << "Task A\n";
});

// Dynamic tasking
auto taskB = taskflow.emplace([](auto &subflow){
  std::cout << "Task B\n";
  subflow.emplace([](){ std::cout << "Task B1\n"; });
});

taskA.precede(taskB);
```

Listing 4.1: Create a task dependency graph of two dependent static tasks and one dynamic task.

A significant change we made is the decoupling of executor and task graph. We define `tf::Executor` class that has a rich set of methods to run a task dependency graph. A task dependency graph can be run by an executor multiple times in arbitrary order. Users can also give a predicate to specify the stopping criteria. Listing 4.2 demonstrates a set of common methods to run a task dependency graph. Line 1:3 creates a taskflow object and adds some tasks. Line 6 creates an executor. An executor is nothing but a pluggable scheduler to dispatch tasks to threads in a shared pool. The simplest way is to execute a task dependency graph only once via the `run`

method (line 9). Alternatively, users can call `run_n` to run a task dependency graph multiple times (line 13). The bottommost call is `run_until` (line 19), which keeps running until the predicate becomes true. All methods accept a callable object as a callback after the task execution completes. To enable more asynchronous control, each of these methods returns a `std::future` for users to inspect the execution status or incorporate non-blocking program flow. It should be noticed that running a task dependency graph multiple times exhibits the most basic composability, by which the same graph is encapsulated in a linear chain of tasks.

```
tf::Taskflow taskflow;

// Add some tasks ...

// Create an executor with 4 threads
tf::Executor executor {4};

// Run the taskflow object once
auto fut = executor.run(taskflow);
fut.get();

// Run the taskflow object 4 times with a callback
taskflow.run_n(taskflow, 4, [](){
  std::cout << "Finish!\n";
}).get();

// Run the taskflow object with a predicate
int counter {4};
executor.run_until(taskflow,
  [&](){ return --count == 0; }
).get();
```

Listing 4.2: Different ways to execute a task dependency graph.

4.3 Composable Tasking

Task dependency graph composition is one of the most important features we add to Cpp-Taskflow. It allows users to create heavily optimized task dependency graphs and reuse them to compose larger graphs and so on so forth. The `tf::Taskflow` class defines a method `composed_of` to enable composition. Specifically, the caller taskflow object adds a module task of the callee taskflow object. Listing 4.3 shows an example of taskflow object composition. Line 1:10 creates a taskflow object with three dependent tasks A1, A2, and A3. Line 12:19 creates another taskflow object with three tasks B1, B2, and B3. Line 22 adds a module task from the first taskflow object and line 25:27 specifies the dependency between tasks. Unlike the `emplace` method that creates a *regular task*, the `composed_of` method creates a *module task* in the graph. A module task is a special task that is aware of which taskflow object to probe during its execution context. We would like to highlight three points of our composition interface. First, there is no copy during the composition, leading to efficient graph sharing and resource utilization. We have strived to resolve many scheduling conflicts due to shared tasks, while providing a high-level execution API to completely separate this low-level controls from users. Second, recursive and nested composition are feasible. A taskflow object can be used to compose multiple taskflow objects and the resulting taskflow object can compose another taskflow object with no restriction. During the composition, the user can add free-standing tasks to the graph to perform computation across different task layers. Adding dependency is extremely easy and flexible through the `precede` method. Finally, the module task works seamlessly with both static and dynamic tasking. This gives users a powerful and unified tasking interface to accomplish large and complex parallel workloads.

```
1  tf::Taskflow fA;
2
3  // Add three tasks
4  auto [A1, A2, A3] = fA.emplace(
5    []() { std::cout << "Task A1\n"; },
6    []() { std::cout << "Task A2\n"; },
7    []() { std::cout << "Task A3\n"; }
8  );
```

```
A3.gather(A1, A2);

tf::Taskflow fB;

// Add three tasks
auto [B1, B2, B3] = fB.emplace(
  []() { std::cout << "Task B1\n"; },
  []() { std::cout << "Task B2\n"; },
  []() { std::cout << "Task B3\n"; }
);

// Compose taskflow object
auto moduleA = fB.composed_of(fA);

// Build dependency between module and regular tasks
B1.precede(moduleA);
B2.precede(moduleA);
moduleA.precede(B3);
```

Listing 4.3: Cpp-Taskflow taskflow object composition code (19 LOC and 167 tokens).

At this point, we are interested in the difference between our composition code and existing libraries. Listing 4.4 is the implementations of Listing 4.3 using TBB flow graph [9]. As shown in the two listings, Cpp-Taskflow has the fewest lines of code and is more readable than the TBB code. To our best knowledge, TBB has no API to directly compose task graphs, so we have to capture the task graph into another task and execute the task graph. This results in longer lines of code and tends to produce bugs if one forgets to execute the task graph. This example clearly shows the conciseness and ease-of-use of the task composition interface in Cpp-Taskflow.

```
using namespace tbb;
using namespace tbb::flow;

graph fA;

```

```
continue_node<continue_msg> A1(fA, []
  (const continue_msg&) {
    std::cout << "Task A1\n";
  }
);
continue_node<continue_msg> A2(fA, []
  (const continue_msg&) {
    std::cout << "Task A2\n";
  }
);
continue_node<continue_msg> A3(fA, []
  (const continue_msg&) {
    std::cout << "Task A3\n";
  }
);

make_edge(A1, A3);
make_edge(A2, A3);

graph fB;

continue_node<continue_msg> B1(fB, []
  (const continue_msg&) {
    std::cout << "Task B1\n";
  }
);
continue_node<continue_msg> B2(fB, []
  (const continue_msg&) {
    std::cout << "Task B2\n";
  }
);
continue_node<continue_msg> B3(fB, []
  (const continue_msg&) {
    std::cout << "Task B3\n";
  }
);
```

```
42 continue_node<continue_msg> moduleA(fB, [&]
43   (const continue_msg&) {
44     A1.try_put(continue_msg());
45     A2.try_put(continue_msg());
46     fA.wait_for_all();
47   }
48 );
49
50 make_edge(B1, moduleA);
51 make_edge(B2, moduleA);
52 make_edge(moduleA, B3);
```

Listing 4.4: TBB hard-coded composition code (48 LOC and 256 tokens).

In addition to the composability, another useful feature is the modularity. Through *inheritance* from `tf::Taskflow` class, users can define their own task dependency graph class as a *single module*. The task dependency graph composition and execution APIs can be directly applied to the customized class as well, obviating the need of an additional wrapper.

With the composability and modularity, a complex design can be decomposed into small components with different parallel patterns. Users can implement and test those patterns individually and combine them in various ways such as nested or concatenated to deliver complex functionality. This can substantially increase programmers' productivity as it enables a structured and efficient way of software engineering.

4.3.1 Unified Task Execution

We modify the execution kernel to enable seamless integration of the reusable and composable task dependency graph with existing task types. To make a task dependency graph reusable, it is necessary to ensure the graph remains unchanged after each execution. During runtime, a task might expand the graph by spawning new nodes to precede the parent node such as dynamic tasking. As a result, in Cpp-Taskflow a task that spawns new tasks will restore its own precedence before scheduling its successor tasks. Letting each task perform the restoration on itself also minimizes the overhead.

Apart from the regular tasks, a task dependency graph can have module tasks through composing other graphs. The execution flow of module task is similar to dynamic tasking except that a module task directly dispatches the composed graph rather than a subflow. A module task will be executed twice:

- First time:
  - The executor first collects the source and sink tasks in the composed graph and lets the sink tasks precede the module task.
  - The executor dispatches the source tasks to execution.
- Second time:
  - The executor removes the successor of sink tasks in composed graph.
  - The executor dispatches the module task's successors to execution.

Figure 4.2 is an example that illustrates scheduling a module task.

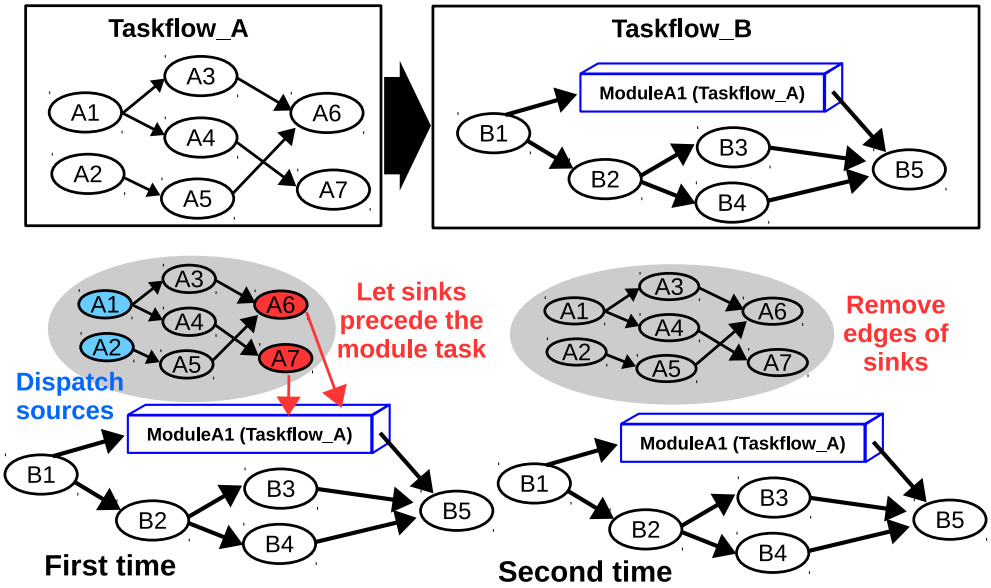


Figure 4.2: An example to illustrate the execution of module task.

4.3.2 Visualize a Task Dependency Graph with Both Regular and Module Tasks

We provide the same API to support visualization of composable task dependency graph to facilitate debugging. A taskflow object can be assigned a name by the `name` method and it has a `dump` method to export its task dependency graph in DOT language [40]. A module task is represented by a *cuboid* to differentiate from the regular tasks. Figure 4.3 shows an example of visualizing composed task dependency graphs.

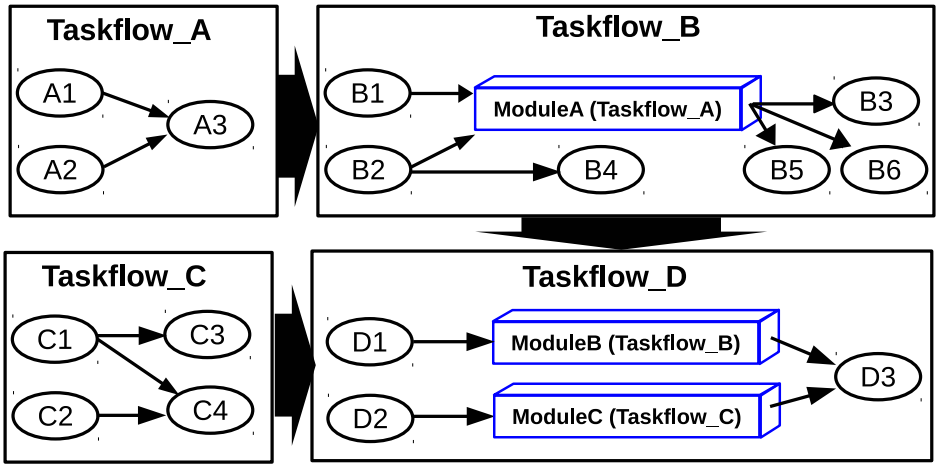


Figure 4.3: Visualization of the task dependency graph D with its regular and module tasks. Note that the arrows between taskflow objects are added deliberately here for clarity.

4.4 Conditional Tasking

In original Cpp-Taskflow, the task graph needs to be **acyclic** and all tasks will be executed **exactly once** in each run. This somehow restricts the expressiveness of Cpp-Taskflow, since a typical program control will have branches like if-else or switch-case statements, and loop constructs such as for-loop, while-loop and so on. Some of these control flows can be realized via dynamic tasking. For example, one can use dynamic tasking to spawn different task graphs at the runtime to mimic the conditional control flow. Repeatedly creating task graph at runtime might incur notable overhead due to the memory allocation. To make the graph model more generic and convenient, we introduce conditional tasking which allows users to build directed

cyclic task graphs and create branches in the execution flow.

Listing 4.5 demonstrates how to use conditional tasking in Cpp-Taskflow, and Figure 4.4 is the resulting task graph. In this example, we create a graph with five static tasks, A, B, C, D and E, and two condition tasks, `cond1` and `cond2`. This graph contains a loop formed by tasks B and `cond1` and a branch consisting of tasks `cond2`, D and E. A condition task is different from other tasks in that it has to return an index of successor task, and the corresponding successor task will be directly scheduled to execution. For instance, if task `cond1` returns 0, then task B will be put into execution. In Figure 4.4, the numbers on the dotted edges of condition tasks are the successor indices. With conditional tasking, users can create task graphs with complex control flows such as iterating subgraph multiple times, switching execution paths at runtime, or combining both, which makes Cpp-Taskflow more powerful and expressive.

