Developing Graph-based Co-scheduling Algorithms with GPU Acceleration

by

Huanzhou Zhu

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

PhD

Department of Computer Science

The University of Warwick

May 2016

Abstract

On-chip cache is often shared between processes that run concurrently on different cores of the same processor. Resource contention of this type causes the performance degradation to the co-running processes. Contention-aware co-scheduling refers to the class of scheduling techniques to reduce the performance degradation. Most existing contention-aware co-schedulers only consider serial jobs. However, there often exist both parallel and serial jobs in computing systems. This thesis aims to tackle these issues. We start with modeling the problem of co-scheduling the mix of serial and parallel jobs as an Integer Programming (IP) problem. Then we construct a co-scheduling graph to model the problem, and a set of algorithms are developed to find both optimal and near-optimal solutions. The results show that the proposed algorithms can find the optimal co-scheduling solution and that the proposed approximation technique is able to find the near optimal solutions. In order to improve the scalability of the algorithms, we use GPU to accelerate the solving process. A graph processing framework, called WolfPath, is proposed in this thesis. By taking advantage of the co-scheduling graph, WolfPath achieves significant performance improvement. Due to the long preprocessing time of WolfPath, we developed WolfGraph, a GPU-based graph processing framework that features minimal preprocessing time and uses the hard disk as a memory extension to solve large-scale graphs on a single machine equipped with a GPU device. Comparing with existing GPU-based graph processing frameworks, WolfGraph can achieve similar execution time but with minimal preprocessing time.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor Dr. Ligang He, whose guidance, encouragement and support have been invaluable to me during my time at the Department of Computer Science at the University of Warwick. I would like to thank you for encouraging my research and for allowing me to grow as a research scientist. Your advice on my research have been priceless.

It is my great pleasure to work with all my lab-mates, particularly Bo Gao Chao Chen, Zhuoer Gu, Peng Jiang, Shenyuan Ren, Xufeng Lin, Chen Gu, Matthew Bradbury, Stephen Roberts and Nentawe Gurumdimma, for their stimulating discussions in science and technology and for all the fun we have had in the last four years.

I would like to thank all my friends, especially Wolfy Wang, Zihao Chen, Qiwei Jin, Yuxuan Zhan, Ye Kuang, Hui Xu, Dake Xu,Yibing Chen.

I would also like to say a heartfelt thank you to my parents for always believing in me and supporting me and encouraging me in all my pursuits.

In the end, I would like to give my special thanks to Momofuku Ando, the inventor of instant noodles. This invention saves me uncountable amount of time and liberates me from thinking one of the biggest problem in my life: what to eat. Without instant noodles, I am very likely died of the starvation long before I complete my PhD.

Declarations

Parts of this thesis have been previously published by the author in the following:

- Huanzhou Zhu, Ligang He, Stephen. A. Jarvis Optimizing job scheduling on multicore computers *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems 2014 (MASCOTS'2014)* [149]

- Huanzhou Zhu, Ligang He, Bo Gao, Kenli Li, Jianhua Sun, Hao Chen, Keqin Li Modelling and Developing Co-scheduling Strategies on Multicore Processors *2015 International Conference on Parallel Processing (ICPP'2015).* [150]

- Ligang He, Huanzhou Zhu, Stephen Jarvis Developing graph-based co-scheduling algorithms on multicore computers *IEEE Transactions on Parallel and Distributed Systems (TPDS).* [52]

In addition, the following works are in progress of being submitted:

- "WolfPath: Accelerating iterative graph searching algorithms on GPUs"

- "WolfGraph: Wolf Graph: Edge-Centric graph processing on GPUs"

Abbreviations

BFS	Breadth First Search
BSP	Bulk Synchronous Parallel
CPI	Cycles Per Instruction
CMR	Cache Miss Rate
CW	Concatenated Window
CEL	Concatenated Edge List
CSR	Compressed Sparse Row
CDF	Cumulative Distribution Function
DI	Distributed Intensity
EVM	Edge Vertex Message
GAS	Gather Apply Scatter
IP	Integer Programming
LLC	Last Level Cache
LPD	Least Performance Degradation
MRC	Miss Rate Curves
MPP	Multi Process Parallel
MTP	Multi Thread Parallel
MER	Maximum Effective Rank
O-SVP	Optimised Shortest Valid Path
PSW	Parallel Sliding Window
PE	Embarrassingly Parallel
PC	Process Communication

RDD	Resilient Distributed Dataset
RDG	Resilient Distributed Graph
SSSP	Single Source Shortest Path
SDP	Stack Distance Profile
SVP	Shortest Valid Path
SVPPE	Shortest Valid Path for Embarrassingly Parallel
SVPPC	Shortest Valid Path for Process Communication
SM	Streaming Multiprocessor
VWC	Virtual Warp Centric

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Multicore processors have now become a mainstream product in CPU industry. In a multicore processor, multiple cores reside and share the resources on the same chip. However, running multiple applications on different cores could cause resource contention, which leads to performance degradation. Many studies have shown that it is possible to isolate some resources, such as disk bandwidth [142] [103], network bandwidth [47] [24] for the co-running jobs. However, it is very difficult to isolate the on-chip last level cache (LLC). This problem is known as the shared cache contention and has been studied extensively in the literature [71], [61], [152], [21]. The existing approaches to addressing on-chip shared cache contention fall into three categories: 1) Architecture-level solutions to improve hardware and provide isolation among threads[102] [121] [68], 2) System-level solutions to partition the cache for each application[143] [79] [69] [20], and 3) Software-level solutions to develop schedulers to reduce the contention [35] [38] [92] [60]. In these three categories, the architecture-level solution is still under active development by the processor vendors. The cache partitioning solution requires many changes in the existing system-level software (such as operating system) and therefore incurs high implementation cost. The third approach, developing contention-aware schedulers, is a fairly lightweight approach, and therefore attracts many researchers' attention, which is also the focus of this work.

1.1 Contention aware co-scheduling

Existing studies of the co-scheduling problem can be classified into two categories. Research in the first category aims at developing practical job scheduling systems that produce solutions on a best effort basis [127] [132], [153], [12],[51] [137]. Algorithms developed in this category are often heuristics-based in order to reduce computation cost. The work in the second category aims to develop the algorithms to either compute or approximate the optimal co-scheduling strategy (referred to as the *optimal co-scheduling problem* in the rest of this thesis) [28] [128] [59]. Due to the NP-hard nature [61] of this class of problems, obtaining an optimal solution is often a computation-expensive process and is typically performed offline. Although an optimal solution is not suitable for direct uses in online job scheduling systems, its solution provides the engineer with a unique insight into how much performance can be extracted if the system were best tuned. Additionally, knowing the gap between current and optimal performance can help the scheduler designers to weight the trade-off between efficiency and quality.

There are some research studies in contention-aware co-scheduling [11][38]. To the best of our knowledge, the existing methods that aim to find the optimal co-scheduling only consider the serial jobs [61]. The work in [61] modelled the optimal co-scheduling problem for serial jobs as an integer programming problem. However, there typically exists both serial and parallel jobs in the computing systems, such as clusters and clouds [139], [32], [114].

Some conventional systems disallow co-scheduling of parallel jobs and serial jobs in the same multicore machine. Take PBS [123] as an example. Although all jobs can share the same node in PBS (that is, parallel and serial jobs will share nodes without any configuration to the batch system), some sites decide to configure the PBS in the way that this does not happen. The main purpose for this is to avoid the performance interference between different types of jobs since the performance degradation caused by interference is either unknown or its prediction is inaccurate.

This also happens in some data centers (clouds). When a task is submitted to a data center, a configuration file for the task is created to specify the rules for the cluster manager to run the task. By default, data centers do not restrict the sharing of the same node between different jobs. But the users can disallow co-locate this task with other tasks in the task configuration file[95]. However, disallowing job co-locations through either configuration or resource reservation is a major contributor to low machine utilization. For example, in the production clusters at Twitter with thousands of servers, managed by Mesos [55], the CPU utilization is consistently below 20%, and a 12,000-server Google cluster managed by Borg consistently achieves CPU utilization of 25-35% [32].

In order to improve resource utilization, it is necessary to consider both parallel jobs and serial jobs when designing co-scheduling systems. However, the existing methods of finding the optimal co-scheduling cannot be applied to the system that contains both parallel and serial jobs.

We use Figure 1.1 to illustrate why different considerations should be taken when co-scheduling parallel jobs. The figure considers two co-scheduling scenarios. In Figure 1a, 4 serial jobs (i.e., 4 processes p_1, ..., p_4) are co-scheduled on two dual-core nodes, while in Figure 1b a serial job (p_4) and a parallel job with 3 processes (i.e., p_1, p_2 and p_3) are co-scheduled. D_i drawn above p_i is the degradation of p_i in the co-scheduling solution. The arrows between p_1 and p_2 as well as between p_2 and p_3 represent the interactions between the parallel processes. In Figure 1a, the objective is to minimize the sum of the performance degradation suffered by each process (i.e.,$D_1 + D_2 + D_3 + D_4$). In Figure 1b, the performance degradation (i.e., increased execution time) of the parallel job is dependent on the processes that has been affected the most and therefore completing the execution last. Therefore, the performance degradation of the parallel job should be computed by $max(D_1, D_2, D_3)$. The objective in Figure

1b is to find the co-scheduling solution that minimizes $max(D_1, D_2, D_3) + D_4$.

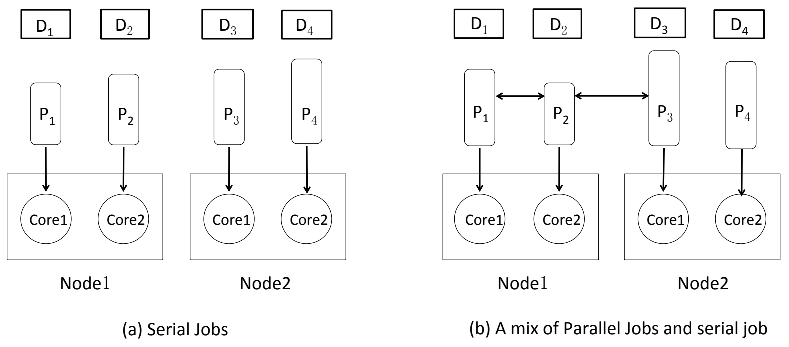


Figure 1.1: An illustration example for the difference between serial and parallel jobs in calculating the performance degradation

In order to address this problem, this thesis proposes a graph-based method to find the optimal co-scheduling solution for a mix of serial and parallel jobs. Two types of parallel job are considered in this thesis: Multi-Process Parallel (MPP) jobs, such as the jobs written in MPI (Message Passing Interface), and Multi-Thread Parallel (MTP) jobs, such as the jobs written in OpenMP.

The graph model presented in this thesis aims to provide theoretical insights into co-scheduling problems. That is, the optimal result founded by our model is used to compare with the scheduling results generated from other scheduling systems. However, due to the high complexity of finding the optimal scheduling solution, this process is timing consuming. Thus, the solving process of the graph model is performed offline. We argue that accelerating the solving process can relieve the model user from the long waiting time. Therefore, in this thesis, we also explore the possibility of accelerating the solving process by using GPU, this is because compare to other parallel processing platforms, GPU has the following advantages: first, GPU can bring high parallelism degree and performance to a single machine. Second, it is more energy-efficient than distributed systems.

1.2 Parallel Graph Processing

Many studies have shown that the graph theory can be used to solve scheduling problems [74], [130] [75] [28] [126] [25]. However, as shown in [150] [52] [149], when modelling optimal scheduling problem into graph problem, the graph size increases exponentially. For example, when using the method presented in [52] to schedule 64 jobs to 16 quad-core machines, the graph size is 16,777,216 nodes and 3,450,553,937 edges. It is challenging to process graphs of such a scale.

The demand for processing large-scale graphs efficiently has been growing recently. This is because graphs can be used to describe a wide range of objects, and computations on graph-based data structures are the core of many applications. Motivated by the need to process very large graphs, many frameworks have been developed for processing large graphs on distributed systems. Such frameworks include GPS [113] MocGraph [147] Trinity [116] Chaos [110] Chronos [48] Gram [138]. However, since developing distributed graph algorithms is challenging, some research studies aim to design the graph processing systems that can handle large graphs (with billions of edges) on a single PC. The results of these studies are FlashGraph [145], PathGraph [140], GraphQ[133], LLAMA [90], Ligra [118][119], Ringo [105]. However, these systems suffer from the limited degree of parallelism provided by conventional processors. To overcome this problem, a lot of research employs GPU to accelerate graph processing due to its massively parallel architecture, such as Medusa[146], Gunrock [135], CuSha [65] and MapGraph [40] [66] [117].

However, using GPU for efficient graph processing remains a challenging and open problem due to the following reasons. First, although GPU provides a massive degree of parallelism compared to CPU, its hardware architecture requires regular data access pattern to achieve the peak performance. However, most graphs are of highly irregular structure, which leads to the problems of irregular memory accesses and underutilization of GPU and consequently limits the performance of graph algorithms on GPU. For example, most existing graph processing techniques [50] [29] [15] [82] employ the vertex-centric processing and rely on the CSR (Compressed Sparse Row) representation of graphs. Due to the poor locality in the CSR representation, however, visiting a node's neighbours usually leads to random memory accesses, which is known as non-coalesced accesses. Second, because GPU has the limited global memory space compared with CPU, it requires frequent data copying between device memory and host memory, which also results in poor performance. Third, as pointed out by Guo et al. [46], in the state-of-the-art GPU-based graph processing systems, the time spent in reading the graph from hard disk to memory and in constructing the data structure in memory constitute a big proportion of the total processing time for a large graph. Reducing this pre-processing time will significantly improve the overall performance of graph processing frameworks. Finally, Existing work of GPU-based graph processing assumes the entire graph can fit into the global memory of GPU. However, there are large-scale graphs that are even bigger than the GPU memory, which makes these works infeasible to solve massive scale graph problems. Lack of support for large-scale graphs beyond the capacity of device memory is pointed out as one of the most critical problems of the existing graph processing methods using GPUs.

1.3 Research Contributions

In this thesis, the problem of finding the optimal co-scheduling for a mix of serial and parallel jobs is modelled as an Integer Programming (IP) problem first. Then the existing IP solvers can be used to find the optimal co-scheduling solution that minimizes the performance degradation. However, the IP-based method suffers from long processing time and poor scalability. Therefore, a graph-based method is further proposed in this thesis. A layered graph is constructed to model the co-scheduling problem on single processor machines. The problem of finding the optimal co-scheduling solutions is then modelled as finding the shortest VALID path in the graph. Moreover, this thesis develops a set of algorithms to find the shortest valid path for both serial and parallel jobs. A number of optimization measures are also developed to increase the efficiency of these proposed algorithms (i.e., accelerate the solving process of finding the optimal co-scheduling solution). After these, the graph model and proposed algorithms are extended to co-schedule a mix of serial and parallel jobs on multi-processor machines.

Moreover, it has been shown that the A*-search algorithm can effectively avoid the unnecessary searches when finding the optimal solution. In this thesis, an A*-search-based algorithm is developed to combine the ability of the A*-search algorithm and the proposed optimization measures in terms of accelerating the solving process. Further, a heuristic method, called heuristic A*-search, is developed to find the near-optimal solutions efficiently. Finally, a flexible approximation technique is proposed so that we can control the scheduling efficiency by setting the requirement for the solution quality.

In order to accelerate the processing speed of our co-scheduling algorithms, a GPU-based graph processing framework called WolfPath is designed to accelerate the processing of the co-scheduling graph, by exploiting the special structure of the co-scheduling graph. WolfPath can also be used to process general graphs by adding the pre-processing steps to convert a general graph into a graph with similar graph structure as in the co-scheduling graph. The design of WolfPath concentrates on addressing the non-coalesced memory access to graph edges and frequent data exchange between GPU and CPU memories. In addition, by taking advantage of the graph structure, WolfPath is also able to process large graphs that cannot fit in the GPU memory.

Due to the long pre-processing time of WolfPath, WolfGraph is finally proposed in this thesis. By using the iterative graph computation model and carefully designed data structure, WolfGraph requires the minimum graph pre-processing. With carefully designed graph structure, WolfGraph requires less pre-processing time than other systems. A two-level thread synchronisation scheme, first in shared memory and then in global memory, is designed in Wolf-

Graph, which proves to be faster than the synchronisation in global memory only. WolfGraph can work as a general purpose graph processing framework that can accelerate any graph algorithms. In addition, by adding the support for the secondary storage, WolfGraph can handle the graphs that are even larger than the host memory.

1.4 Thesis Organisation

In Chapter 1, we discuss the motivations of the research presented in this thesis and outline the main research contributions. In Chapter 2.2, we describe two important components in designing co-scheduling systems: performance prediction and scheduling strategy. In Chapter 2.3, we discuss the challenges of processing large scale graphs and review some state-of-the-art techniques developed in graph process systems.

In Chapter 3, we extend the existing work from co-scheduling serial jobs to co-scheduling a mix of serial and parallel jobs. In Chapter 4, we use GPU to parallelise the algorithms proposed in Chapter 3, and extend the algorithms to a GPU-based graph processing framework, WolfPath. Chapter 5 presents WolfGraph, a general purpose GPU-based graph proposing framework that aims to reduce the graph pre-processing and support both GPU acceleration and secondary storage device. Finally, Chapter 6 concludes the thesis and discusses the future work.

Chapter 2

Literature Review

2.1 Introduction

In multicore processor architectures, multiple cores reside on a chip are not fully independent but share resources such as caches and memory controllers with neighbouring cores. These shared resources are managed exclusively in hardware and are job-unaware. They treat the requests from different jobs running on different cores as if they were all requests from one single source. This means that they do not enforce any fairness or partitioning when different jobs use the resources. This is resulting in performance degradation due to competition for shared resources. Some researchers observed that an application could slow down by multiple folds if it shares resources with processes running on neighbouring cores, comparing with it running alone.

There has been significant interest in the research community in addressing shared resource contention on multicore processors. The majority of work required modifications to hardware and falls into one of two classes: performance aware cache modification [107] [58] [122] [120] [83] or performance-aware DRAM controller memory scheduling [84] [125] [57] [101]. These proposed solutions require changes to the hardware, major changes to the operating system, or both. As such, the majority of these techniques have only been evaluated in simulation and, as of this writing; none of these promising solutions have yet been implemented in commercial systems.

Another research trend tends to deal with shared resource contention on the level of job scheduling [59] [134] [36] [11]. In this context job scheduling refers to mapping jobs to the cores of the multicore processors. Different mappings result in different combinations of jobs competing for shared resources. Some job combinations compete less aggressively for shared resources than others. Contention mitigation via job scheduling aims to find the job mappings that lead to the best possible performance.

The most common objective optimized by contention-aware schedulers is overall throughput. These schedulers typically aim to reduce the contention of jobs with high resource usage and thus lower the overall performance degradation for the workload. In designing such schedulers, there are two very important aspects that need to be considered: performance prediction and co-scheduling strategies. The performance prediction is used to estimate the performance when multiple jobs co-run together. The co-scheduling strategy makes the co-scheduling decision based on the information acquired by the prediction. Existing co-scheduling strategies can be classified into two categories. Researches in the first category aims at developing practical job scheduling systems that produce solutions based on heuristics algorithms [127] [132], [153], [12],[51] [137]. The work in the second category aims to develop the algorithms to either compute or approximate the optimal co-scheduling strategy [28] [128] [59].

As discussed in the last chapter, in this thesis, the GPU has been used to accelerate the solving process of the algorithms we proposed. Hence, in this chapter, we also reviewed the existing work in the literature on the graph processing techniques.

Most existing graph processing techniques can be classified into the following three categories: the first category is distributed systems. Such frameworks include Pregel[93], GraphLab[85], PowerGraph[43], GraphX [44]. However, developing distributed graph algorithms is challenging, some research studies aim to design the graph processing systems that can handle large graphs (with billions of edges) on a single PC. The results of these studies lead to the second category, which is single machine shared memory graph processing systems, such as PathGraph [140], GraphQ [133], LLAMA [90] and GridGraph [151]. The third category graph processing technique is heterogeneous processing systems. In these systems, the accelerators (e.g., GPU, FPGA) are used to overcome the limited degree of parallelism provided by conventional processors. The results of these studies are Medusa[146], Gunrock [135], CuSha [65] and MapGraph [40].

This chapter contains two parts: the first part concentrates on reviewing the co-scheduling technologies, and the second part focuses on reviewing the graph processing systems. In the first part, we first discuss the existing performance prediction technologies in Section 2.2.2. Then we review the current co-scheduling strategies in Section 2.2.3. In the second part, we first discuss the challenges faced by graph processing systems designer in Section . We review the existing distributed graph systems in Section 2.3.2. We discuss the single machine graph processing systems in Section 2.3.3. In Section 2.4, we discuss the state-of-art GPU accelerated graph processing systems. We summarise and conclude this chapter in Section 2.5.

2.2 Job Co-scheduling

2.2.1 Overview of Co-Scheduling Problems

Job co-scheduling has been used to address the contention problem in multi-core systems. Its strategy is to assign jobs to cores in a way that the overall degradation is minimized. Normally, the input to the co-scheduling system is the performance of a job when it co-runs with other jobs. The output is the suitable schedules decision determined by the co-scheduling strategy.

The scheduling decision produced by the co-scheduling strategy has a huge impact on the performance of running jobs. The experiments conducted by Jiang in [61] can be used to demonstrate the effect of different scheduling strategies.