```
tf::Taskflow flow;

auto A = flow.emplace([](){ std::cout << "TaskA\n"; });
auto B = flow.emplace([](){ std::cout << "TaskB\n"; });
auto C = flow.emplace([](){ std::cout << "TaskC\n"; });
auto D = flow.emplace([](){ std::cout << "TaskC\n"; });
auto E = flow.emplace([](){ std::cout << "TaskE\n"; });

srand(1);
auto cond1 = flow.emplace([](){ return rand()%2; });

// Create a loop
A.precede(B);
B.precede(cond1);
cond1.precede(B, C);

```

```
17  auto cond2 = flow.emplace([](){ return rand()%2; });
18
19  // Create a branch
20  C.precede(cond2);
21  cond2.precede(D, E);
```

Listing 4.5: An example to demonstrate conditional tasking

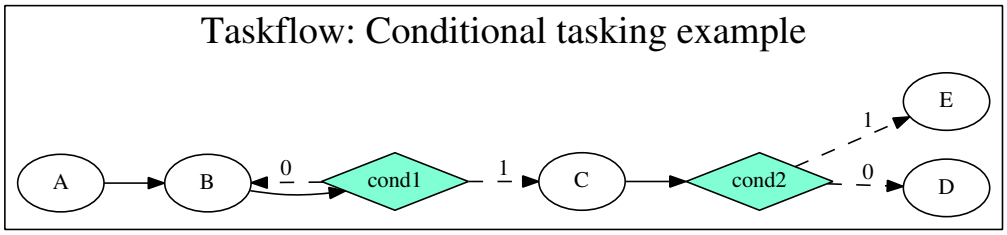


Figure 4.4: The task dependency graph of Listing 4.5.

However, integrating conditional tasking with other taskings in Cpp-Taskflow is a challenging task. There are three problems needed to be solved:

Problem 1: How to define a task's dependency?

Problem 2: How to determine if a task graph has finished execution?

Problem 3: How to efficiently implement conditional tasking?

For the first problem, the original task scheduling rule states that a task is ready for execution when its dependencies are fulfilled. However, a task might be preceded by its successor task in conditional tasking, and the rule will fail to work in this case. For example, in Figure 4.4, task B is preceded by task `cond1`, and task B will never be ready according to the rule. To solve the first problem, we categorize the dependency into two classes: we define the dependency between a condition task and its successors as *weak dependency*, and others are *strong dependency*. Then we introduce new task scheduling rules with strong and weak dependency as follows:

Rule 1: We define the tasks without both weak and strong dependency in a task graph as source tasks. The source tasks are the starting point of a task graph execution.

Rule 2: At runtime, for those tasks other than condition tasks, their successors are scheduled to execution when their strong dependency is fulfilled.

Rule 3: For condition tasks, the specified successor task will be directly put into execution regardless of the strong dependency.

Task graphs with or without condition tasks can be correctly scheduled under these three rules.

For the second problem, in original Cpp-Taskflow, since all tasks will be executed exactly once, a task graph is deemed completed when all sink tasks (tasks without successors) have finished, and we can easily implement this by using a counter to track the number of finished sink tasks during runtime. At runtime, whenever a sink task finishes execution, the counter will be decremented by one and the graph is deemed finished when the counter becomes zero. However, this no longer holds after integrating conditional tasking. With the conditional tasking, a task might not be executed or be executed multiple times, which makes the above sink task counting method invalid.

To solve the second problem, we observe two critical properties about the task graph execution.

Property 1: The tasks of different task graphs will not exist in a worker's private queue at the same time.

Property 2: At least one task of the graph is kept by the executor anytime during the task graph execution.

With these two properties, we can determine that a task graph is completed if it has no task kept in any queue and worker. Based on this observation, we associate each topology with a counter to track the number of tasks held by the executor. The counter is updated in following two cases:

1. Whenever a task is ready for execution and added into a queue, the counter is incremented by one.

2. Whenever a worker finishes a task derived from a queue, the worker decrements the counter by one.

When the counter becomes zero, the graph has no outstanding tasks and is deemed finished. The last worker that updates the counter shall set up the `future` object to mark the graph status as completion. This counting method works well when task graph is not modified during execution. But one problem of this new method is to handle dynamic tasking under joined mode and composed tasks where both will spawn a task graph during runtime. Originally, Cpp-Taskflow handles the spawned graph by having its sink tasks precede the parent task to ensure the spawned graph will finish before scheduling the parent task's successor tasks. Figure 4.5 shows an example of joined dynamic tasking. In Figure 4.5, task B spawns a new task graph consisting of task B1, B2 and B3. Both task B2 and B3 will be set to precede task B (parent task) so that task D will guarantee to be executed after the spawned graph finishes. The red edges are added by the executor to ensure correct execution order between the spawned task graph and the parent task's successors. However, when the spawned graph contains condition

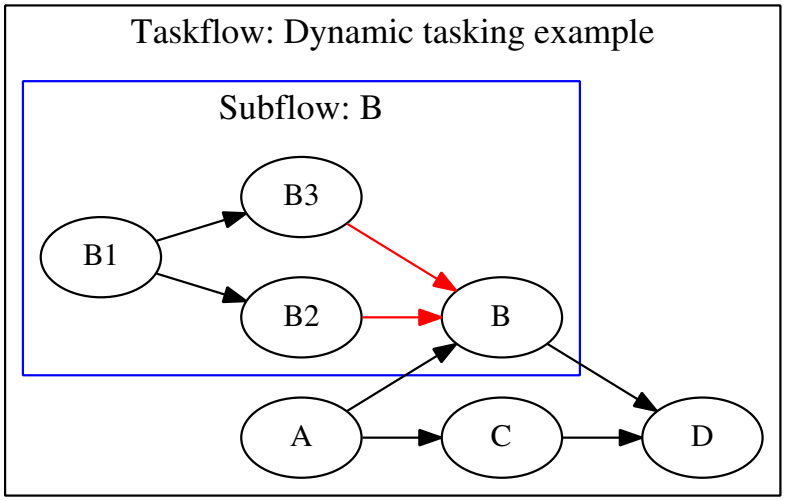


Figure 4.5: An example to show dynamic tasking with joined mode. The red edges are deliberately added by the executor at runtime to ensure task D can only start after the spawned graph finishes.

tasks, adding dependency edges between the sink tasks in spawned graph and the parent task is no longer guaranteed to respect the execution order. Take Figure 4.6 as an example: it not feasible to add a dependency edge between the tasks in subflow and task B to ensure correct execution order between the new spawned graph and task B's successor. To resolve this issue, we

need to know when the spawned task graph finishes execution and be able to reach the parent task after the spawned task graph finishes. We first solve the latter via adding a pointer in each task. The pointer either stores the address of the parent task or is `nullptr` if the task is not in a spawned task graph. The former is indeed equivalent to determining when a task graph with condition tasks will finish. As the aforementioned solution we can use a counter to keep track of the number of tasks in the spawned graph that are kept in the executor. However, since not every task will spawn task graphs, adding an additional counter to each node specifically for the spawned graph is considered overkill. Alternatively, we can reuse the parent task's dependency counter for this purpose. Since each node maintains a parent pointer and we will only revisit the parent task after its spawned task graph finishes, reusing the dependency counter is safe and can reduce memory consumption.

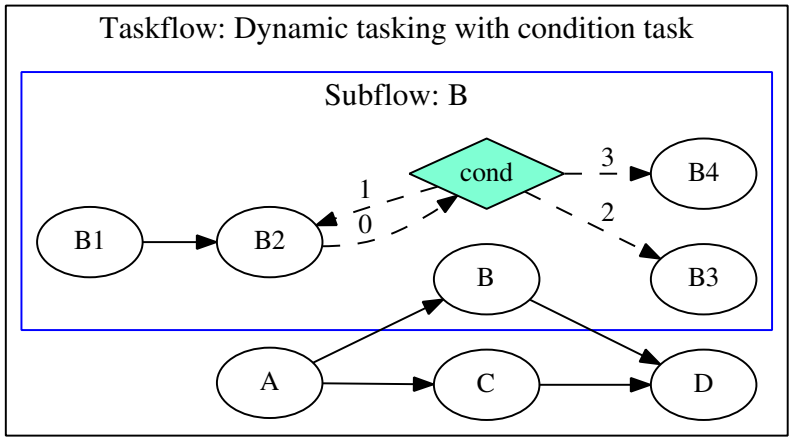


Figure 4.6: An example of dynamic tasking with condition task.

For the last problem, the primary change brought by conditional tasking is determining when the task graph finishes, and as previously mentioned we solve this problem by having a counter to keep track of the tasks kept in the executor. Unlike the original task graph where we only need to update the counter after executing sink tasks, we need to update the counter whenever a task is added into queue or after a task is executed by a worker. To integrate the counter update, there are two functions that need to be modified: (1) `invoke_task` and (2) `exploit_task`. Because a counter could be concurrently modified by multiple workers, the update operation has to be atomic, which can be slower than a non-atomic operation. Therefore, reduc-

ing the number of accesses to the counter is critical to the performance, and we propose two optimizations to lower the number of accesses to the counter.

We explain these optimizations with the pseudo code of the two modified functions below. Algorithm 7 is the pseudocode of `invoke_task`. To clearly explain our optimization, Algorithm 7 only shows the modified part due to the counter update. The worker first invokes task according to the type of tasking, and then decrements the dependency of successor tasks and increments the graph's counter (line 9-28). When a successor task is ready, the task will be put into cache if the cache is empty (line 12-14), otherwise we move the cached task to queue and cache the new ready task (line 14-26). Theoretically, whenever a task is added into queue, the worker has to increment the counter. We can prevent this by adding the number of successors to the counter when inserting the first successor task to the queue. (line 15-22). The rationale is that the worker can only dispatch all successor tasks to execution at most, and therefore we can overestimate the number of inserted tasks for the first time to prevent future increments (line 17-21). Then, we calculate the difference between `num_spawns` (real number of inserted tasks) and `num_successors`, and add it to the worker's local variable `num_executed` (line 31). The `num_executed` records the number of tasks executed by the worker and will be subtracted from the counter later on.

The second optimization is in Algorithm 8 which is the pseudocode of task exploitation. As explained in the previous chapter, in the `exploit_task` the worker will repeatedly pop and execute tasks from local storage (cache and local queue) until exhausted (line 8-47). Theoretically, whenever a task is removed from the queue and executed (line 13), the worker has to decrement the counter. We can reduce the number of decrements by accumulating the number of executed tasks in a local counter (`num_executed`) (line 17), and then decrementing the counter at the end. A worker needs to perform the decrement (1) when successive ready tasks have different parent tasks (line 18-29), which happens due to joined dynamic tasking or composed tasks, and (2) when tasks in local storage are exhausted (line 30-45).

Algorithm 7: The new `invoke_task` function

```
Input: w: the worker's associated data
Input: t: a task holder
1
2  // First invoke the task based on its tasking type,
3  // refer to Chapter 2:  Algorithm 1 line 1-17
4
5  num_spawns ← 0;
6  w.cache ← NIL;
7  num_successors ← t.num_successors();
8  // Update successors' dependency and the graph's counter
9  for s ∈ t's successors do
10     AtomDec(s.dependencies);
11     if s.dependencies == 0 then
12         if w.cache == NIL then
13             w.cache ← s;
14         else
15             if num_spawn == 0 then
16                 /* increment the counter for the first time  */
17                 if t.parent == NIL then
18                     AtomInc(t.topology().counter, num_successors);
19                 else
20                     AtomInc(t.parent().counter, num_successors);
21                 end
22             end
23             num_spawn ← num_spawn + 1;
24             schedule(w.cache);
25             w.cache ← s;
26         end
27     end
28 end
29 if num_spawns > 0 then
30     /* record the difference for counter adjustment       */
31     w.num_executed ← num_successor - num_spawns;
32 end
```

Algorithm 8: The new `exploit_task` function.