The experiments conducted in [61] compare four different scheduling strategies, the optimal strategy; two well designed hierarchical strategies (hierarchical perfect matching and greedy algorithms) and a random strategy. The experiment results show that the optimal strategy degrades the overall average performance by 5.14%. The hierarchical perfect matching algorithm reduces the average degradation to 5.21%, whereas the greedy algorithm reduces it to 4.51%. The schedules produced by the random strategy causes more than 20% average performance degradation, which is about 300% worse than the optimal strategy. On the other hand, the schedules produced by the two approximation algorithms have 1.4% and 0.7% more degradations than the optimal strategy on average.

The above example demonstrates the importance of co-scheduling strategy and how it can affect the performance of co-running jobs. It also shows the needs for developing the optimal strategy. As used in the above example, the result produced by the optimal co-scheduling can be used to benchmark other scheduling algorithms.

2.2.2 Performance Prediction

In order to make co-scheduling decisions, it is important to know the performance when multiple jobs co-run together. However, the search space is too big to benchmark all job combinations beforehand. Consider a system with two quad core CPUs, where the last level cache (LLC) is shared among all four cores on each CPU. There are $8! = 40,320$ ways to map 8 jobs onto the 8 available cores. Many of these mappings are redundant. For example, if cores 0,1,2 and 3 share a LLC, then in all the mappings of four jobs, A, B, C and D, to 4 cores, the mappings {(A, 0), (B, 1), (C, 2), (D, 3)} and {(A, 1), (B, 3), (C, 2), (D, 0)} are equivalent in terms of performance. Nevertheless, there are still $\binom{8}{4} = 70$ performance-unique mappings. For a cluster with thousands of nodes, the huge number of mappings makes it infeasible to benchmark them all to find optimised co-scheduling decisions. As such, it is very helpful to predict the performance of different mappings without actually benchmarking all possible mapping combinations.

Many different techniques have been proposed for modelling performance degradation when the applications share resources on multicore systems [136], [16] [34] [148], [68], [36], [144], [19] [30] [129] [111]. The majority of the work focuses on sharing the LLC, which many researchers believed was the primary source of contention. The best known technique uses Stack Distance Profiles (SDP) and Miss Rate Curves (MRC) to predict the performance of multiple processes sharing the LLC. SDPs were first proposed by Mattson [96] and first used for the prediction purposes by [19]. The two techniques are a concise description of the memory reuse patterns of an application and a measure of the benefit derived from additional cache space.

In Chandra's method, SDP is used to record the hits and misses of each cache line when each process is running alone. The SDC model tries to construct a new SDP that merges the separate SDPs of individual processes that are to be co-running together. This model relies on the intuition that a process that reuses its cache lines more frequently will occupy more cache space than other processes. Based on this, the SDC model examines the cache hit count of each process' stack distance position. For each position, the process with the highest cache hit count is selected and copied into the merged profile. After the last position, the effective cache space for each process is computed based on the number of stack distance counters in the merged profile.

While SDPs were shown to be an effective tool for modelling performance degradation in the LLC, the main limitation of this method is that it is difficult to obtain the SDP information online on current systems. One approach to get around the need of using SDP is to approximate the cache occupancy of competing processes by using LLC misses and access rate, which can be easily collected by using the hardware performance counter available in most modern processors. In [7], they predict the performance of a process in a mapping by calculating the cache occupancy ratio first, which is the ratio of the LLC access rate of this process to the LLC access rate of all the processes that share the same LLC cache. Then, they calculate the LLC miss rate that this process should experience under the measured LLC miss rate and the calculated cache occupancy ratio. A linear regression model is then used to predict the CPI (Cycles Per Instruction) from the predicted LLC miss rate and the measured L1 miss rate.

However, this method is rather complex. The recently proposed co-schedulers are trying to mitigate the problem. The schedulers proposed by [70], [37] [152] approximate performance degradation with the LLC miss rate only. Based on the observation and experiments, they showed that the applications that suffer from high LLC cache miss rate would suffer from high performance degradation. In addition, they also found that the jobs with high LLC cache miss would stress the entire memory hierarchy, and therefore these applications should not be scheduled together.

2.2.3 Co-scheduling strategies

Many co-scheduling schemes have been proposed to reduce the shared cache contention in a multi-core processor. Different metrics can be used to indicate the resource contention, such as Cache Miss Rate (CMR), overuse of memory bandwidth, and performance degradation of co-running jobs. These schemes fall into the following two classes.

The first class of co-scheduling schemes aims at improving the runtime schedulers and providing online scheduling solutions. The work in in [11, 37, 152] proposed a decision mechanism called distributed intensity (DI). The mechanism first sorts all jobs to be scheduled based on their miss rates. It then begins pairing applications from both ends of the list. The application with a higher miss rate is paired with one with a lower rate. This process is conducted every time when a new job enters the queue, a job terminates or in every predefined time period.

The work in [70] proposes the decision mechanisms called OBS-L, OBS-X and OBS-C. OBS-L attempts to reduce cache interference by spreading the total misses across all cache groups (a cache group consists of the shared LLC and the cores attached to it). Whenever a core becomes available, it selects a job whose miss rate is most complementary to other jobs that are sharing the cache. The OBS-X works by adding new jobs to the cache-group that has the smallest total miss rate. In addition, by monitoring the system workload at real time, it moves the jobs with the highest miss rate from the cache-group with the highest total miss rate to the cache-group with the lowest miss rate, so that the miss rate can be spread more evenly. Finally, OBS-C is based on the observation when pairing cache-heavy and cache-light jobs together, and the cache-light jobs tend to suffer less due to such pairing. Based on this observation, OBS-C extends the time slices of cache-light jobs, similar to the idea proposed by [36].

Based on the activity vectors [97], the scheduler presented in [98] makes the decision in the following way: A job's activity vector records its usage of system resources during the previous time slice, such as memory-bus, the LLC and the rest of the core. This usage is normalized to the theoretical maximum usage. The proposed scheduler performs the thread migrations, so that the jobs with complementary activity vectors can be co-scheduled together. They formalize this concept by measuring the variability of the activity vectors of jobs within the run queue. Higher variability indicates that the current jobs will yield high performance if being co-scheduled together. Another decision mechanism they proposed is called sorted co-scheduling. It works by grouping cores into pairs and attempts to schedule only complementary jobs on each core within the pair.

The work in [134] demonstrated that rearranging the scheduling order of input jobs can reduce the cache contention. In their work, by monitoring the cache miss of running jobs on each core and the jobs in the run queue, they rearrange the order of the jobs in run queue periodically. The purpose of their scheduler is to co-scheduler jobs with complementary cache miss rate together, therefore, if the job current running has a high cache miss, the job with low cache miss rate will be move to the front of the run queue and verse vice. Hence, the complementary jobs will be co-scheduled together.

The work discussed above only considers the co-scheduling of serial jobs. In some cluster systems managed by conventional cluster management software such as PBS, the systems are configured in the way that parallel and serial jobs cannot share different cores on the same chip. This happens too in some data centres, where when a user submits a job, s/he can specify in the job's configuration file the rule of disallowing the co-scheduling of this job with other jobs on different cores of the same chip [95]. The main purpose of doing these is to avoid the performance interference between different types of jobs. However, disallowing the co-scheduling of parallel and serial jobs causes very poor resource utilization, especially as the number of cores in multicore machines increases.

Coupling with the support of accurate performance predictions, some popular cluster management systems [55] [95] [32] have been developed to co-schedule different types of jobs, including parallel jobs and serial jobs, to improve resource utilization. For example, The work in [95] presents a characterization methodology called Bubble-Up to enable the accurate prediction of performance degradation due to interference in data centres. The work has conducted the experiments with the real-world large-scale applications in Googles production data centres. The results demonstrated that the proposed prediction methodology can predict the performance interference between co-locating applications with an accuracy within 1% to 2% of the actual performance degradation.

The work in [32] applies the classification techniques to accurately determine the impact of interference on performance for each job. A cluster management system called Quasar is then developed to increase resource utilization in data centres through co-scheduling. Quasar can make better resource allocation (i.e., allocating the right amount of resources for each job) and resource assignment (i.e., selecting the specific servers and cores that will satisfy a given resource allocation) decisions to mitigate the interference between co-located workloads. The applications managed by Quasar include multi-server parallel jobs, such as Hadoop, Storm and Spark, and single-server jobs such as PARSEC, SPLASH-2, BioParallel. A single-server job can be run on a single core or multiple cores using multi-threading. Quasar co-schedules parallel jobs and single-server jobs and uses the single-server jobs to fill any cluster capacity unused by parallel jobs.

Mesos [55] is a platform for sharing commodity clusters between multiple diverse cluster management frameworks, such as Hadoop [10], Torque [123], Spark [141] and etc, aiming to improve cluster utilization. Mesos has been used in production clusters such as at Twitter. In Mesos, the tasks from different cluster management frameworks (e.g., MPI jobs or serial jobs submitted to Torque and MapReduce jobs submitted to Hadoop) can be co-located in the same multicore server unless esoteric interdependencies between frameworks explicitly require that certain tasks from two frameworks cannot be co-located. The authors also pointed out that the situation of esoteric interdependencies is rare in practice.

The second class of co-scheduling schemes focuses on providing the basis for conducting performance analysis. It mainly aims to find the optimal co-scheduling performance offline, in order to provide a performance target for other co-scheduling systems. The extensive research is conducted in [61] to find the co-scheduling solutions. The work models the co-scheduling problem for serial jobs as an Integer Programming (IP) problem, and then uses the existing IP solver to find the optimal co-scheduling solution. It also applies the perfect matching algorithm to find the optimal co-scheduling solution on 2-core processors. When the processor has more than 2-core (e.g., 4-core), a hierarchical perfect matching is proposed to first use perfect matching to schedule the jobs on two groups of cores. Then each group is further partitioned into two equal-sized subgroups of virtual cores and the perfect matching is applied again on these two subgroups. This procedure repeats until each subgroup contains only one physical core. They also proposes a greedy algorithm, which takes the following steps to find a good co-scheduling solution: 1) obtaining the set of all possible u-cardinality sets, 2) computing the politeness of a job, which is defined as the reciprocal of the sum of the degradations of all u-cardinality sets that contains that job, 3) in each round of scheduling, adding into the final co-scheduling solution the u-cardinality that satisfies these two conditions: i) it contains the job whose politeness is the smallest in the unallocated jobs; ii) its total degradation is smallest.

The co-scheduling studies in the [61] only consider the serial jobs and mainly apply the heuristic approach to find the solutions. Although it can obtain the optimal co-scheduling solution, it is only for serial jobs.

The work presented in this thesis falls into the second class. In this thesis, a new graph based method is developed to find the optimal co-scheduling solution offline for both serial and parallel jobs. Moreover, this thesis develops a set of approximation methods to find the near-optimal co-scheduling solution with lower time complexity. In this work, we choose SDC model to predicate the performance degradation of the co-running jobs. This is because although LLC cache miss rate can be easily obtained using the hardware counter and this value can be used to guide the scheduling, it is unable to show how much degradation a job will suffer when it is co-running with other jobs. On the other hand, as discussed above, the SDC model can predicate the performance degradation value. In addition, our scheduling method works offline. Therefore, obtaining SDP offline is not an issue in our work.

2.3 Graph processing systems

In this section, we first summarise the challenge in parallel graph processing, and then review the state-of-the-art graph processing systems proposed in the literature.

2.3.1 Challenges in Graph Processing

The problems that are abstracted as graphs have some characteristics that make efficient parallelisation nontrivial. As stated in [88] [53], processing graph in parallel suffers from data-driven computation, irregular and unstructured data structure, poor locality and high data access to computation ratio. We discuss these characteristics in detail in the following.

- **Data Driven computation** Graph computation is completely driven by the graph data. That being said, when executing a graph algorithm, the computation is dictated by the vertex and edge structure of the graph rather than being directly expressed by the code. As stated in [87] and [45], the performance of graph processing algorithms/frameworks are highly dependent on the graph structure. Hence how to design a graph processing system that can produce the same performance on different graph structures remains an open question. In addition, another problem caused by the data driven computation is that it is difficult to parallelise the graph algorithm according to its computational structure. This is because the structure of computation in the algorithm is not known beforehand.

- **Irregular and Unstructured data structure** The data in graph problems are typically highly irregular and unstructured. When processing graph in parallel, the graph data need to be partitioned first. Then, depending on the platform, the partitioned workload will be assigned to each compute node, process or thread. However, the irregular graph structure makes it difficult to evenly partition the data. The workload balance between different computation units cannot be achieved with unevenly partitioned data, which limits the scalability of the target system.

- **Poor locality** Graphs are used to represent the relationships between objects. However, the relationship between these objects may be irregular and unstructured. Therefore, the locality of computations and data access patterns is poor. On the modern processors, locality is the key to achieve high performance [94] [67] [33]. Thus, it is hard to achieve high performance for graph algorithms, especially for the graph computation requiring to access secondary storage devices, because random access to the hard disk is very slow compared to other components in computers.

Therefore, how to organise the graph data and design the corresponding computation model to improve the data locality is the key to improve the graph processing performance.

- **High data access to computation ratio** Most graph computation algorithms are based on the iterative computation model. In each iteration, the algorithms need to traverse the graph structure to fetch the data required by the computation. Most graph algorithms are computation light [64] and their memory access to computation ratio is high. As discussed before, the locality of the graph data is poor. Hence, the runtime is dominated by the wait for the memory access.

In next three sections, we review some well known graph processing algorithms and frameworks on different architectures, and discuss how these solutions tackle the challenges mentioned in this section.

2.3.2 Distributed Graph Processing System

The most widespread class of parallel machines are distributed systems. A distributed system consists of multiple processing units where each unit has its own private memory. Data is partitioned over the separate nodes and explicit communication (e.g. message passing) is required to synchronize computation. Scaling out refers to adding more processing units to the system [100]. With cloud computing this type of scaling is available through Infrastructure as a Service (IaaS) [4].

Recently, many MapReduce [31] based general purpose data processing framework such as [17] [10] [141], and graph processing frameworks such as Pegasus [62] and GPS [113], started exploiting efficient parallel processing of large volumes of data on the distributed systems. In this section, we review some well known distributed graph processing systems proposed in the literature.

Pregel [93] is a distributed graph processing system developed by Google. The design of its vertex centric model is based on BSP (bulk synchronous

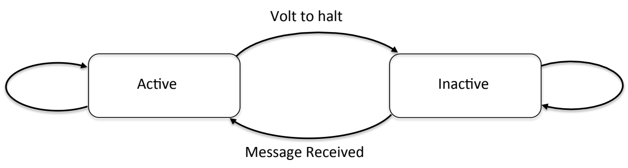


Figure 2.1: The Vertex State Machine in Pregel

parallel)[131]. Pregel partitions a graph based on vertices. Each graph partition contains a set of vertices and the edges associated with these vertices. Each partition is processed by one compute node in the system. In Pregel, the user defines the computation function. And this computation function will be invoked iteratively until the final result is found.

The iterative process in Pregel is performed as follows: in each iteration (or a superstep in Pregel's term), each active vertex (the vertex that needs to be computed) invokes the computation function. In the computation function, the vertex first reads the message sent from by its neighbours in the last iteration, computes the updated value based on the message received, and then sends the updated value to its neighbours through its outgoing edges. Its neighbours will receive this value as a message in the next iteration. Once a vertex sends the message, its state will change to inactive. An inactive vertex will become active when it receives a new message. The whole process terminates when there is no active vertex in the system. This process is called Vertex State Machine in Pregel and is illustrated in Figure 2.1.

Pregel uses message passing to communicate between different compute nodes. In each iteration, the number of messages send from one vertex to its neighbours is unlimited. The corresponding receiver will read these messages in the next iteration. In the distributed environment, in order to reduce the communication cost between compute nodes, the user can define a combination function to group multiple messages into one message and send it to another machine.

Giraph [5] is an open source implementation of Pregel. It is used by Facebook to process the graphs with the scale of trillions of edges [23]. Facebook added the multi-threading support to the Giraph. So the performance of graph loading, writing and computation have been improved compare to Giraph and Pregel.

GraphLab [86] [85] is an asynchronous distributed shared-memory system. Unlike the synchronous communication in the BSP model, the GraphLab develops an asynchronous model called GAS (gather, apply, scatter) to implement its parallel computation in the distribute environment. A program in GraphLab implements a user-defined GAS function for each vertex. To avoid the imbalanced workload caused by high-degree vertices in graphs, a recent version of GraphLab, called PowerGraph [43], introduces a new graph partition scheme to handle this challenge. PowerGraph uses an algorithm called Vertex-cut partitioning, which partitions an input graph by cutting the vertex set, so that the edges of a high-degree vertex will be handled by multiple compute units. As a trade-off, vertices are replicated across compute units, and communication among compute units are required to guarantee that the vertex value on each replica remains consistent.

The GAS model works in the following way: in the Gather phase, each active vertex collects information from its neighbours' vertices and edges. In the Apply phase, each active vertex can update its value based on the information gathered and its old value. Finally, in the Scatter phase, each active vertex can activate the adjacent vertices. However, unlike Pregels message passing paradigm, GraphLab can only gather information from adjacent edges and scatter information to them, which limits the functionality of the GAS model.

GraphLab maintains a global scheduler. The compute units fetch vertices from the scheduler for processing, and add the neighbours of these vertices into the scheduler if needed. The GraphLab engine executes the user-defined GAS function on each active vertex until no vertex remains in the scheduler. The GraphLab scheduler determines the order to activate vertices, which enables GraphLab to provide both synchronous and asynchronous scheduling.

Unlike the synchronous model, in the asynchronous execution, change made

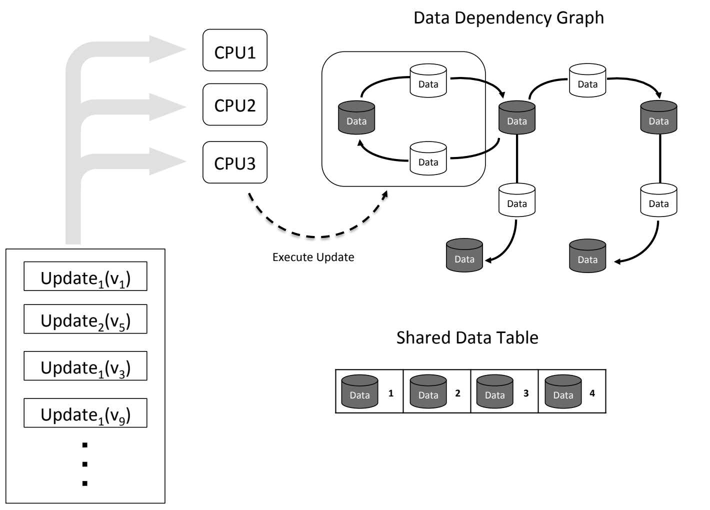


Figure 2.2: The execution model of Graphlab

to each vertex and edge during the Apply phase is committed immediately and visible to subsequent computation. Asynchronous execution can accelerate the convergence of some algorithms. For example, the PageRank algorithm can converge much faster with the asynchronous execution. However, asynchronous execution may incur the extra cost due to locking/unlocking.

GraphLab also provides synchronous execution model. In each iteration, it executes the GAS phases in order. The GAS function of each active vertex runs synchronously with a barrier at the end of each iteration. Changes made to the vertex value are committed at the end of each iteration. Vertices activated in an iteration are executed in the subsequent iteration. The whole process is shown in Figure 2.2.

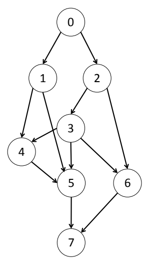
GraphX [44] is a parallel graph processing system. It supports GraphLab and Pregel abstractions. GraphX is built on top of the more general data processing system Spark [141]. It introduces RDG (resilient distributed graph), which is an extension to the RDD (resilient distributed dataset) in Spark. With such extension, many graph algorithms can be implemented easily with the Spark's build in operations, such as *join*, *map* and $group - by$. In order to use Spark's build in operation, GraphX also redesigned the vertex-cut algorithm.

Chaos [110] is a graph processing system designed for analytics on big graphs using clusters. Chaos builds on the X-Stream (discussed in detail in next section) [109] single-machine graph processing system, but scales out to multiple machines. Chaos adopts an edge-centric and GAS (Gather-Apply-Scatter) programming model. Due to the use of the edge-centric model, Chaos partitions the graph for sequential storage access, rather than for locality and load balance, resulting in much lower pre-processing times. In Chaos, the graph data is distributed uniformly and randomly across the cluster and does not attempt to achieve locality, based on the observation that in a small cluster network bandwidth far outstrips storage bandwidth. In addition, Chaos uses work stealing to allow multiple machines to work on a single partition, thereby achieving load balance at runtime. With all these features, Chaos is limited only by the aggregate bandwidth and capacity of all storage devices in the entire cluster.

2.3.3 Graph Processing Systems on a single machine

Nowadays, with the ever increasing processing power of CPUs and large storage capacity, it is possible to process large-scale graph on a single machine. Compared to distribute systems, processing a graph on a single machine has the following advantages. First, the communication overhead is low. On a single machine with multiple cores/processors, the communication between these computing units is achieved by the shared memory, which is much faster compared to the network connection used in distributed systems. Second, from a user's perspective, managing, programming, debugging and optimising codes on a single machine is much easier than on distributed systems. Third, hardware and energy cost is relatively low.