```
Input: w: the worker's associated data
Input: t: a task holder
1  if t ≠ NIL then
2      if AtomInc(num_actives) == 1 and num_thieves == 0 then
3          notifier.notify_one();
4      end
5      tpg ← t.topology();
6      par ← t.parent();
7      w.num_executed ← 1;
8      do
9          execute(t);
10         if w.cache ≠ NIL then
11             t ← w.cache;
12         else
13             t ← pop(w.queue);
14             if t ≠ NIL then
15                 /* compare parents of tasks                    */
16                 if t.parent() == par then
17                     w.num_executed ← w.num_executed + 1;
18                 else
19                     if par == NIL then
20                         AtomDec(tpg.counter, w.num_executed);
21                     else
22                         AtomDec(par.dependencies, w.num_executed);
23                         if par.counter == 0 then
24                             schedule(par);
25                         end
26                     end
27                     par ← t.parent();
28                     w.num_executed ← 1;
29                 end
30             else
31                 /* tasks are exhausted                         */
32                 if par == NIL then
33                     AtomDec(tpg.counter, w.num_executed);
34                     if tpg.counter == 0 then
35                         mark_finished(tpg);
36                     end
37                 else
38                     AtomDec(par.dependencies, w.num_executed);
39                     if par.counter == 0 then
40                         t ← par;
41                         par ← t.parent() ;
42                         w.num_executed ← 1;
43                     end
44                 end
45             end
46         end
47     while t ≠ NIL;
48     AtomDec(num_actives);
49 end
```

In both cases, if the parent pointer is not a `nullptr`, we decrement the dependency of its parent task via the pointer. If the dependency of the parent task becomes zero, this means the spawned task graph finishes and we schedule the parent task to execution (line 23-25 and line 39-43). Similarly, if the parent pointer is `nullptr` and the counter becomes zero, we mark the graph as finished (line 34-36).

4.5 Conclusion

In this chapter, we have shown the improvements we made to Cpp-Taskflow on both the programming model and interface. First, we separate the task graph and executor, allowing multiple task graphs to coexist and run in any order. Second, we add a composable task interface to support task graph composition. Composability enables developers to build large and complex applications by assembling small and simple task graphs, instead of coding a complex task graph from scratch. Lastly, we introduce a powerful feature: conditional tasking. With conditional tasking, task graphs can contain cycles and flexibly change the execution flow during runtime, which are very hard to do and even infeasible in Cpp-Taskflow before.

CHAPTER 5

ANALYTICAL PLACEMENT WITH GPU

5.1 Introduction

VLSI global placement is a pivotal stage in physical design flow. Substantial research effort has been devoted to global placement [41, 42, 43, 44, 45, 46, 47, 48, 49, 50]. Among existing placement methods, the analytical approaches, especially the nonlinear placement, have obtained the best quality up to the present. However, compared with other approaches such as simulated annealing or partitioning, the nonlinear methods suffer from a slower performance. The reason is that nonlinear methods apply mathematical programming to derive the solution, which involves huge numbers of arithmetic operations and becomes the bottleneck of performance. Therefore, the idea is to exploit the parallelism of GPU to speed up the computations in nonlinear placement.

GPU is well-known for its capability to conduct massive computations concurrently. There are several research works on applying GPU to EDA applications [51] [52] [53] [54] [55], and some focus on using GPU on EDA placement. The authors of [51] propose a fast sparse matrix-vector multiplication method based on GPU and utilize the method to expedite a quadratic placer. Another paper [52] demonstrates the feasibility of accelerating simulated annealing placement with GPU. The placement models in both papers are different from state-of-the-art nonlinear placement and it is not clear how the methods can be extended to benefit nonlinear placement. The paper [53] applies GPU to optimize the performance of an analytical placer. The authors adopt a straightforward parallelization method such as delegating the outer loop of a nested loop to GPU threads which does not require modifying the original computing scheme. However, adherence to the CPU computing scheme restricts the method from fully exploiting potential paral-

lelism brought by GPU, and some critical issues such as imbalance workload cannot be effectively resolved due to the framework's inherent limitation. Furthermore, due to the limited GPU compute capability, their method has to compromise with reduced numerical accuracy which degrades the solution quality.

In this chapter, we consider the cost model that is broadly used by existing nonlinear placement approaches. The cost model of nonlinear placement approaches can be generally formulated as

$$\begin{aligned} &\text{minimize} && \text{Wirelength} \\ &\text{subject to} && D_{bin} \leq D_{threshold} \end{aligned}$$

The wirelength is a differentiable function, e.g. log-sum-exp [56] or weighted average [57], that approximates the half-perimeter wirelength (HPWL) and the D_{bin} is the bin density on the layout. Mathematical optimization such as the iterative gradient descent method is commonly applied to minimize the cost. As a result, fast computation of the wirelength gradient and the bin density is important to the performance of the placer, and we develop two GPU approaches to accelerate the computing of the wirelength gradient and the density, respectively. We summarize our contributions as follows:

- Our method is faster than the CPU methods and can obtain further speedup over a straightforward GPU parallelization. The efficiency of our methods has been evaluated through experimenting on a set of contest benchmarks.
- Our method does not design for a specific placer; instead, placers that adopt the same cost model can apply the proposed method to achieve performance improvement.
- Reproducibility is guaranteed in the proposed methods. A stable and reproducible output is particularly useful in software debugging.

5.2 Wirelength Computation

Wirelength is one of the most important cost functions in VLSI placement. A commonly used wirelength model is the half perimeter wirelength (HPWL)

which sums the width and height of the bounding box formed by the pins. However, HPWL is a non-differentiable function and thus cannot be directly used in the analytical placement method. Several approximation models are proposed and one popular approach is the Logarithm-Sum-Exponential (LSE) model [56]. The LSE of a given net n with m pins on it can be calculated as follows:

$$\begin{aligned} LSE(n) = \gamma\{\ln(\sum_{i=0}^{m} e^{x_i/\gamma}) + \ln(\sum_{i=0}^{m} e^{-x_i/\gamma})\} + \\ \gamma\{\ln(\sum_{i=0}^{m} e^{y_i/\gamma}) + \ln(\sum_{i=0}^{m} e^{-y_i/\gamma})\} \end{aligned} \tag{5.1}$$

In equation (5.1), each (x_i, y_i) is the x and y coordinate of a pin on n and γ is a predefined constant. Since the gradient calculation is identical in both x and y directions, we only discuss the computation on x below. For a pin connected with k nets, its wirelength gradient can be derived by differentiating the LSE equations of the k nets and sum them up. Below is the equation to calculate the wirelength gradient of a given pin p in the x direction (the gradient in the y direction can be derived using the same formula by replacing the x coordinate with the y coordinate).

$$Grad_x(p) = \sum_{i=0}^{k}\{\frac{e^{x_p/\gamma}}{\sum\limits_{v \in n_i} e^{v_x/\gamma}}\} - \sum_{i=0}^{k}\{\frac{e^{-x_p/\gamma}}{\sum\limits_{v \in n_i} e^{-v_x/\gamma}}\} \tag{5.2}$$

The implementation to calculate equation (5.2) on CPU can be divided into two steps. In the first step, for each net, we compute the summation of the exponential term for each pin on the net. In the second step, we use the summations from the first step following equation (5.2) to derive the gradients of each pin in both x and y directions. Algorithm 9 shows the pseudo code to compute the wirelength gradient of each pin on the x coordinate, where the first step is from line 3 to line 12 and line 13 to 21 is the second step.

Algorithm 9: Wirelength gradient on x using CPU

```
Input: P: Pins
Input: N: nets
Output: Grad: gradients of each pin
1  ExpSum ← {} ;
2  NegExpSum ← {} ;
3  for each n in N do
4      sum ← 0 ;
5      neg_sum ← 0 ;
6      for each p of n do
7          sum ← sum + e^(x_p/γ) ;
8          neg_sum ← neg_sum + e^(−x_p/γ) ;
9      end
10     ExpSum ← ExpSum ∪ {sum} ;
11     NegExpSum ← NegExpSum ∪ {neg_sum} ;
12 end
13 for each p in P do
14     left_term ← 0 ;
15     right_term ← 0 ;
16     for each n of p do
17         left_term ← left_term + 1/ExpSum[n] ;
18         right_term ← right_term + 1/NegExpSum[n] ;
19     end
20     Grad[p] ← e^(x_p/γ) * left_term − e^(−x_p/γ) * right_term ;
21 end
```

5.2.1 Wirelength Gradient on GPU

GPU is suitable for the gradient computation due to its ability to do massive computations concurrently. To utilize GPU for wirelength gradient computing, an intuitive way is to launch two kernels sequentially with the first kernel executing the first step and another the second step. To be more specific, in the first kernel (Algorithm 10), we assign a thread to a net to compute the exponential sum of its pin coordinates, and in the second kernel (Algorithm 11), a thread is delegated to compute the gradients for a pin.

This method is simple and does not require any modification to the CPU algorithm. However, there are two deficiencies in this method:

- As the number of pins on nets are disparate and pins have different numbers of connected nets, the memory access can be very inefficient and threads can suffer from imbalance workload; for example, a pin

Algorithm 10: GPU Kernel 1 on exponential sum

Input: P: Pins
Input: N: nets
Output: $ExpSum$, $NegExpSum$

```
1 id ← blockSize * blockId + threadId ;
2 sum ← 0 ;
3 neg_sum ← 0 ;
4 for each p of N[id] do
5 |   sum ← sum + e^(x_p/γ) ;
6 |   neg_sum ← neg_sum + e^(−x_p/γ) ;
7 end
8 ExpSum[id] ← {sum} ;
9 NegExpSum[id] ← {neg_sum} ;
```

Algorithm 11: GPU Kernel 2 on wirelength gradient

Input: P: Pins
Input: N: nets
Input: $ExpSum$, $NegExpSum$
Output: $Grad$: gradients of each pin