Comparing with the fast growth rate of real-world problems, such as analysis on social networks or the web graph, the increasing rate of memory size is much slower. Therefore, processing large graphs from the persistent storage becomes the mainstream in the graph processing design based on single machines. Since



(a) Input Graph

Shard 1

src	dst	value
0	1	X_1
	2	X_2
1	4	X_3
2	3	X_4
3	4	X_5

Shard 2

src	dst	value
1	5	X_6
3	6	X_7
4	5	X_8

Shard 3

src	dst	value
2	6	X_9
3	5	X_{10}
5	7	X_{11}
6	7	X_{12}

(b) Shard Representation

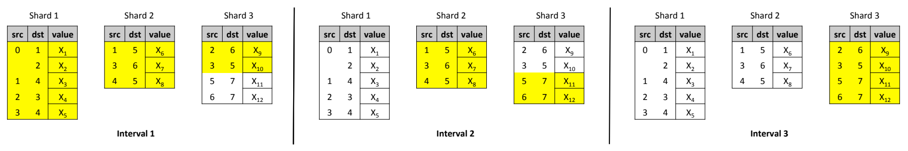
Figure 2.3: Shard Representation of the Graph

most single machine frameworks use the hard disk as the memory extension, how to read from and write to the hard disk efficiently becomes the primary challenge in designing the efficient graph processing framework on these systems. In this section, we discuss the techniques used by some representative single machine graph processing frameworks.

GraphChi [76] is the first graph processing framework that can handle large-scale graphs on a single PC. GraphChi uses the vertex-centric model. In order to process the graphs from the hard disk, it introduces two new techniques to process large graphs in a single PC.

GraphChi uses an innovative out-of-core data structure called shard to reduce the amount of random access to hard disk. Before the computation, GraphChi first pre-processes the graph data. The input data will be partitioned into sub-graphs, each of which is called a shard. Each shard contains a set of vertices and all the inward edges of these vertices. In each shard, edges are sorted in ascending order according to the source vertices ID. The partition method used by GraphChi guarantees that the number of edges in each shard is similar and the size of each shard should be able to fit in the memory. An example of shard representation is shown in Figure 2.3.

GraphChi also developed a method called parallel sliding windows (PSW). During the computation, GraphChi loads the first shard into the memory, and then searches other shards and loads out-edges (the source vertex of these edges



Shard 1

src	dst	value
0	1	X_1
	2	X_2
1	4	X_3
2	3	X_4
3	4	X_5

Shard 2

src	dst	value
1	5	X_6
3	6	X_7
4	5	X_8

Shard 3

src	dst	value
2	6	X_9
3	5	X_{10}
5	7	X_{11}
6	7	X_{12}

Figure 2.4: GraphChi Execution model

are destination vertex in current shard) of the current shard from other shards into memory as well. Once the process of the current shard is finished, it moves to the next shard, and repeats the above process. The whole computation terminates when all the shards have been processed. Organising the graph into shards and computing with PSW can guarantee the sequential read from the hard disk, and hence maximises the performance of the hard disk I/O. An example of PSW execution is shown in Figure 2.4. In this example, the edges labelled with yellow colour represent the edges loaded into memory during that execution interval.

Unlike vertex-centric model used in GraphChi, X-Stream [109] employs an edge-centric computation model, and uses vertices to store the computation state. The computation in X-Stream can be break into scatter gather phases. The scatter phase of the computation takes the edges as the input, and produces an output of updates. In each iteration it reads an edge, reads the data field of its source vertex, and, if needed, appends an update to the output. The gather phase takes the updates produced in the scatter phase as its input. It does not produce any output. For each update in the input stream, it updates the data value of its destination vertex.

Such model enables the sequential access to the edges stored in the hard disk. However, the access to the vertices data is random. To solve this problem, X-Stream introduces streaming partition. A streaming partition consists of a vertex set, an edge list, and an update list. The vertex sets of different streaming partitions are mutually disjoint. The edge list of a streaming partition consists of all edges whose source vertex is in the partition's vertex set. The update list of a streaming partition consists of all updates whose destination vertex is

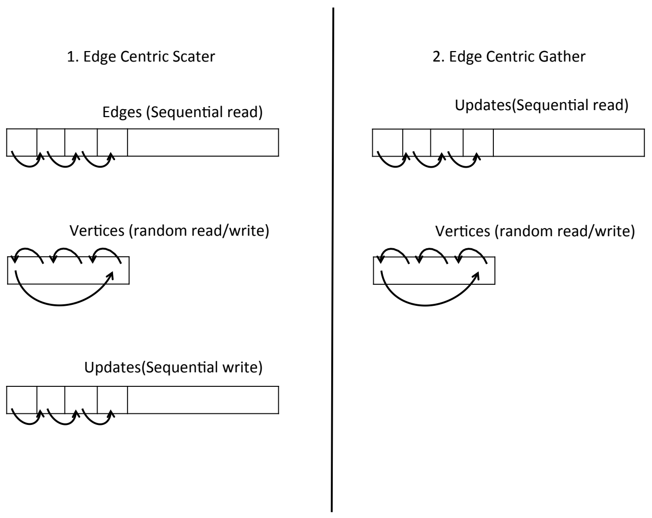


Figure 2.5: Edge Centric Execution model

in the partition's vertex set. Figure 2.5 illustrates the edge centric scatter and gather.

With streaming partitions, the scatter phase iterates over all streaming partitions, and the gather phase also iterates over all streaming partitions. For each streaming partition, the scatter phase reads its vertex set, iterative over its edge list, and produces an output of updates. These updates need to be re-arranged such that each update appears in the update list of the streaming partition containing its destination vertex. This is called the shuffle phase. The shuffle takes as its input the updates produced in the scatter phase, and moves each update to the update list in the streaming partition that containing the destination vertex of the update. After the shuffle phase is completed, the gather phase can start.

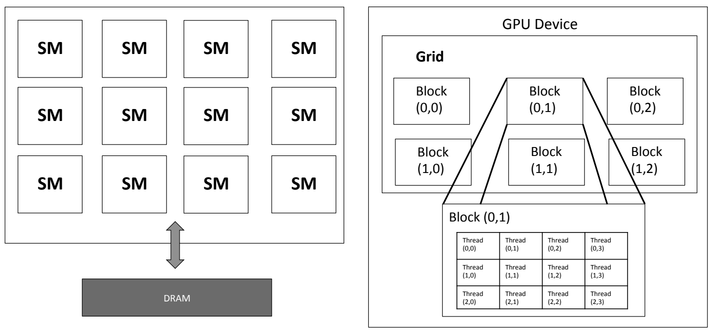
Based on GraphChi, VENUS [22] developed a vertex-centric streamlined computation model. In this model, the graph is partitioned into g-shard and v-shard first. The g-shard is similar to the shard used in GraphChi. It contains a vertex set and all the incoming edges of these vertices. However, unlike GraphChi sorted the edges based on IDs, in VENUS, it places the edges that have same destination vertex next to each other in the g-shard. Each g-shard has a corresponding v-shard, which is used to store the value of the source and destination vertex from the g-shard. VENUS also maintains a global vertex value table, so the v-shard can read and write the vertex value from it. During the computation, VENUS only loads one g-shard into the memory, and does not update the outgoing edges of the current g-shard. Apart from that, the computation can start when finish all incoming-edges of one vertex, so the computation and loading the rest edges can be overlapped. Hence, the performance of VENUS is better than GraphChi.

GridGraph [151] adopts similar streamlined processing model on a single machine. In the GridGraph system, edges are further divided into smaller grids rather than shards in the GraphChi system. Meanwhile, GridGraph applies a 2-level hierarchical partitioning of the grids, which organizes several adjacent grids into a larger virtual grid. In this way, GridGraph can not only ensure data locality but also reduce the amount of disk I/O.

TurboGraph [49] is another disk-based graph processing system, which proposes a novel parallel model, pin-and-slide. It contains a list of slotted pages. Each page contains the outgoing edges of several vertices. TurboGraph system uses a buffer pool in the memory to store several pages. It also divides the vertices into several chunks to ensure data access locality. Each chunk is loaded into memory in sequence and updated by each edge in the buffer pool. TurboGraph requires SSD to ensure its performance because it uses parallel I/O.

2.4 Graph Processing Systems Accelerated by the GPU

The single machine graph processing systems suffer from the limited degree of parallelism provided by conventional processors. To overcome this problem, much research employs GPU to accelerate graph processing due to its massively parallel architecture. However, due to the irregular natural of the graph structure, using GPU for efficient graph processing remains a challenging and open



(a) The simple Cuda Hardware architecture (b) The simple Cuda programming model

Figure 2.6: GPGPU architecture and thread execution model in Cuda

problem due to the following reasons. Because GPU's hardware architecture requires regular data access pattern to achieve the peak memory performance, also, the irregular structure will cause workload imbalance across threads/thread block and intra-warp divergence. In this section, we first briefly describe the GPGPU architecture and CUDA's programming model, and then we review some existing methods and frameworks that designed to overcome the above problem.

GPGPU and CUDA

Using GPU as a general computing unit has attracted considerable attention. GPU provides massive parallel processing power. As the host for the GPU device, CPU organizes and invokes the kernel functions that execute on GPU. As shown in Figure 2.6a, A GPU device consists of a number of streaming multiprocessors (SM), each comprising simple processing engines, called CUDA cores in the NVIDIA terminology [27]. Each SM has its own shared memory, which is equally accessible by all CUDA cores in the SM. At any given cycle, the CUDA cores in a SM execute the same instruction on different data items. SMs communicate with each other through the global memory of GPU.

From the programmers' perspective, the CUDA model [27] is a collection of threads running in parallel. A collection of threads (called a block) runs on a multiprocessor at a given time. Multiple blocks can be assigned to a single multiprocessor and their execution is time-shared. A single execution on a device generates a number of blocks. A collection of all blocks in a single execution is called a grid (Figure 2.6b). All threads of all blocks executing on a single multiprocessor divide its resources equally amongst themselves. Each thread and block is given a unique ID that can be accessed within the thread during its execution. The code running on GPU is actually executed in groups of 32 threads, what NVIDIA calls a warp. The threads within the same warp can run simultaneously on a streaming multiprocessor (SM). The programmer decides the number of blocks and threads to be executed. If the number of threads is more than the warp size, they are time-shared internally on the multiprocessor.

Graph processing with GPGPU

Using GPU for graph processing was first introduced by Harish et al. [50]. Since then, the CSR format has become the mainstream representation to store graphs on GPU. Merrill et al. [99] present a work efficient BFS algorithm. They also use different approaches to minimize the workload imbalance. Virtual Warp Centric has been proposed in [56] to tackle the workload imbalance problem and reduce the intra-warp divergence.

Medusa [146] is a graph processing framework designed to help programmers use the GPU computing power with writing only sequential code. To achieve this goal, Medusa provides a set of user-defined APIs to hide the GPU programming details. Medusa can support multiple GPUs. Medusa extends the Bulk Synchronous Parallel (BSP) model by applying an Edge-Vertex-Message (EMV) model. The EMV model breaks down the vertex-centric workload into separate chunks, each chunk contains vertices, edges, and messages. Compared with a vertex-centric programming model, the EMV model can achieve better workload balance of threads. In addition, a graph-aware message buffer scheme is designed by Medusa to achieve better performance of processing messages between vertices. To maintain this buffer, a message index needs to be stored for each edge.