```
1 id ← blockSize * blockId + threadId ;
2 left_term ← 0 ;
3 right_term ← 0 ;
4 for each n of P[id] do
5 |   left_term ← left_term + 1/ExpSum[n] ;
6 |   right_term ← right_term + 1/NegExpSum[n] ;
7 end
8 Grad[id] ← e^(x_p/γ) * left_term − e^(−x_p/γ) * right_term ;
```

coordinate can be read multiple times in different threads and some threads can perform more computations than others.

- Same values can be computed several times in different threads, leading to the inefficient use of computing resources. For example, a pin p_1 can be connected to nets n_1 and n_2 and thus the exponential value of the p_1 coordinate will be computed twice in different threads of the first kernel.

5.2.2 Our GPU Implementation

To overcome these deficiencies, we propose a new GPU implementation flow containing five steps. To prevent computing the same value repetitively among threads, we first launch a kernel to calculate the exponential values of each pin's coordinates and store the result in a vector for later use (Algorithm 12). Based on the fact that a circuit can be represented as a sparse graph, we construct a sparse $(0, 1)$-matrix where the rows correspond to nets, columns correspond to pins, and the value of an entry (i, j) is 1 if pin j is in net i. With the sparse matrix and the vector of the exponential values, the exponential sum of each net can be derived through multiplying the sparse matrix with the vector (Algorithm 13). The next step is to launch a kernel to compute the reciprocal of the exponential sum for each net (Algorithm 14). To calculate the summation of reciprocals for each pin, a kernel is used to sum the reciprocals of all connected nets and a key observation here is that the summation can also be obtained by multiplying a sparse matrix with the reciprocals (Algorithm 15), where the sparse matrix is the transpose of the sparse matrix in the second step. The last step is to derive the gradient by adding the sum of reciprocals for each pin.

Algorithm 12: GPU Step 1 on exponential values

Input: P: Pins
Output: $ExpVal$,$NegExpVal$

1 $id \leftarrow blockSize * blockId + threadId$;
2 $p \leftarrow P[id]$;
3 $ExpVal[id] \leftarrow e^{x_p/\gamma}$;
4 $NegExpVal[id] \leftarrow -e^{-x_p/\gamma}$;

Algorithm 13: GPU Step 2 on exponential sum

Input: P: Pins
Input: N: Nets
Input: $ExpVal$, $NegExpVal$
Output: $ExpSum$,$NegExpSum$

1 $ExpSum \leftarrow SparseMatrix(P, N) * ExpVal$;
2 $NegExpSum \leftarrow SparseMatrix(P, N) * NegExpVal$;

A major concern of using GPU is the overhead incurred from data transfer between CPU and GPU. The proposed GPU method requires two data

Algorithm 14: GPU Step 3 on reciprocal exponential values

Input: $ExpSum$,$NegExpSum$
Output: $RecExpVal$, $RecNegExpVal$

1 $id \leftarrow blockSize * blockId + threadId$;
2 $RecExpVal[id] \leftarrow 1/ExpSum[id]$;
3 $RecNegExpVal[id] \leftarrow 1/NegExpSum[id]$;

Algorithm 15: GPU Step 4 on summation of reciprocals

Input: P: Pins
Input: N: Nets
Input: $RecExpSum$,$RecNegExpSum$
Output: $RecSum$, $RecNegSum$

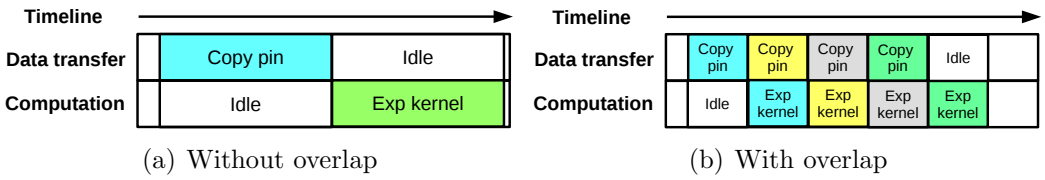
1 $RecSum \leftarrow SparseMatrix(P, N)^T * RecExpSum$;
2 $RecNegSum \leftarrow SparseMatrix(P, N)^T * RecNegExpSum$;

Algorithm 16: GPU Step 5 on gradient of each pin

Input: P: Pins
Input: N: Nets
Input: $RecSum$, $RecNegSum$, $ExpVal$, $NegExpVal$
Output: $Grad$

1 $id \leftarrow blockSize * blockId + threadId$;
2 $Grad[id] \leftarrow RecSum[id] * ExpVal[id] +$
3 $\qquad RecNegSum[id] * NegExpVal[id]$;

transfers: one is to transfer the pin coordinates from CPU to GPU memory in the beginning and another is to copy the gradients back to CPU memory. Although the data transfer overhead is inevitable, we can further reduce the overhead by using streams. A stream is similar to a job queue which holds GPU operations to be executed sequentially, whereas operations in separate streams can run concurrently if available resource exists. Hence, we can create several streams to overlap data transfers with computations through dispatching GPU operations on subsets of data to different streams. Considering overlapping the step 1 kernel (Algorithm 12) by copying pin coordinates to GPU, we first divide pins into disjoint subsets and map each subset to a stream, then a copy operation and a kernel for computing exponential value are enqueued into each stream to operate on the corresponding data. By having multiple streams process different subsets, we can keep the copy device and execution device occupied [58] as shown in Figure 5.1.



(a) Without overlap (b) With overlap

Figure 5.1: Comparison of data transfer with and without overlapped with computations.

Our proposed flow has two benefits over the straightforward GPU implementation:

- We transform the two nested loops, the most time-consuming parts, to two sparse matrix multiplications. A sparse matrix can be stored in various formats such as a compressed sparse row (csr) or a coordinate list (coo) and those data structures unleash more opportunities to optimize the memory access and reorder the computations for balancing the workload.
- Data movement between processing units is a common bottleneck in heterogeneous computing and our approach reduces the overhead by overlapping computations with data transfers through utilizing streams.

5.3 Density Computation

Density computation is an essential step in analytical placement methods. During the optimization process, the density will be evaluated in every iteration and the placement can stop once the cells' overlap is lower than a predefined threshold. In this section, we first formulate the density computation problem and present a CPU implementation, then we demonstrate a straightforward GPU implementation and discuss its deficiencies. Lastly, we propose a high-precision GPU implementation and introduce two techniques to further improve the performance.

5.3.1 Density Problem Formulation

We consider a general formulation where the layout is a two-dimensional grid and the cells to be placed are rectangular. To compute the density,

the first step is to accumulate the overlapped area between bins and the cells. Then the density can be derived by dividing the accumulated area in each bin by the unit bin area (a coarser density map can be formed by combining multiple bins into a single bin). Parallelizing the second step is pretty straightforward, so in this section we will focus on parallelization of the overlapped area accumulation.

5.3.2 Area Accumulation on CPU

Algorithm 17: Area accumulation on CPU

```
   Input: Cells, Grid
   Output: OverlapArea
 1 for each b ∈ Grid do
 2 |  OverlapArea[b] ← 0 ;
 3 end
 4 for each c ∈ C do
 5 |  overlap_bins ← FindOverlapBins(Grid, c) ;
 6 |  for each b ∈ overlap_bins do
 7 |  |  area ← FinOverlapArea(b, c) ;
 8 |  |  OverlapArea[b] ← area + OverlapArea[b] ;
 9 |  end
10 end
```

To compute the overlapped area between cells and bins, an intuitive way is to loop through each cell and add the overlapped area of the cell to corresponding bins. Algorithm 17 is the pseudo code to accumulate overlapped area for each bin. Reproducibility is guaranteed in this CPU implementation as the floating point additions are executed in deterministic order.

To speed up the computation, an instinctive way is to use the multiple cores in CPU to have several threads doing the accumulation concurrently. Consider line 4 in Algorithm 17: cells can be partitioned into subsets with nearly equal size and each thread is responsible for a subset of cells. A challenge of parallel programming is to maintain the data integrity under multi-thread execution. Notice that in line 8, the accumulated area in each bin is shared among all threads and concurrent access to this data might result in data race.

A simple and efficient solution is to use atomic operation. An atomic

operation serializes the access to the data without using locks, which protects the data from running into the race condition and can still maintain good performance. However, in C++ the standard library does not provide atomic operations for the floating type. An alternative is to implement the atomic floating operation via the atomic *compare-and-exchange* instruction. The compare-and-exchange instruction atomically checks whether the destination value is equal to a given value and replaces the destination value by a new value if the predicate is satisfied. Therefore, a thread can first take a snapshot of the destination bin and then use the compare-and-exchange instruction with the snapshot value to update the bin. Algorithm 18 presents the pseudo code of the multi-threaded area accumulation. In lines 1 and 2, we launch multiple threads and assign a subset of cells to each thread. From lines 3 to 14, each thread adds the overlapped area of each cell to the corresponding bins via the compare-and-exchange instruction in line 11.

Algorithm 18: Multi-threaded area accumulation on CPU

```
   Input: Cells, Grid
   Output: Bin
1  launchThreads() ;
2  myCell ← AssignCells(myThreadId, Cells) ;
3  for each c ∈ myCell do
4      overlap_bins ← FindOverlapBins(Grid, c) ;
5      for each b ∈ overlap_bins do
6          area ← FindOverlapArea(b, c) ;
7          update ← false ;
8          while update ≠ true do
9              snapshot ← Bin[b] ;
10             new_value ← snapshot + area ;
11             update ← CAE(&Bin[b], snapshot, new_value) ;
12         end
13     end
14 end
15 synchronizeAllThreads() ;
```

5.3.3 Area Accumulation on GPU

The multi-threaded CPU approach of area accumulation can also be applied to GPU. For the GPU method, we launch a kernel with assigning a thread to

each cell to compute the overlapped area among bins. In contrast to the CPU, modern GPUs support atomic operation for the floating type, circumventing the need of using compare-and-exchange instruction. However, there are two deficiencies in this approach:

- No guarantee of reproducibility: Although the atomic operation resolves the data race problem, it does not admit reproducibility. The reason is that floating point arithmetic is non-associative [59]. As atomic operation does not enforce a deterministic order on thread execution, given the same operands the result could be slightly different every time.

- Imbalanced workload: As the size of the cells is not uniform, the number of bins crossed by different cells could differ greatly. Therefore, divergence might occur in the kernel due to the inconsistent iterations among threads (line 6 in Algorithm 17), which might hamper the performance.

To solve the first problem, we adopt a high-precision method [60]. The method represents every floating number by N 64 bit integers where each integer carries a fraction of the floating number and N controls the representable range of floating number. Our idea is to perform accumulation on the integers converted from floating numbers and the resulting integers can then be translated into floating numbers after the accumulation finishes. As the arithmetic addition on integers is associative, given the same inputs the outcome of accumulation will be identical. Another benefit of this method is obviating the need to impose a predefined execution order among threads which might be detrimental to performance.

To evenly distribute the workload among threads, we come up with a computation flattening technique. Since the dimensions of cells and bins are known in the beginning and remain unchanged during placement, we can derive the maximum number of bins that intersect with each cell before placement. With this information, we can determine the total number of threads to be launched by summing up the number of bins intersecting with each cell, and assign a thread to compute the overlap area between a cell and one of its intersected bins and add the result to the corresponding bin. For example, Figure 5.2 shows flattening computation of two cells.

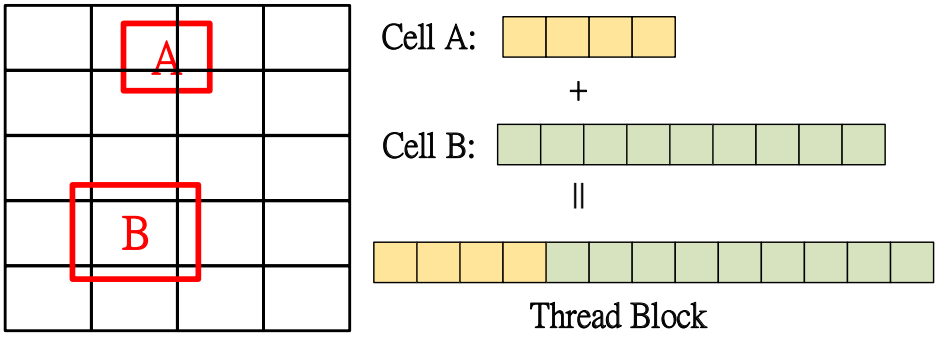


Figure 5.2: An example illustrates computation flattening. Cells A and B intersect with four and nine bins respectively and 13 threads are created to compute the overlapped area for each portion.

Algorithm 19: High-Precision area accumulation on GPU

Input: $Cells$, $Grid$, $myCellId$, $myBinId$, $numStream$

Output: Bin

```
1  cudaMemSet(hpBinArea, 0) ;
2  cudaCreateStream(Streams, numStream) ;
3  for each s ∈ Streams do
4      s ← copyPin(myCellId, Cells) ;
5      s ← countOverlap(myCellId, Cells, hpBinArea) ;
6  end
7  cudaDeviceSynochronize() ;
8  for each s ∈ Streams do
9      s ← convertToFloat(myBinId, hpBinArea, fBinArea) ;
10     s ← copyDensity(myBinId, fBinArea, Bin) ;
11 end
```

This approach also requires two data transfers: one is to send the cell coordinates to GPU memory and another is to fetch the accumulated area from GPU memory. To further reduce the transfer overhead, we use streams to overlap both data transfers with two computation kernels. For the cell coordinates transfer, we map subsets of cells to streams and a kernel is enqueued into each stream to compute the overlapped area in the high-precision format for the cells. For the second data transfer, we associate a subset of bins to streams and each stream enqueues a kernel to convert the integers to floating numbers for the bins and copy the results to CPU memory. Algorithm 19 is the pseudo code of the high-precision GPU method with streams.

5.4 Experimental Results

We implement all programs in C++. The GPU used in the experiment is NVIDIA GeForce GTX 1080 and the CPU is Xeon 3.0 GHz Quad cores with 32 GB memory. We implement a gradient descent placer based on the LSE wirelength model and conduct experiments on the benchmarks from the 2015 ISPD routing-driven placement contest [61]. The statistics of the benchmarks are in Table 5.1. For GPU programs, we record the runtime from host (CPU) side, including kernel launch latency, the overhead of pin coordinates (host to GPU) and results (GPU to host) transfer.

Table 5.1: Benchmark statistic

Benchmark	Cells	Nets
mgc_fft_1	32,281	33,307
mgc_fft_2	32,281	33,307
mgc_matrix_mult_1	155,325	158,527
mgc_matrix_mult_a	149,650	154,284
mgc_matrix_mult_b	146,435	151,612
mgc_pci_bridge32_a	29,517	29,985
mgc_pci_bridge32_b	28,914	29,417
mgc_des_perf_1	112,644	112,878
mgc_des_perf_a	108,288	110,281
mgc_des_perf_b	112,644	112,878
mgc_edit_dist_a	127,413	131,134
mgc_fft_a	30,625	32,088
mgc_fft_b	30,625	32,088
mgc_superblue12	1,286,948	1,293,413
mgc_superblue11_a	925,616	935,613
mgc_superblue16_a	680,450	697,303

5.4.1 Wirelength

We implement the wirelength gradient computation with four methods: CPU, CPU with four threads, straightforward GPU parallelization, and our proposed GPU method. The cuSPARSE library [62] is adopted for sparse matrix multiplication in our method. The wirelength gradient computation is not affected by cells' locations, and we report the results in one iteration. Table 5.2 lists the runtime of the four methods. Among the four methods, the

CPU method is the slowest and the proposed GPU method is the fastest over all test cases. Considering the average speedup, our method outperforms the CPU, CPU with four threads and the GPU loop parallelization by 173×, 93× and 8× respectively.

5.4.2 Density

For the density experiment, we implement four methods for comparison: CPU, CPU with four threads (with the compare-and-exchange technique to ensure data integrity), proposed GPU method with and without using streams. We set the N in the high-precision method to 3, i.e., each floating number is represented by three 64-bits integers.

For each test case, the placement stops when successive cell displacement is small and we record the average computation time. The grid size is 2048×2048 for the superblue family and 1024×1024 for the others. Table 5.3 lists the runtime for each test case. From the Table 5.3, the GPU method with streams has the best performance in all test cases while the multi-threaded CPU method only obtains minor improvement in a few benchmarks. The compare-and-exchange operation is the primary cause for the slower performance of the multi-threaded method. Unlike the atomic operation which serializes the access to data, the threads that failed to update the bin using the compare-and-exchange operation have to retrieve the new value and compete for the access (lines 8-12 in Algorithm 18), resulting in high overhead when there are many overlaps between cells.

5.5 Conclusion

In this chapter, we present GPU approaches to accelerate the wirelength gradient and density computation. For the wirelength gradient, we convert the summations into sparse matrix multiplications, which effectively mitigates the non-uniform workload among threads and increases the performance. For the density, we propose a computation flattening technique to resolve the imbalance workload. The runtime can be further reduced by overlapping the computation and data transfer using CUDA stream. Lastly, a high-precision method is integrated into our approach to ensure reproducible results.

Table 5.2: Wirelength gradient computation (μs)

Benchmark	CPU (A)	CPU mt (B)	GPU (C)	Our (D)	A/D	B/D	C/D
mgc fft 1	62818	42856	3965	922	68.13	46.48	4.30
mgc fft 2	63155	45405	3962	878	71.93	51.71	4.51
mgc matrix mult 1	250341	146679	7145	1074	233.09	136.57	6.65
mgc matrix mult a	241860	160679	6883	1050	230.34	153.03	6.56
mgc matrix mult b	235409	153353	6824	1028	229.00	149.18	6.64
mgc pci bridge32 a	58918	36293	5889	645	91.35	56.27	9.13
mgc pci bridge32 b	57155	34299	5828	600	95.26	57.17	9.71
mgc des perf 1	174971	95828	16798	865	202.28	110.78	19.42
mgc des perf a	168382	95718	15887	852	197.63	112.35	18.65
mgc des perf b	174674	105847	17087	884	197.60	119.74	19.33
mgc edit dist a	205182	116110	11107	991	207.05	117.16	11.21
mgc fft a	61156	35510	3742	876	69.81	40.54	4.27
mgc fft b	61032	34818	3767	896	68.12	38.86	4.20
mgc superblue12	2520856	933960	38512	10083	250.01	92.63	3.82
mgc superblue11 a	1619743	636320	16169	5671	285.62	112.21	2.85
mgc superblue16 a	1194524	464355	12480	4301	277.73	107.96	2.90
Avg speedup					173.43	93.91	8.38

Table 5.3: Area accumulation (μs)

Benchmark	CPU (A)	CPU mt (B)	GPU w/o stream (C)	GPU w/ stream (D)	A/D	B/D	C/D
mgc_fft_1	11290.84	12493.41	7321.55	2419.29	4.67	5.16	3.03
mgc_fft_2	8093.70	8860.68	6047.33	2189.98	3.70	4.05	2.76
mgc_matrix_mult_1	15005.26	19605.24	7917.75	3059.32	4.90	6.41	2.59
mgc_matrix_mult_a	6846.51	8750.37	4906.24	2083.73	3.29	4.20	2.35
mgc_matrix_mult_b	7027.02	8671.27	4974.14	2117.35	3.32	4.10	2.35
mgc_pci_bridge32_a	6003.96	5876.86	5507.50	2078.25	2.89	2.83	2.65
mgc_pci_bridge32_b	3173.07	2993.10	4425.70	1794.05	1.77	1.67	2.47
mgc_des_perf_1	12641.25	18864.96	7431.84	2556.22	4.95	7.38	2.91
mgc_des_perf_a	6966.61	8736.17	5253.17	2084.12	3.34	4.19	2.52
mgc_des_perf_b	9942.77	13533.10	6257.44	2408.73	4.13	5.62	2.60
mgc_edit_dist_a	9579.94	12900.03	6292.26	2613.12	3.67	4.94	2.41
mgc_fft_a	3158.54	3378.08	4385.56	1790.72	1.76	1.89	2.45
mgc_fft_b	3011.20	3355.41	4348.91	1778.45	1.69	1.89	2.45
mgc_superblue12	80574.98	73570.18	26326.61	11367.5	7.09	6.47	2.32
mgc_superblue11_a	49883.19	57774.44	19044.40	8728.10	5.72	6.62	2.18
mgc_superblue16_a	32067.20	29805.87	16426.19	7477.87	4.29	3.99	2.20

CHAPTER 6

A DISTRIBUTED POWER GRID ANALYSIS FRAMEWORK

6.1 Introduction

As the technology continues to advance, analyzing a power distributed network that corporates billions of transistors becomes a critical challenge. Traditionally, power analysis engineers partitioned the problem into smaller and manageable pieces, and ran each on a single multi-threading machine. However, according to [63], analyzing a power grid with 136 million nodes on a single multi-core machine can take hundreds of GBs of memory and several hours to finish. Building such a high-end computer is expensive and unscalable to the ever-increasing design complexities. As a result, EDA vendors are driving the need for distributed power grid analysis.

Researchers have proposed parallel computing methods for power grid analysis [64] [65] [66] [67] [68]. Existing works are based on either multi-threading in a shared memory storage or distributed computations across different nodes. The work reported in [64] [65] involved developing parallel power grid simulators by taking the advantage of multi-cores with shared memory to speed up the computing. Although the shared memory model is advantageous in data communication, it relies on expensive hardware resources to gain more scalability. The work reported in [66] [67] involved designing parallel computing schemes by partitioning data and distributing the computations across multiple machines using the low-level massage passing interface (MPI) library [69]. While MPI provides a layer of abstraction over the network communication, it suffers from many distinct notations to express the parallelism. The bottom-up design principle of MPI is analogous to assembly languages in terms of writing parallel code. For example, users have to manually name the machines for process mapping and hard-code message passing for serialization and deserialization. It also requires a

significant amount of coding effort when the software changes to the next generation. Taken together, these issues discourage developers from being productive and innovative. Nevertheless, building a distributed power grid analysis beyond MPI remains an open problem.

While existing big-data tools hold much promise for distributed computing [70], EDA researchers remain skeptical about the applicability for many reasons [71]. First, power grid analysis is compute-intensive whereas big data computing focuses on I/O processing. Second, the MapReduce paradigm assumes data can be split into independent chunks while the power grid data are not easily separable. Third, the mainstream programming languages of big data are JVM languages that do not appeal to the language need of the power grid (C/C++). As a consequence, we need a specialized distributed framework for power grid analysis.

In this chapter, we introduce a distributed power grid analysis framework based on the stream graph model. The goal of this work is, instead of solving the power grid analysis with domain-specific techniques, to investigate the programmability, extensibility, and scalability of distributed power grid analysis at the framework level. We summarize our contributions as follows:

- We show that with the use of the *stream graph* programming paradigm, programming distributed power grid analysis can be greatly simplified. Unlike MPI which is based on low-level message passing API, the stream graph is a higher-level abstraction to express parallelism. We focus on developing the framework based on the algorithmic specification, without wrestling with system-specific implementation details.
- We show that with a customized scheduler, we are able to maximize the resource utilization in a cluster. Our scheduler is tailored for the compute-intensive power grid analysis. We demonstrate that our scheduler can effectively leverage the CPU usage for this particular workload.
- We show that our framework is a more flexible and scalable alternative to MPI-based solutions. We can flexibly partition the power grid to different subdomains regardless of the number of cores, which is impossible in MPI due to its architectural limitation.

We implement our framework on DtCraft[1] [6], a distributed execution

[1]We use DtCraft version 0.0.1 for the implementation.

engine for high-performance applications, for our experiment. The experimental results show that our distributed power grid framework achieves comparable performance to MPI-based solutions. We also demonstrated the effectiveness of our scheduler in an emulated production environment. Compared with DtCraft's default scheduler, our scheduler effectively reduces the total execution time in the emulated production environment.

6.2 Distributed Power Grid Analysis

The goal of power grid analysis is to solve following system of equations extracted from the associated circuit:

$$GV = I,$$

G : A matrix formed by the conductance of components

V : A vector consists the voltage of nodes (unknown)

I : A vector consists the independent current sources

By solving the above linear system, the voltage drop at each node can be derived by comparing the node voltage V with the supply voltage. Directly solving the system is not practical when there are millions of nodes in the circuit. One feasible solution is domain decomposition [65] [72] [73] which partitions the problem into subsets and solves them in parallel. The Additive Schwarz Method (ASM), one type of the domain decomposition methods, is especially suitable for large sparse system [66]. In this work, we adopt the geometric ASM method with 2D partitioning proposed by [66] for distributed direct current (DC) analysis, which is proved to have minimum data communication. The geometric ASM method for DC analysis can be summarized as four steps:

S1: Partition the circuit into subdomains.

S2: Solve each subdomain independently.

S3: Synchronize and exchange the boundary values of subdomains.

S4: Go to S2 if any of the subdomains do not converge.

The geometric ASM method is suitable for distributed computing as it can be directly parallelized by assigning the subdomains to different processors.

6.2.1 Existing Works and Limitations

Based on the geometric ASM method, researchers developed a number of distributed power grid analysis systems using MPI [66] [72] [73]. The MPI programming model is processor-centric. A MPI program consists of several processes with each process attached to a processor, and a typical MPI program can have a number of processes less than or equal to the number of available processors. Even though over-subscription is possible in MPI, it is discouraged by the official website due to performance degradation. The processes form a communication group and each process has a unique number called rank for identification. Processes can send or receive data through using the rank in a set of APIs. Based on the message passing model, a distributed DC analysis program with MPI is shown in Algorithm 20.

In Algorithm 20, the power grid is partitioned into $W \times H$ subdomains and the MPI program launches $W \times H$ processes while assigning each process a subdomain. In particular, the $0th$ rank process is different from others in that it not only solves a subdomain but also has to partition the power grid, gather and redistribute the boundary values, and check the convergence of all processes. Notice that in line 8, a process handles the subdomain based on the rank automatically assigned by MPI and the rank is limited by the number of available CPU cores.

Although users can implement a distributed computing program by directly including the MPI library and utilizing the low-level APIs, there are several disadvantages of the MPI model:

- The number of subdomains is limited by the processors and users are only allowed to subscribe processes up to the number of physical processors. Also, this is a constraint to launching the program (`mpirun -n [number of cores]`). This fundamentally restricts our problem-solving logic to deliver an effective and scalable solution.

- To manage all processes running concurrently in the MPI model, an MPI program needs to explicitly use conditional instructions or branch predicate to separate the execution flows of different processes. This complicates the MPI program structure and also makes the MPI program difficult to extend to incremental analysis [74], where some processes might change the values in subdomains during analysis.

Algorithm 20: MPI-based Distributed DC analysis