Totem [42] is a graph processing system that can leverage both the CPU and the GPU (hybrid) as computing units by assigning graph partitions to them. Totem can also support multiple GPUs (with or without the CPU). Totem uses a vertex-centric programming abstraction under the BSP model. Totem strictly uses CSR to represent graphs in-memory. To alleviate the cost of communication between partitions, Totem uses user-provided aggregation to reduce the amount of messages and maintains two sets of buffers on each computing unit for overlapping communication and computation. Totem implements other optimizations to improve the performance, for example, partitioning graphs by the vertex degrees and placing higher degree vertices on CPU. However, as graph size increases, only a fixed portion of graph able to fit in GPUs memory, resulting in GPU underutilization.

MapGraph [40] is an open-source project to support high-performance graph processing. MapGraph uses a modified Gather-Apply-Scatter model to present each iteration of graph processing algorithms. In the Gather phase, vertices collect updated information from incoming edges and/or outgoing-edges. During the Apply phase, every active vertex in the current iteration updates its value. In the Scatter phase, vertices send out messages to their neighbours. For each iteration, MapGraph maintains an array called frontier, which consists of active vertices, to reduce the computation. The frontier for the next iteration is created in the Scatter phase of the current iteration. MapGraph stores graph in CSR format. MapGraph adapts two strategies, dynamic scheduling and two-phase decomposition, to balance the workload of different threads.

The graph processing solutions described above use the CSR format to represent the graph, and hence suffer from the random access to the graph data. Unlike systems described before, CuSha [65] uses G-Shards and CW (concatenated window) representation to avoid warp execution inefficiencies and non-coalesced memory access. The G-shards representation is same as shard structure used in GraphChi. Because each G-shard is processed by one thread block in CuSha, the threads within one threads block will update the outgoing edges' value in other thread blocks after the computation, however, updating vertex value in this way will cause GPU underutilization. To overcome this problem, CuSha develops CW representation, which is an array that combines all the incoming and outgoing edges of each G-shard. Similar to G-shard, each CW is processed by one thread block, because all the edges are grouped together, the threads do not need to access other thread blocks' data to update their value. Although CuSha's methods are effective, such representations consume 2 to 2.5 times more space than CSR, which can hinder the framework from processing very large graphs.

The works described above assume that the entire graph can fit into the global memory of GPU. However, there are large-scale graphs that are even bigger than GPU memory, which makes these works infeasible to solve massive scale graph problems. To overcome this problem, GraphReduce [115], the most stat-of-art framework that aims to process graphs exceeds the GPU memory capacity. It computation model is similar to GraphChi and CuSha. To perform computation, it first partitions the graph into shards, unlike shard used in GraphChi, each shard in GraphReduce contains a set of vertex and all the edges associated with these vertices. Within a shard, the in-edges are sorted in the order of their destinations, and the out-edges are sorted in the order of their sources. The size of each shard should be able to fit in GPU's global memory. During the computation, it loads one or more shards into GPU memory, and uses GAS model to perform the graph computation. However, GraphReduce requires that the entire graph fit in host (CPU) memory, which limited it processing capability. In addition, it has a long pre-processing time because it requires sorting both in and out edges within a shard.

The graph processing method presented in this thesis uses GPU to accelerate the processing speed on a single machine. To overcome the limited memory on a single machine, our work uses the hard disk as an extension to the memory. In addition, our work carefully analyses the benefits that the graph pre-processing conducted by the existing work brings. We then develop a new GPU-based method to accomplish the efficient execution with minimal pre-processing time.

2.5 Summary

The scheduling system plays an important role in modern cluster/cloud systems and has the huge impact on the performance of these systems. In this chapter, we first provided a comprehensive survey of the various scheduling systems/algorithms. Most of the co-scheduling systems discussed in this chapter are based on heuristics algorithms. The problem that faces these co-scheduling systems is the computation of optimal solutions. Without knowing optimal schedules, it is hard to precisely determine how good a scheduling algorithm is. However, the existing optimal co-scheduling system reviewed in this chapter only considers the serial jobs and mainly apply the heuristic approach to find the optimal solutions. We address these problems in this thesis by presenting a set of graph based co-scheduling algorithms. The co-scheduling algorithms we present in this thesis is the first to explore the optimal co-scheduling with a mixture of both serial and parallel jobs.

Because the algorithms we present in this thesis are based on graph theory, therefore, we also studied the graph processing techniques in this chapter. We first discussed the challenges in designing graph processing system in detail and presented a thorough survey of the state-of-the-art of the graph processing frameworks. Based on the review, we discovered two major problems of current GPU based solutions: the non-coalesced memory access and expensive pre-processing. In this thesis, we also present a new graph processing framework to overcome these problems.

Chapter 3

Co-Scheduling of Serial and Parallel Jobs

3.1 Introduction

Co-schedulers in the literature that proport to be optimal only consider serial jobs (each of which runs on a single core). For example, the work in [61] modelled the optimal co-scheduling problem for serial jobs as an integer programming problem. However, in modern multi-core systems, especially those in cluster and cloud platforms, both parallel and serial jobs co-exist[139], [32], [55], [114]. In order to address this problem, this chapter proposes a new method to determine the optimal co-scheduling solution for a mix of serial and parallel jobs. Two types of parallel jobs are considered in this chapter: Multi-Process Parallel (MPP) jobs, such as MPI jobs, and Multi-Thread Parallel (MTP) jobs, such as OpenMP jobs. In this chapter, we first propose a method to co-schedule MPP and serial jobs, and then extend this method to handle MTP jobs.

Resource contention presents different features in single processor and multi-processor machines. In this chapter, a layered graph is first constructed to model the co-scheduling problem on single processor machines. The problem of finding the optimal co-scheduling solutions is then modelled as finding the shortest *valid* path in the graph. Further, this chapter develops a set of algorithms to find the shortest valid path for both serial and parallel jobs. A number of optimization measures are also developed to increase the scheduling efficiency of these proposed algorithms (i.e., to accelerate the solving process of finding the optimal co-scheduling solution). After these, the graph model and proposed algorithms are extended to co-scheduling parallel jobs on multi-processor machines.

It has been shown that the A*-search algorithm is able to effectively avoid unnecessary searches when finding optimal solutions. In this chapter, an A*-search-based algorithm is also developed to combine the ability of the A*-search algorithm and the proposed optimization measures in terms of accelerating the solving process. Finally, a flexible approximation technique is proposed so that we can control the scheduling efficiency by setting the requirement for the solution quality.

We conduct experiments with real jobs to evaluate the effectiveness of the proposed co-scheduling algorithms. The results show that i) the proposed algorithms can find the optimal co-scheduling solution for both serial and parallel jobs, ii) the proposed optimization measures can significantly increase the scheduling efficiency, and iii) the proposed approximation technique is effective in the sense that it is able to balance the scheduling efficiency and the solution quality.

The remainder of this chapter is organized as follows. Section 3.2 formalizes the co-scheduling problem for both serial and MPP jobs, and presents a graph-based model for the problem. Section 3.4 presents methods and optimization measures to determine the optimal co-scheduling solution for serial jobs. Section 3.5 extends the methods proposed in Section 3.4 to incorporate MPP jobs and presents an optimization technique for the extended algorithm. In Section 3.6, we extend the graph-based model and proposed algorithms for multi-processor machines; Section 3.7 then adjusts the graph model and the algorithms for MTP jobs. Section 3.8 presents the A*-search-based algorithm; a clustering

Table 3.1: The summary of all symbols used in this Chapter

n	The number of jobs to be scheduled.
u	The number of cores per processor.
m	The number of processors.
j_i	The job i of all jobs.
ct_i	The computation time when job i runs alone.
S_i	The set of jobs that job i co-run with.
$ct_{i,S}$	The computation time when job i co-runs with job set S.
$d_{i,S}$	The performance degradation when job i co-runs with job set S.
x_{i,S_i}	The binary variable indicating whether or not the job S_i is in the final scheduling.
δ_j	The parallel job j.
p_i	The process i of a parallel job.
P	The number of parallel jobs.
d_{δ_j}	The maximum performance degradation of parallel job j among all processes.
γ_i	The number of the neighbouring processes that p_i has.
$\alpha_i(k)$	The amount of data that process i needs to communicate with its k-th process.
B	The bandwidth for inter processor communication.
$b_i(k)$	The k-th neighbouring process of process p_i
$\beta(k, S_i)$	The binary variable indicating whether or not the job $b_i(k)$ is in job set S_i

approximation technique is proposed in Section 3.10 to control the scheduling efficiency according to the required solution quality. Experimental results are presented in Section 3.11 and finally, Section 3.12 concludes the chapter.

3.2 Formalizing the job co-scheduling problem

First, in Subsection 3.2.1, we briefly summarize the approach in [61] to formalizing the co-scheduling of serial jobs. Subsection 3.2.2 then formalizes the objective function for co-scheduling a mix of serial and MPP jobs, and Subsection 3.3 presents a graph model for the co-scheduling problem. This section focusses on single processor machines, i.e., all CPU cores reside on the same chip. For convenience purposes, we listed all the symbols used in this chapter in Table 3.1.

3.2.1 Formalizing the co-scheduling of serial jobs

The work in [61] shows that due to resource contention, co-running jobs generally run slower on a multi-core processor than if they run alone. This performance degradation is called the *co-run degradation*. When a job i co-runs with the jobs in a job set S, the co-run degradation of job i can be formally defined as in Eq. 3.1, where ct_i is the computation time when job i runs alone, S is a set of jobs and $ct_{i,S}$ is the computation time when job i co-runs with the set of jobs in S. Typically, the value of $d_{i,S}$ is a non-negative value.

$$d_{i,S} = \frac{ct_{i,S} - ct_i}{ct_i} \tag{3.1}$$

In the co-scheduling problem considered in [61], n serial jobs are allocated to multiple identical u-core processors so that each core is allocated with one job. m denotes the number of u-core processors needed, which can be calculated as $\frac{n}{u}$ (if n cannot be divided by u, we can simply add $(u - n \bmod u)$ imaginary jobs which have no performance degradation with any other jobs). The objective of the co-scheduling problem is to find the optimal way to partition n jobs into m u-cardinality sets, so that the sum of $d_{i,S}$ in Eq.3.1 over all n jobs is minimized. This objective can be formalized as the following IP problem shown in Eq.3.2, where x_{i,S_i} is the decision variable of the IP and S_i is a job set that co-runs with job j_i. The decision constraints of the IP problem are shown in Eq.3.3 and Eq.3.4. Note that the number of all job sets that may co-run with job j_i (i.e., the number of all possible S_i) is $\binom{n-1}{u-1}$.

$$min \sum_{i=1}^{n} d_{i,S_i} x_{i,S_i} \tag{3.2}$$

$$x_{i,S_i} = \begin{cases} 1 & \text{if } j_i \text{ is co-scheduled with } S_i, \\ 0 & \text{otherwise.} \end{cases} \quad 1 \leq i \leq n \tag{3.3}$$

$$\sum_{\forall S_i} x_{i,S_i} = 1, \quad 1 \leq i \leq n \tag{3.4}$$

3.2.2 Formalizing the co-scheduling of serial and parallel jobs

We first model the co-scheduling of embarrassingly parallel (*PE*) jobs (i.e., those with no dependency between parallel processes), and then extend the model to co-schedule parallel jobs with inter-process dependencies (denoted by the term *PC*). An example of a PE job is parallel Monte Carlo simulation [108]. In such an application, multiple slave processes are running simultaneously to perform the Monte Carlo simulations. After a slave process completes its portion of work, it sends the result back to the master process. After the master process receives the results from all slaves, it reduces the final result (i.e., by calculating the average). An example of a PC job is an MPI application for matrix multiplication. In both types of parallel job, the completion time of a job is determined by the slowest process in the job.

IP model for PE jobs

Eq.3.2 cannot be used as the objective for finding the optimal co-scheduling of parallel jobs. This is because Eq.3.2 will sum up the degradation experienced by each process of a parallel job. However, as explained above, the slowest process determines the finish time of a parallel job. In the case of the PE jobs, a bigger degradation of a process indicates a longer execution time for that process. Therefore, no matter how small degradation other processes have, the execution flow in the parallel job has to wait until the process with the biggest degradation finishes. Thus, the finish time of a parallel job is determined by the biggest degradation experienced by all its processes. Therefore, co-scheduling a mix of serial jobs and PE jobs can be modelled as the following IP problem.

$$min(\sum_{j=1}^{P}(\max_{p_i \in \delta_j}(d_{i,S_i} \times x_{i,S_i})) + \sum_{i=1}^{n-P} d_{i,S_i} \times x_{i,S_i}) \tag{3.5}$$

subject to:

$$\sum_{\forall S_i} x_{i,S_i} = 1, \quad 1 \leq i \leq n \tag{3.6}$$

In above equations, Eq.3.7 denotes the maximum degradation of a parallel job. The total degradation is calculated using Eq. 3.5, where n is the number of all processes (a serial job has one process and a PE has multiple processes), δ_j is a parallel job, P is the number of parallel jobs, S_i is the set of processes that may co-run with process p_i. Eq. 3.6 lists the decision constraints of Eq.3.5, which are the same as those for the IP modelling for serial jobs, i.e., Eq.3.3 and Eq.3.4.

$$d_{\delta_j} = \max_{p_i \in \delta_j} d_{i,S_i} \tag{3.7}$$

The max operation in Eq 3.5 can be eliminated by introducing an auxiliary variable y_j for each parallel job δ_j. Each y_j has the following inequality relation with the original decision variables.

$$for\ all\ p_i \in \delta_j, d_{i,S_i} x_{i,S_i} \leq y_j \tag{3.8}$$

Therefore, the objective function in (3.5) is transformed to

$$min(\sum_{j=1}^{P} y_j + \sum_{i=1}^{n-P}(d_{i,S_i} \times x_{i,S_i})) \tag{3.9}$$

IP model for PC jobs

In the case of the PC jobs, the slowest process in a parallel job is determined by both performance degradation and communication time. Therefore, we define the *communication-combined degradation*, which is expressed using Eq.3.10, where $c_{i,S}$ is the communication time taken by parallel process p_i when p_i co-runs with the processes in S_i. As with d_{i,S_i}, c_{i,S_i} also varies with the co-scheduling solutions. We can see from Eq.3.10 that for all processes in a parallel job, the one with the biggest sum of performance degradation (in terms of the computation time) and the communication has the greatest value of d_{i,S_i}, since the computation time of all processes (i.e., ct_i) in a parallel job is the same when a parallel job is evenly balanced. Therefore, the greatest d_{i,S_i} of all processes in a parallel job should be used as the communication-combined degradation for that parallel job.

When the set of jobs to be co-scheduled includes both serial jobs and PC jobs, we use Eq.3.10 to calculate d_{i,S_i} for each parallel process p_i, and then we replace d_{i,S_i} in Eq.3.5 with that calculated by Eq.3.10 to formulate the objective of co-scheduling a mix of serial and PC jobs.

$$d_{i,S_i} = \frac{ct_{i,S_i} - ct_i + c_{i,S_i}}{ct_i} \tag{3.10}$$

Whether Eq.3.5 replaced with d_{i,S_i} calculated by Eq.3.10 still makes an IP problem depends on the form of c_{i,S_i}. Next, we first present the modelling of c_{i,S_i}, and then use an example for illustration.

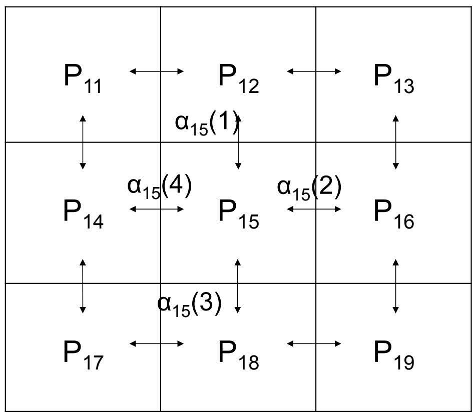
Assume that a parallel job has regular communication patterns among its processes. c_{i,S_i} can be modelled using Eq.3.11 and 3.12, where γ_i is the number of the neighbouring processes that process p_i has corresponding to the decomposition performed on the data set to be calculated by the parallel job, $\alpha_i(k)$ is the amount of data that p_i needs to communicate with its k-th neighbouring process, B is the bandwidth for inter-processor communication (typically, the communication bandwidth between the machines in a cluster is same), $b_i(k)$ is p_i's k-th neighbouring process, and $\beta_i(k, S_i)$ is 0 or 1 as defined in Eq.3.12. $\beta_i(k, S_i)$ is 0 if $b_i(k)$ is in the job set S_i co-running with p_i. Otherwise, $\beta_i(k, S_i)$ is 1.

Essentially, Eq.3.11 calculates the total amount of data that p_i needs to communicate, which is then divided by the bandwidth B to obtain the communication time. $\beta_i(k, S_i)$ in Eq.3.11 is further determined by Eq.3.12. Note that p_i's communication time can be determined by only examining which neighbouring processes are not in the job set S_i co-running with p_i, no matter which machines that these neighbouring processes are scheduled to. Namely, c_{i,S_i} can be calculated by only knowing the information of the local machine where process p_i is located. Therefore, such a form of c_{i,S_i} makes Eq.3.10 still be of an IP form that can solved by the existing IP solvers.

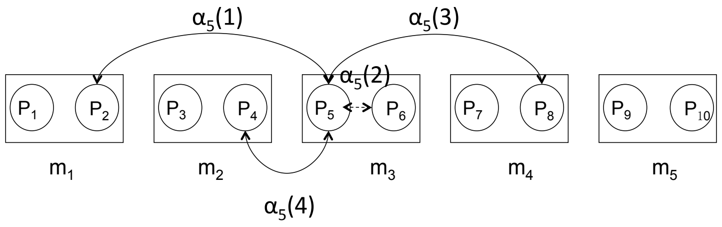
$$c_{i,S_i} = \frac{1}{B}\sum_{k=1}^{\gamma_i}(\alpha_i(k) \times \beta_i(k, S_i)) \tag{3.11}$$

$$\beta_i(k, S_i) = \begin{cases} 0 & \text{if } b_i(k) \in S_i \\ \\ 1 & \text{if } b_i(k) \notin S_i \end{cases} \tag{3.12}$$

We use an example as shown in Figure 3.1 to illustrate the calculation of c_{i,S_i}. Figure 3.1a represents a data set to be calculated on. Assume that a typical 2-dimensional decomposition is performed on the data set, resulting in 9 subsets of data. And 9 processes are used to calculate these 9 subsets in parallel (e.g., using MPI). The arrows between the data subsets in Figure 3.1a represent the communication pattern between the processes. Assume that the parallel job is labelled as δ_1 and the 9 processes in δ_1 are labelled as $p_1 ... p_9$. Also assume that these 9 processes are scheduled on 2-core machines as shown in Figure 3.1b. Now consider process p_5. It has intra-processor communication with p_6 and inter-processor communications with p_2, p_4 and p_8. Since the intra-processor communication can occur simultaneously with inter-processor communication and the intra-processor communication is always faster than the inter-processor communication, the communication time taken by process p_5 in the schedule, i.e., $c_{5,\{p_6\}}$, is $\frac{1}{B}(\alpha_5(1) + \alpha_5(3) + \alpha_5(4))$. Note that in typical 1D, 2D or 3D decompositions, the data that a process has to communicate with



(a) An exemplar parallel job δ_1 and its inter-process communication pattern



(b) A schedule of the parallel job δ_1 and a serial job p_{10} on 2-core machines

Figure 3.1: An illustrative example for modelling the communication time

the neighbouring processes in the same dimension are the same. In Figure 3.1, for example, $\alpha_5(1) = \alpha_5(3)$ and $\alpha_5(2) = \alpha_5(4)$.

3.3 The graph model for co-scheduling

This chapter proposes a graph-based approach to find the optimal co-scheduling solution for both serial and parallel jobs. In this section, the graph model is first presented, and the intuitive strategies to solve the graph model are then discussed.

3.3.1 The graph model

As formalized in Section 3.2.1, the objective of solving the co-scheduling problem for serial jobs is to find a way to partition n jobs, j_1, j_2, ..., j_n, into m u-cardinality sets, so that the total degradation of all jobs is minimized. The number of all possible u-cardinality sets is $\binom{n}{u}$. In this chapter, a graph is constructed, called the co-scheduling graph, to model the co-scheduling problem for serial jobs (we will discuss in Section 3.5 how to use this graph model to handle parallel jobs). There are $\binom{n}{u}$ nodes in the graph and a node corresponds to a u-cardinality set. Each node represents a u-core processor with u jobs assigned to it. The ID of a node consists of a list of the IDs of the jobs in the node. In the list, the job IDs are always placed in an ascending order. The weight of a node is defined as the total performance degradation of the u jobs in the node. The nodes are organized into multiple levels in the graph. The i-th level contains all nodes in which the ID of the first job is i. At each level, the nodes are placed in ascending order of their ID's. A *start* node and an *end* node are added as the first (level 0) and the last level of the graph, respectively. The weights of the start and the end nodes are both 0. The edges between the nodes are dynamically established as the algorithm of finding the optimal solution progresses. Such organization of the graph nodes will be used to help reduce the time complexity of the co-scheduling algorithms proposed in this chapter. Figure 3.2 illustrates the case where 6 jobs are co-scheduled to dual-core processors. The figure also shows how to code the node IDs in the graph and how to organize the nodes into different levels. Note that for clarity we do not draw all edges.