```
Input: C: circuit
Input: W: width
Input: H: height
MPI_Init();
rank ← MPI_Rank();
subdomains ← ∅;
if rank == 0 then
    PartitionGrid(C, W, H);
end
MPI_Sync();
subdomains[rank] ←ReadGrid(rank);
bd_value_num ← CountBD(subdomains[rank]);
MPI_Gather( 0, bd_nums, bd_value_num);
if rank == 0 then
    bd_array ← CreateBoundaryArray(bd_nums) ;
end
converge ← False;
solution ← {0};
while !converge do
    Solve(subdomains[rank], solution);
    bd_value ← ExtractBoundary(solution);
    converge ← Check(solution);
    MPI_Gather( 0, bd_array, bd_value);
    MPI_Gather( 0, result, converge);
    if rank == 0 then
        converge ← IsConverge(result);
        Reorder(bd_array);
    end
    MPI_Scatter(0, bd_array, bd_value);
    UpdateBD(bd_value, solution);
    MPI_broadcast(0, converge);
end
```

These disadvantages have a negative impact on the programmability and scalability of the MPI-based distributed power grid analysis, and it is desirable to have a novel distributed computing framework that does not suffer from the same issues.

6.3 Distributed Power Grid Analysis based on Stream Graph

6.3.1 Stream Graph Model

Stream graph [6] is a new programming model that aims for distributed computing, especially for high-performance (compute-intensive) applications. A stream graph is a high-level abstraction that describes the program as a directed graph, where vertices and edges encapsulate the data flow and a sequence of computations. The model is simple yet generic as several parallel computing patterns such as the MapReduce can also be represented by the stream graph formulation. An important feature of the stream graph model is that the computations are asynchronous, i.e. computations are only executed when the associated data arrive. This makes the stream graph a competitive solution for performance-driven applications.

6.3.2 DC Analysis in Stream Graph

Based on the stream graph programming paradigm, we formulate the DC analysis as a stream graph with two types of vertices: synchronization vertex and worker vertex. The stream graph of DC analysis consists of one synchronization vertex and N worker vertices where N is the number of subdomains, and there are two directed edges connecting the synchronization vertex and each worker vertex. In general, the synchronization vertex serves as a hub that exchanges data between worker vertices and determines whether the solution converges or not, while the worker vertex is responsible for solving a subdomain and reporting the result to the synchronization vertex. Algorithm 21 presents the stream graph for DC analysis. A synchronization vertex is first inserted into the graph in line 2. Then we insert a worker vertex and two directed edges to the graph (line 7 - 11) and execute the graph in line 12. Figure 6.1 shows a stream graph of a power grid with four subdomains.

The program initializes required data from invoking the synchronization vertex's callback once to prepare subdomains. Then the synchronization vertex notifies the worker vertices of the corresponding subdomains by sending

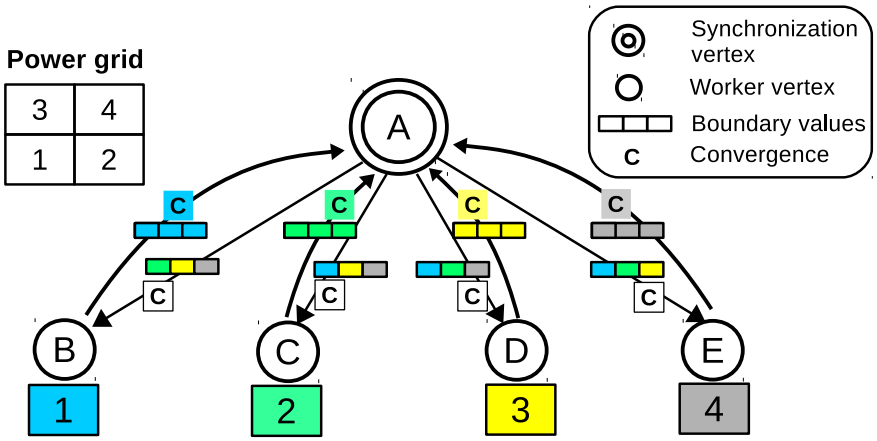


Figure 6.1: A stream graph of a power grid with four worker vertices each owning a subdomain and a synchronization vertex to coordinate the four workers.

Algorithm 21: DC analysis using stream graph

Input: C: circuit
Input: W: width
Input: H: height

```
1  Graph G ;
2  sync ← InsertV(G, sync_cb(C,W,H)) ;
3  workers ← {} ;
4  to_worker ← {} ;
5  to_sync ← {} ;
6  N ← W * H ;
7  for i = 1 to N do
8      workers[i] ← InsertV(G, worker_cb()) ;
9      to_worker[i] ← InsertE(G, sync, workers[i], worker_edge_cb( )) ;
10     to_sync[i] ← InsertE(G, workers[i], sync, sync_edge_cb()) ;
11 end
12 dispatch(G) ;
```

a signal through edges. Algorithm 22 presents the callback of the synchronization vertex. In line 1, the power grid is first partitioned into W×H subdomains and then each subdomain index is passed to a worker vertex along the directed edge (line 2 - 7).

For the input edge callbacks of both types of vertices, we use finite state machines to establish a communication protocol to react to different types of input data. In the callback of a synchronization vertex's input edge, there are two states iterating. The *CHECK* state checks the results from worker

vertices to decide the whole solution converges or not and the *RECV* state reorders the received boundary values and distributes them to the worker vertices to continue to next iteration if the solution is not converged. Algorithm 23 shows the details in the callback of a synchronization vertex at input side. In line 3, the CHECK state gathers the results from worker vertices and informs all worker vertices of the global status once all results are received (line 4 - 11). The callback is removed when reaching convergence (line 10). From line 13 to 23, in the RECV state the synchronization vertex collects and transmits the new boundary values to the worker vertices.

There are three states in the callback of worker vertex's input edge: *INIT*, *COMPUTE* and *WAIT_RESULT*. The INIT state is the first state where the worker vertex waits for the subdomain index, and the COMPUTE and WAIT_RESULT state are used to respectively handle the new boundary values and global result. Algorithm 24 presents the callback of a worker vertex at input side. In the INIT state, each worker vertex first receives a subdomain index from the synchronization vertex (line 2 - 6). Then a worker vertex solves its own subdomain and replies the result and transits to the WAIT_RESULT state (line 23 - 28). In the WAIT_RESULT state, the worker vertex waits for the global result. The callback is removed if the whole solution converges (line 9 - 12); otherwise the worker vertex sends the boundary values to the synchronization vertex and transits to the COMPUTE state (line 13 - 15). In the COMPUTE state (line 18 - 20), when a worker vertex receives the updated boundary values, it proceeds to solve the subdomain with the new values and sends the result back.

Algorithm 22: Callback of a synchronization vertex

```
Input: C: circuit
Input: W: width
Input: H: height
Input: E: edge ids
1 PartitionGrid(C, W, H) ;
2 for i = 1 to W do
3     for j = 1 to H do
4         id ← SubdomainId(i, j) ;
5         send( edges[i][j], id ) ;
6     end
7 end
```

Algorithm 23: Input edge callback of a synchronization vertex

```
Input: id: edge id
switch state do
    case CHECK
        recv( results[id] ) ;
        if all workers are recv then
            done ← AllConverge(results) ? True : False ;
            for i = 1 to N do
                send( i, done ) ;
            end
            state ← RECV ;
            return done ? REMOVE_THIS_CB:DEFAULT ;
        end
    end
    case RECV
        recv( bd_vectors[id] ) ;
        if all workers are recv then
            Reorder(bd_vectors) ;
            for i = 1 to N do
                send( i, bd_vectors[i] ) ;
            end
            state ← CHECK ;
            return DEFAULT ;
        end
    end
endsw
```

The proposed framework has several benefits over the MPI model

- In contrast to the static (manual) mapping of processes to processors in MPI, the stream graph programming paradigm enables task parallelism, i.e. callbacks can be executed on any core in an asynchronous manner, which allows users to create more partitions than the available processors.

- By packaging sequential computations into callbacks and assembling the sequential blocks into a parallel program, the stream graph formulation has better code readability and makes debugging easier, whereas the MPI program is more complex as processes with different execution trajectories are put in the same block and parallelizations are expressed by various low-level APIs (the functions with `MPI_*` prefix).

Algorithm 24: Input edge callback of worker vertex

```
Input: id: edge id
1  switch state do
2      case INIT
3          recv( subdomain_id ) ;
4          my_subdomain ← ReadGrid( subdomain_id ) ;
5          goto 23 ;
6      end
7      case WAIT_RESULT
8          recv( result ) ;
9          if result then
10             Output(id, solution) ;
11             return REMOVE_THIS_CB ;
12         end
13         state ← COMPUTE ;
14         send(id, bd_value) ;
15         return DEFAULT ;
16     end
17     case COMPUTE
18         recv( bd_value ) ;
19         UpdateBD(bd_value, solution) ;
20         goto 23 ;
21     end
22 endsw
23 Solve(my_subdomain, solution) ;
24 bd_value ← ExtractBoundary(solution) ;
25 converge ← Check(solution) ;
26 state ← WAIT_RESULT ;
27 send( id, converge ) ;
28 return DEFAULT;
```

- The stream graph formulation lets users assign the resource requirements in a fine-grained manner. A subgraph can have an individual demand, which allows the scheduler to make a more effective cluster resource utilization.

Combining the above benefits, our framework has better programmability and scalability than the MPI. We believe our framework stands out as a unique solution for distributed power grid analysis, considering the software design and the architectural decision we made.

6.4 Application-specific Resource Control Plug-in

Job scheduling is an important issue in distributed computing as the scheduling has a huge impact on overall system performance. There are many types of resources in a cluster such as CPUs and memory, and different workloads can have diverse demands on the resources. In this section, we first outline the default scheduler in DtCraft, then we introduce a scheduler that is tailored for CPU bound applications such as the power grid analysis to enhance the system performance.

6.4.1 Default Scheduler

The default scheduler in DtCraft adopts a best-fit method to match a job's tasks to machines based on their resource (CPU + memory) requirements. In contrast to the CPUs that are shared among processes, memory claimed by a process will not be available to others during execution. As a result, memory is regarded as a hard constraint and any process that violates the memory constraint will trigger an out-of-memory error and be terminated. The policy of the default scheduler is first-come-first-served and non-preemptive. Whenever the scheduler receives a job from users, it seeks to find a feasible scheduling for the job if there is no job waiting ahead. The scheduler first takes a snapshot of the current status of machines, then for each task in the job, the scheduler scans through the machines to create a list of machines that have enough memory to accommodate the task, and among those candidates the best-fit machine, the one with the least memory, is matched to the task. A job cannot be scheduled if any one of its task fails to be matched to a machine. The failed job will be stored into a queue for future processing. Whenever a job finishes execution and releases the memory, the scheduler will examine the queue to process the waiting jobs. The advantage of this method is that it reduces memory fragmentation which could spare more room to have more jobs scheduled.

6.4.2 Proposed Scheduler

Even though the default scheduler aims to process as many jobs as possible at the same time, the cluster can experience performance slowdown due to

the imbalanced workload among machines. A significant deficiency of the default scheduling policy is the underutilization of CPUs since the default scheduler tends to assign jobs to the machines that are either partially or fully loaded while there still exist idle machines. Figure 6.2 is an example of how the default scheduler assigns a task to a partially loaded machine with the other two machines being unused. Because processes hold CPUs in turn,

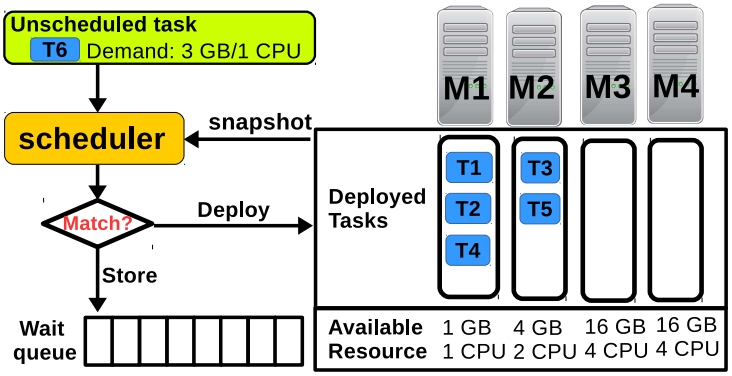


Figure 6.2: In this example, the default scheduler will assign the task T6 to the machine M2 as M2 is the best-fit among all available machines.

excessive processes will lead to shorter time slice owned by each process and higher overhead of frequent context switch, which can dramatically increase the total runtime.

To have better utilization of the cluster resource as well as improve the cluster performance, load balance must be considered in scheduling. We propose a scheduler for balancing the workload of cluster machines. Aside from the memory usage, in order to evenly distribute the workload we integrate the CPU usage and average CPU load in the past one minute into scheduling to decide the deployment. To gauge the workload of each machine, we record the CPU demands of tasks allocated on each machine and define the ratio of total CPU demands to the number of CPUs on the machine as load index. As memory is a hard constraint, during the job scheduling we first collect the machines that satisfy the memory requirement. Then, rather than selecting the machine with the least available memory, a task is matched to the machine with the smallest load index and in case of a tie, the machine with smaller average CPU load in the past one minute is preferred. The goal of using load index to determine the task placement is to proportionally distribute the workload. Algorithm 25 presents the algorithm of the proposed scheduler. In Line 8 - 14, the scheduler first checks the memory capacity

Algorithm 25: Load-aware scheduling algorithm

```
   Input: M: machines
   Input: J: a job
1  snapshot ← {};
2  foreach m ∈ M do
3  |  snapshot ← snapshot ⋃ m;
4  end
5  foreach t ∈ J do
6  |  best ← null;
7  |  foreach s ∈ snapshot do
8  |  |  if s.memory >= t.memory then
9  |  |  |  if s.load < best.load or best == null then
10 |  |  |  |  best ← s;
11 |  |  |  else if s.load == best.load and s.loadavg < best.loadavg
   |  |  |  then
12 |  |  |  |  best ← s;
13 |  |  |  end
14 |  |  end
15 |  end
16 |  if best == null then
17 |  |  P ← ∅ ;
18 |  |  break ;
19 |  end
20 |  else
21 |  |  P ← P ⋃ (t, best) ;  // P:  mapping of tasks to machines
22 |  |  snapshot[best].memory− = t.memory ;
23 |  |  snapshot[best].load+ = t.cpu/snapshot[best].cpu ;
24 |  end
25 end
```

to satisfy the hard constraint. Then for those qualified machines, we deploy the task on the least utilized machine by comparing their load indices and average load over one minute for tie-breaking.

6.5 Experimental Results

We first compare two implementations of distributed DC analysis, the stream graph and the MPI model, to demonstrate their performance on both the single machine and the distributed environment. Next we compare the proposed scheduler with the default scheduler in an emulated production environment.

6.5.1 Stream Graph versus MPI

We conduct experiments on a single machine and a cluster respectively to evaluate both implementations, and a set of industrial power grid benchmarks released by IBM [75] is used throughout the experiments. We use the network file system (NFS) to allow file sharing across the machines. In the single machine experiment, the machine is equipped with a 2.4 GHz quad-core CPU and 35 GB memory. Due to the available number of cores, we partition the circuit into four (2×2) subdomains to evaluate the MPI program. Since the stream graph does not have the processor binding issue, we further test the stream graph model with the 3×3 and 4×4 partitions to investigate possible performance improvement.

Table 6.1 lists the results of the single machine experiment. To adequately compare both models, in addition to the total execution time we also record the matrix solving time, which does not include the latency of transferring partitioned files on NFS. For the 2×2 partitions, the runtime of stream graph is only moderately higher than the MPI's and both exhibit a similar performance scale. Comparing the 2×2 partition with the 3×3 and 4×4 partitions, the performance is further improved by partitioning the circuit into smaller subdomains to reduce the matrix solving time.

Next we evaluate their performance in a cluster consisting of 9 machines, each with a 3.2 GHz quad-core CPU and 24 GB memory. We test four partition sizes: 3×3, 4×4, 5×5 and 6×6. For the sake of fairness, in the stream graph model a subdomain is assigned one CPU core so that each machine can accommodate at most four subdomains, which has the same effect as the CPU binding in the MPI model. We record the time of solving the matrix and the total runtime (including the file partition) in Table 6.2.

In all types of partitions, the matrix solving time of stream graph is close to the MPI model's for all benchmarks, and the difference does not scale with the circuit size, indicating the performance of stream graph is comparable to the MPI model.

Table 6.1: Runtime (sec) of MPI versus Stream graph on single machine

Testcase	Size	Solve Time (2x2)		Total Time (2x2)			Solve time (stream graph)		Total Time (stream graph)	
		MPI	Stream graph	MPI	Stream graph		3x3	4x4	3x3	4x4
y200	10513442	1,061.45	1,246.63	1,133.46	1,302.94		544.44	749.20	592.30	795.65
y250	6727562	628.15	717.60	676.21	754.30		262.03	266.03	294.22	296.09
y300	4688899	251.82	294.99	284.67	320.52		156.08	154.70	178.36	176.37
y400	2627442	48.72	68.38	66.82	82.49		45.96	49.61	58.14	61.25
y500	1680602	25.77	36.31	37.94	45.43		25.76	25.29	33.69	32.95
y600	1171822	12.50	18.41	20.77	24.74		18.25	15.50	23.96	20.99
y800	655896	6.44	10.71	11.07	14.51		7.82	7.49	10.97	10.68
y1000	419522	2.85	5.27	5.79	7.61		4.21	4.17	6.27	6.18

Table 6.2: Runtime (sec) of MPI versus Stream graph (ours) on a cluster with 9 machines

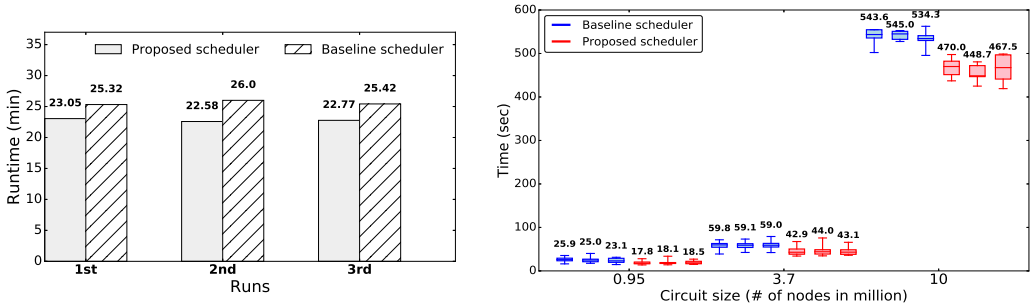
Testcase	Decomposition	Solve Time		Total Time	
		MPI	Ours	MPI	Ours
y200	6 x 6	90.60	109.12	149.42	163.40
y250	6 x 6	34.40	45.36	70.583	82.29
y300	5 x 5	22.52	28.06	43.95	52.59
y400	5 x 5	7.52	10.04	19.35	22.75
y500	4 x 4	5.21	6.77	14.79	16.87
y600	4 x 4	3.26	5.03	9.96	12.54
y800	3 x 3	2.25	3.65	7.54	7.97
y1000	3 x 3	1.10	2.58	4.41	5.63

6.5.2 Production-Mode Evaluation

The scheduler experiments are undertaken on Amazon's Elastic Compute Cloud and we use 10 EC2 instances where each instance has 4 CPUs and 16 GB memory. The first experiment is to evaluate the effectiveness of our scheduler on handling workload composed of jobs in different scales. We select three types of circuits whose power grids have 0.95, 3.7 and 10 million nodes respectively to represent jobs with small, medium and large scale. The stream graph each has 4 (small), 8 (medium) and 16 (large) worker vertices respectively. There are one hundred jobs in total and the numbers of jobs for each type are 27, 68 and 5, which are distributed normally to simulate the job composition in realistic situations. The jobs are randomly permuted and we submit a job every 10 seconds.

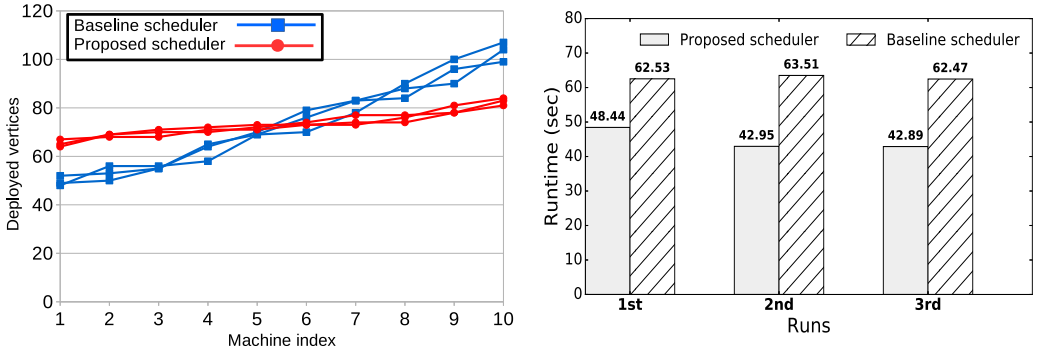
For the sake of fairness, we run the experiment three times for both the baseline scheduler and the proposed scheduler and record all results. Figure 6.3 (a) shows the total time from submitting the first job to the finish of the last job. Compared with the baseline scheduler, the proposed scheduler reduces the total time by an average of 10%, implying the runtime of each job in our scheduler has been shortened by 10%.

To understand the impact of schedulers on the runtime of each job, Figure 6.3 (b) records the distribution of completion time on jobs with different sizes. By using the proposed scheduler, the average completion time of the small, medium and large-sized jobs is decreased by 24%, 22% and 14% re-



(a) (b)

Figure 6.3: (a) The total execution time (minutes) for the three runs. (b) The runtime (seconds) distribution for the three sizes of benchmarks in all runs. The number on the top of each box is the median value, and the top and bottom whiskers represent the maximum and minimum values.



(a) (b)

Figure 6.4: (a) The number of vertices deployed on each machine in all runs. (b) The average runtime (seconds) of a benchmark for the three runs in a simulated production environment.

spectively and the improvement does not change from run to run. To know the resource usage of machines, Figure 6.4 (a) depicts the number of vertices deployed on each machine (sorted in ascending order). We observe that there exists a huge gap of the deployed vertices between machines in baseline scheduler, whereas for our scheduler the maximum difference of deployed vertices between machines is one third that of the default scheduler, implying our scheduler can effectively balance the workload to achieve better cluster resource utilization. As the baseline scheduler tends to allocate vertices to the machine with least space, it is expected that the workload distribution can be very non-uniform with some machines being overloaded while others

stay idle, leading to inefficient resource usage and slower system performance.

The next experiment is to evaluate the schedulers with jobs arriving in Poisson distribution manner. A common assumption in production environment is that the arrival time follows the Poisson distribution, where the arrival times of jobs are independent to each other and the probability of a job arriving over an interval is proportional to the interval length. Therefore, in contrast to the previous experiment where jobs are uniformly distributed along the timeline, in the Poisson distribution the job arrival rates in some intervals are higher than in others. We set the average arrival rate to 0.1 (i.e. the average arrival time of a job is 10 seconds) and submit 100 medium-sized jobs. Figure 6.4 (b) shows that the proposed scheduler's average completion time of a job is around 20% to 30% less than the default scheduler's. We observe that the number of vertices deployed on each machine can vary greatly in the default scheduler, resulting in a low resource utilization and slower performance.

6.6 Conclusion

This chapter introduces a distributed power grid analysis framework based on the stream graph programming model. The framework enables flexible power grid decomposition regardless of the available CPU cores and this feature is useful for seeking potential performance improvement. Moreover, a load-aware scheduler is proposed to balance the machine workloads and effectively promote the overall system resource utilization. The experimental results show that the framework has comparable performance to the MPI-based framework and the effectiveness of the load-aware scheduler. We believe we have opened a new direction for the distributed power grid analysis. Our idea can inspire EDA engineers to rethink the way to parallelize EDA algorithms.

REFERENCES

[1] T. Huang, C. Lin, G. Guo, and M. Wong, "Cpp-Taskflow: Fast task-based parallel programming using modern C++," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2019, pp. 974–983.

[2] Cpp-Taskflow, "https://github.com/cpp-taskflow/cpp-taskflow."

[3] C. Lin, T. Huang, G. Guo, and M. D. F. Wong, "An efficient and composable parallel task programming library," in *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, 2019, pp. 1–7.

[4] C. Lin and M. D. F. Wong, "Accelerate analytical placement with GPU: A generic approach," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2018, pp. 1345–1350.

[5] C.-X. Lin, T.-W. Huang, T. Yu, and M. D. F. Wong, "A distributed power grid analysis framework from sequential stream graph," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, ser. GLSVLSI 18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3194554.3194560 p. 183188.

[6] T. Huang, C. Lin, and M. D. F. Wong, "DtCraft: A distributed execution engine for compute-intensive applications," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2017, pp. 757–765.

[7] T.-W. Huang and M. D. F. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD 15. IEEE Press, 2015, p. 895902.

[8] OpenMP, "[online]. available: https://www.openmp.org/."

[9] Intel Threading Building Blocks, "[online]. available: https://www.threadingbuildingblocks.org/intel-tbb-tutorial."

[10] "C++ named requirements: Callable." [Online]. Available: https://en.cppreference.com/w/cpp/named_req/Callable

[11] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, no. 5, p. 720748, Sep. 1999. [Online]. Available: https://doi.org/10.1145/324133.324234

[12] D. Chase and Y. Lev, "Dynamic circular work-stealing deque," in *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA 05. New York, NY, USA: Association for Computing Machinery, 2005. [Online]. Available: https://doi.org/10.1145/1073970.1073974 pp. 21–28.

[13] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP 95. New York, NY, USA: Association for Computing Machinery, 1995. [Online]. Available: https://doi.org/10.1145/209936.209958 pp. 207–216.

[14] R. D. Blumofe and D. Papadopoulos, "The performance of work stealing in multiprogrammed environments," University of Texas at Austin, USA, Tech. Rep., 1998.

[15] X. Ding, K. Wang, P. B. Gibbons, and X. Zhang, "BWS: Balanced work stealing for time-sharing multicores," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys 12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2168836.2168873 p. 365378.

[16] N. S. Arora, R. D. Blumofe, and C. G. Plaxton, "Thread scheduling for multiprogrammed multiprocessors," in *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA 98. New York, NY, USA: Association for Computing Machinery, 1998. [Online]. Available: https://doi.org/10.1145/277651.277678 p. 119129.

[17] G. Contreras and M. Martonosi, "Characterizing and improving the performance of Intel Threading Building Blocks," in *2008 IEEE International Symposium on Workload Characterization*, Sep. 2008, pp. 57–66.

[18] K. Agrawal, Y. He, and C. E. Leiserson, "Adaptive work stealing with parallelism feedback," in *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 07. New York, NY, USA: Association for Computing Machinery, 2007. [Online]. Available: https://doi.org/10.1145/1229428.1229448 pp. 112–120.

[19] Yi Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–12.

[20] O. Tardieu, H. Wang, and H. Lin, "A work-stealing scheduler for X10's task parallelism with suspension," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2145816.2145850 pp. 267–276.

[21] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA 00. New York, NY, USA: Association for Computing Machinery, 2000. [Online]. Available: https://doi.org/10.1145/341800.341801 pp. 1–12.

[22] K. Singer, Y. Xu, and I.-T. A. Lee, "Proactive work stealing for futures," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3293883.3295735 pp. 257–271.

[23] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: A scalable locality-aware adaptive work-stealing scheduler," in *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[24] A. Kukanov and M. J. Voss, "The foundations for scalable multi-core software in Intel Threading Building Blocks," in *Intel Technology Journal*, vol. 11, Nov. 2007.

[25] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI 98. New York, NY, USA: Association for Computing Machinery, 1998. [Online]. Available: https://doi.org/10.1145/277650.277725 pp. 212–223.

[26] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick, "Deadlock-free scheduling of X10 computations with bounded resources," in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA 07. New York, NY, USA: Association for Computing Machinery, 2007. [Online]. Available: https://doi.org/10.1145/1248377.1248416 pp. 229–240.

[27] K. Agrawal, C. E. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[28] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA 09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1640089.1640106 pp. 227–242.

[29] J. B. Dogan, "Go's work-stealing scheduler," https://rakyll.org/scheduler/.

[30] H. Ribic and Y. D. Liu, "Energy-efficient work-stealing language runtimes," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2541940.2541971 p. 513528.

[31] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli, "Correct and efficient work-stealing for weak memory models," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP 13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2442516.2442524 p. 6980.

[32] "Eigen EventCount." [Online]. Available: https://eigen.tuxfamily.org/dox/unsupported/EventCount_8h_source.html

[33] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 contest on incremental timing analysis," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 882–889.

[34] "Intel 64 and IA-32 Architectures Software Developer Manuals," https://software.intel.com/en-us/articles/intel-sdm.

[35] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *ASPLOX*. ACM, 2000, pp. 234–244.

[36] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *NIPS*, 2015, pp. 2503–2511.

[37] E. Ayguadé and D. Jiménez-González, "An approach to task-based parallel programming for undergraduate students," *J. Parallel Distrib. Comput.*, vol. 118, no. P1, pp. 140–156, Aug. 2018.

[38] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, "A taxonomy of task-based parallel programming technologies for high-performance computing," *J. Supercomput.*, vol. 74, no. 4, pp. 1422–1434, Apr. 2018. [Online]. Available: https://doi.org/10.1007/s11227-018-2238-4

[39] S. Palkar, J. Thomas, D. Narayanan, P. Thaker, R. Palamuttam, P. Negi, A. Shanbhag, M. Schwarzkopf, H. Pirk, S. Amarasinghe, S. Madden, and M. Zaharia, "Evaluating end-to-end optimization for data analytics applications in Weld," *VLDB*, vol. 11, no. 9, pp. 1002–1015, 2018.

[40] The DOT Language, "https://www.graphviz.org/."

[41] A. B. Kahng and Qinke Wang, "Implementation and extensibility of an analytic placer," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 5, pp. 734–747, May 2005.

[42] C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf placement and routing package," *IEEE Journal of Solid-State Circuits*, vol. 20, no. 2, pp. 510–522, April 1985.

[43] M. C. Yildiz and P. H. Madden, "Improved cut sequences for partitioning based placement," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, June 2001, pp. 776–779.

[44] N. Viswanathan, M. Pan, and C. Chu, "FastPlace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control," in *2007 Asia and South Pacific Design Automation Conference*, Jan 2007, pp. 135–140.

[45] T.-C. Chen, T.-C. Hsu, Z.-W. Jiang, and Y.-W. Chang, "NTUplace: A ratio partitioning based placement algorithm for large-scale mixed-size designs," in *Proceedings of the 2005 International Symposium on Physical Design*, ser. ISPD'05. New York, NY, USA: Association for Computing Machinery, 2005. [Online]. Available: https://doi.org/10.1145/1055137.1055188 pp. 236–238.

[46] T. Chan, J. Cong, and K. Sze, "Multilevel generalized force-directed method for circuit placement," in *Proceedings of the 2005 International Symposium on Physical Design*, ser. ISPD'05. New York, NY, USA: Association for Computing Machinery, 2005. [Online]. Available: https://doi.org/10.1145/1055137.1055177 pp. 185–192.

[47] P. Spindler and F. M. Johannes, "Fast and robust quadratic placement combined with an exact linear net model," in *2006 IEEE/ACM International Conference on Computer Aided Design*, Nov 2006, pp. 179–186.

[48] U. Brenner, M. Struzyna, and J. Vygen, "BonnPlace: Placement of leading-edge chips by advanced combinatorial algorithms," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 27, no. 9, pp. 1607–1620, Sep. 2008. [Online]. Available: https://doi.org/10.1109/TCAD.2008.927674

[49] M. Kim, D. Lee, and I. L. Markov, "SimPL: An effective placement algorithm," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 1, pp. 50–60, Jan 2012.

[50] J. Lu, P. Chen, C. Chang, L. Sha, D. J. H. Huang, C. Teng, and C. Cheng, "ePlace: Electrostatics based placement using Nesterov's method," in *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2014, pp. 1–6.

[51] Y. Deng, B. D. Wang, and S. Mu, "Taming irregular EDA applications on GPUs," in *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*, Nov 2009, pp. 539–546.

[52] A. Al-Kawam and H. M. Harmanani, "A parallel GPU implementation of the Timber Wolf placement algorithm," in *2015 12th International Conference on Information Technology - New Generations*, April 2015, pp. 792–795.

[53] J. Cong and Y. Zou, "Parallel multi-level analytical global placement on graphics processing units," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, ser. ICCAD'09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1687399.1687525 pp. 681–688.

[54] K. Zhai, W. Yu, and H. Zhuang, "GPU-friendly floating random walk algorithm for capacitance extraction of VLSI interconnects," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2013, pp. 1661–1666.

[55] Y. Han, K. Chakraborty, and S. Roy, "A global router on GPU architecture," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, Oct 2013, pp. 78–84.

[56] W. C. Naylor, R. Donelly, and L. Sha, "Non-linear optimization system and method for wire length and delay optimization for an automatic electric circuit placer," Oct. 9 2001, US Patent 6,301,693.

[57] M.-K. Hsu, Y.-W. Chang, and V. Balabanov, "TSV-aware analytical placement for 3D IC designs," in *Proceedings of the 48th Design Automation Conference*, ser. DAC'11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2024724.2024875 pp. 664–669.

[58] Mark Harris, "How to overlap data transfers in CUDA C/C++," https://devblogs.nvidia.com/parallelforall/how-overlap-data-transfers-cuda-cc.

[59] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Comput. Surv.*, vol. 23, no. 1, pp. 5–48, Mar. 1991. [Online]. Available: https://doi.org/10.1145/103162.103163

[60] P. E. Small, R. K. Kalia, A. Nakano, and P. Vashishta, "Order-invariant real number summation: Circumventing accuracy loss for multimillion summands on multiple parallel architectures," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 152–160.

[61] I. S. Bustany, D. Chinnery, J. R. Shinnerl, and V. Yutsis, "ISPD 2015 benchmarks with fence regions and routing blockages for detailed-routing-driven placement," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, ser. ISPD'15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2717764.2723572 pp. 157–164.

[62] cuSPARSE library, https://developer.nvidia.com/cusparse.

[63] C. Wei, H. Chen, and S. Chen, "Design and implementation of block-based partitioning for parallel flip-chip power-grid analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 3, pp. 370–379, March 2012.

[64] J. W. Demmel, J. R. Gilbert, and X. S. Li, "An asynchronous parallel supernodal algorithm for sparse Gaussian elimination," *SIAM Journal on Matrix Analysis and Applications*, vol. 20, no. 4, pp. 915–952, 1999. [Online]. Available: https://doi.org/10.1137/S0895479897317685

[65] V. Y. Voronov and N. N. Popova, "Parallel power grid simulation on platforms with multi core processors," in *2009 International Conference on Computing, Engineering and Information*, April 2009, pp. 144–148.

[66] T. Yu, Z. Xiao, and M. D. F. Wong, "Efficient parallel power grid analysis via additive Schwarz method," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD 2012. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2429384.2429468 pp. 399–406.

[67] L. Grigori, J. W. Demmel, and X. S. Li, "Parallel symbolic factorization for sparse LU with static pivoting," *SIAM Journal on Scientific Computing*, vol. 29, no. 3, pp. 1289–1314, 2007. [Online]. Available: https://doi.org/10.1137/050638102

[68] Q. He, W. Au, A. Korobkov, and S. Venkateswaran, "Parallel power grid analysis using distributed direct linear solver," in *2014 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, Aug 2014, pp. 866–871.

[69] "MPICH." [Online]. Available: https://www.mpich.org/

[70] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012. [Online]. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia pp. 15–28.

[71] T. Huang, M. D. F. Wong, D. Sinha, K. Kalafala, and N. Venkateswaran, "A distributed timing analysis framework for large designs," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016, pp. 1–6.

[72] Kai Sun, Quming Zhou, Kartik Mohanram, and D. C. Sorensen, "Parallel domain decomposition for simulation of large-scale power grids," in *2007 IEEE/ACM International Conference on Computer-Aided Design*, Nov 2007, pp. 54–59.

[73] "PETSC." [Online]. Available: http://www.mcs.anl.gov/petsc/

[74] P. Sun, X. Li, and M. Ting, "Efficient incremental analysis of on-chip power grid via sparse approximation," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2011, pp. 676–681.

[75] S. R. Nassif, "Power grid analysis benchmarks," in *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC 2008. Washington, DC, USA: IEEE Computer Society Press, 2008, pp. 376–381.