In the constructed co-scheduling graph, a path from the start to the end node forms a valid co-scheduling solution if the path does not contain duplicated jobs; this we call a *valid path*. The distance of a path is defined as the sum of the weights of all nodes on the path. Finding the optimal co-scheduling solution is equivalent to finding the shortest valid path from the start to the end node. It is straightforward to know that a valid path contains at most one node from

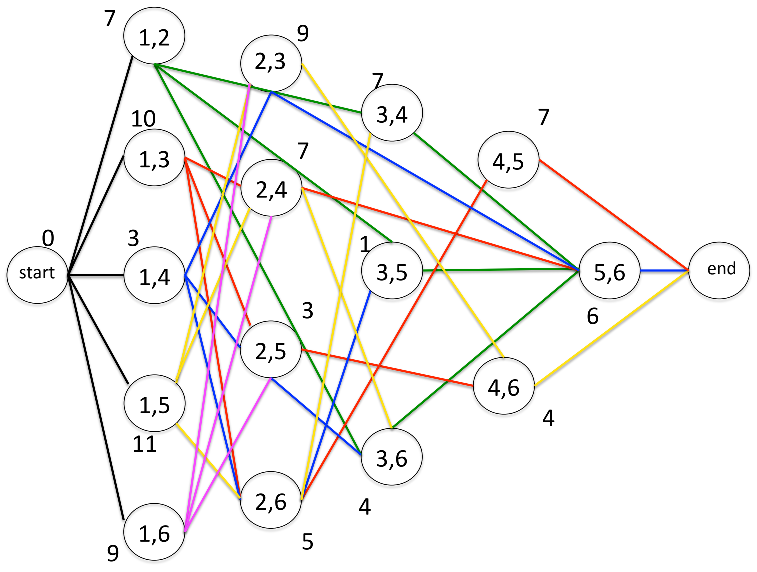


Figure 3.2: The exemplar co-scheduling graph for co-scheduling 6 jobs on Dual-core machines; the list of numbers in each node is the node ID; A number in a node ID is a job ID; The edges of the same color form the possible co-scheduling solutions; The number next to the node is the node weight, i.e., total degradation of the jobs in the node.

each level in the graph.

3.3.2 Intuitive strategies to solve the graph model

We first try to solve the graph model using Dijkstra's shortest path algorithm [26]. However, we find that Dijkstra's algorithm can not be directly applied to find the correct solution. This can be illustrated using the example in Figure 3.2. In order to quickly reveal the problem, let us consider only five nodes in Figure 3.2, $\langle 1,5\rangle, \langle 1,6\rangle, \langle 2,3\rangle, \langle 4,5\rangle, \langle 4,6\rangle$. Assume the weights of these nodes are 11, 9, 9, 7 and 4, respectively. Out of all these five nodes, there are two valid paths reaching node $\langle 2,3\rangle$: $\langle\langle 1,5\rangle, \langle 2,3\rangle\rangle$ and $\langle\langle 1,6\rangle, \langle 2,3\rangle\rangle$. Since the distance of $\langle\langle 1,6\rangle, \langle 2,3\rangle\rangle$, which is 18, is shorter than that of $\langle\langle 1,5\rangle, \langle 2,3\rangle\rangle$, which is 20, the path $\langle\langle 1,6\rangle, \langle 2,3\rangle\rangle$ will not been examined again according to Dijkstra's algorithm. In order to form a valid schedule, the path $\langle\langle 1,6\rangle, \langle 2,3\rangle\rangle$ has to connect to node $\langle 4,5\rangle$ to form a final valid path $\langle\langle 1,6\rangle, \langle 2,3\rangle, \langle 4,5\rangle\rangle$ with the distance of 25. However, we can see that $\langle\langle 1,5\rangle,\langle 2,3\rangle,\langle 4,6\rangle\rangle$ is also a valid schedule and its distance is less than that of $\langle\langle 1,6\rangle,\langle 2,3\rangle,\langle 4,5\rangle\rangle$. However, the schedule $\langle\langle 1,5\rangle,\langle 2,3\rangle,\langle 4,6\rangle\rangle$ is dismissed by Dijkstra's algorithm during the search for the shortest path.

The main reason for this problem is that Dijkstra's algorithm only records the shortest subpaths reaching a certain node and dismisses other optional subpaths. This is fine for searching for the shortest path, but in our problem we have to search for the shortest *valid* path. Dijkstra's algorithm searches up to a certain node in the graph, recording only the shortest subpath up to that node. As such, not all nodes among the unsearched nodes can form a valid schedule with the current shortest subpath, which may cause the shortest subpath to connect to nodes with bigger weights. As illustrated, a subpath that has been dismissed by Dijkstra's algorithm may be able to connect to the unsearched nodes with smaller weights and therefore generate a shorter final valid path.

In order to address the above problem, an intuitive strategy is to revise Dijkstra's algorithm so that it will not dismiss any subpath, i.e., to allow the algorithm to record every visited subpath. Then, the path with the smallest distance among all examined and complete paths is the optimal co-scheduling result. This strategy is equivalent to enumerating all possible subpaths in the graph. The time complexity of such a strategy is very high, which will be discussed when we compare it with the SVP algorithm presented in Subsection 3.4.1. This time complexity motivates us to design more efficient algorithms to find the shortest valid path. In the next section, we propose a more efficient algorithm to find the shortest valid path; this we call the SVP (Shortest Valid Path) algorithm.

3.4 Shortest valid path for serial jobs

3.4.1 The SVP algorithm

In order to tackle the problem highlighted in the application of Dijkstra's algorithm, the following *dismiss strategy* is adopted by the SVP algorithm:

SVP records all jobs that an examined sub-path contains. Assume a set of sub-paths, S, each of which contains the same set of jobs (the set of graph nodes that these paths traverse are different). SVP only keeps the path with the smallest distance and other paths are dismissed in further searches for the shortest path.

This strategy will clearly demonstrate improved efficiency compared with the intuitive, enumerative strategy, i.e., the SVP algorithm examines far fewer subpaths than the enumerative strategy. This is because, for all different subpaths that contain the same set of jobs, only one subpath (the shortest) will spawn further subpaths and all other subpaths will be discarded.

The SVP algorithm is outlined in Algorithm 3.1. The main differences between SVP and Dijkstra's algorithm lie in three aspects: 1) The invalid paths, which contain the duplicated jobs, are disregarded by SVP during the searching; 2) The dismiss strategy is implemented; 3) No edges are generated between nodes before SVP starts and the node connections are established as SVP progresses. In this way, only the node connections spawned by the recorded subpaths will be generated and this will therefore further improve performance.

In Algorithm 3.1, *Graph* is implemented as a two-dimensional linked list with the first dimension linking the graph levels and the second linking all nodes in a level. Q is a list of objects and, an object v, has four attributes: a job set ($v.jobset$), the current shortest valid path relating to the job set ($v.path$), the distance of the shortest path ($v.distance$) and the level of the last node in the shortest path ($v.level$). Q is initialized as having the object for the start node (Lines 2-3). Each iteration, SVP selects from Q such an object v that has the smallest distance (Lines 4 and 30). In contrast, in Dijkstra's algorithm we find a data structure (denoted by Q') which is similar to Q. In Q', each element holds

Algorithm 3.1: The SVP Algorithm

```
1: SVP($Graph$)
2:   $v.jobset = \{Graph.start\}$; $v.path = Graph.start$; $v.distance = 0$; $v.level = 0$;
3:   add $v$ into $Q$;
4:   Obtain $v$ from $Q$;
5:   while $Graph.end$ is not in $v.jobset$
6:       for every level $l$ from $v.level + 1$ to $Graph.end.level$ do
7:           if job $l$ is not in $v.jobset$
8:               $valid\_l = l$;
9:               break;
10:      $k = 1$;
11:      while $k \le \binom{n-valid\_l}{u-1}$
12:          if $node_k.jobset \cap v.jobset = \phi$
13:              $distance = v.distance + node_k.weight$;
14:              $J = v.jobset \cup node_k.jobset$;
15:              if $J$ is not in $Q$
16:                  Create an object $u$ for $J$;
17:                  $u.jobset = J$;
18:                  $u.distance = distance$;
19:                  $u.path = v.path + node_k$;
20:                  $u.level = node_k.level$
21:                  Add $u$ into $Q$;
22:              else
23:                  Obtain $u'$ whose $u'.jobset$ is $J$;
24:                  if $distance < u'.distance$
25:                      $u'.distance = distance$;
26:                      $u'.path = v.path + node_k$;
27:                      $u'.level = node_k.level$
28:          $k+ = 1$;
29:      Remove $v$ from $Q$;
30:      Obtain the $v$ with smallest $v.distance$ from $Q$;
31:  return $v.path$ as the shortest valid path;
```

a node and the shortest distance up to that node. In each iteration, Dijkstra's algorithm selects the node with the shortest distance among all nodes stored in Q' and extends the subpath to that node.

At Line 5, our SVP algorithm enters a loop which visits every node and finds the shortest valid path in the graph. Given a v obtained in Line 4, since a valid path cannot contain duplicated jobs, there should not exist edges between nodes at the same level. Therefore, the algorithm starts from $v.level + 1$ (Line 6) to search for a valid level, which is a level that contains at least one node that can form a valid path with the current subpath.

After a valid level is found, the algorithm continues to search this level for valid nodes that can append to the current subpath and form a new valid subpath (Lines 10-11). Lines 12-26 implement our so-called dismiss strategy.

If a valid node, k, is found, the algorithm calculates the distance after the current subpath extends to node k and constructs a new job set J that is $v.jobset \cup k.jobset$ (Lines 12-14). If J is not in Q (Line 15), an object u' is created for J and added to Q (Lines 16-21). If J is in Q, then the algorithm checks whether this new subpath has a shorter distance than the one recorded for the job set J (Line 24). If so, the attributes of the object corresponding to J are updated (Lines 25-27).

The termination condition of SVP is when the v obtained from Q contains the *Graph.end* (Line 5). This is because when v contains *Graph.end*, it means *v.path* is already a complete valid path. Moreover, since *v.path* has the smallest distance in Q, all subpaths recorded in Q will form longer complete paths than *v.path*. Therefore, *v.path* is the shortest valid path.

We now present an example to illustrate the workings of the SVP algorithm. In order to simplify the description, we only consider how to find the shortest path from five nodes in Figure 3.2, i.e., $\langle 1,5\rangle, \langle 1,6\rangle, \langle 2,3\rangle, \langle 4,5\rangle, \langle 4,6\rangle$. Assume the weights of these nodes are 11, 9, 9, 7 and 4, respectively. At the beginning of Algorithm 1, we initialize Q with node $\langle start\rangle$ (Lines 2-3). In the first iteration of the while loop (Line 5), the algorithm examines the nodes $\langle 1,5\rangle, \langle 1,6\rangle$, and adds two job sets, (1,5) and (1,6), in Q with the weights 11 and 9 respectively.

At the end of the first iteration, the algorithm selects the job set (1,6) from Q since it is the job set with the smallest weight in Q (Line 30). Then the algorithm enters into the second iteration and searches for the valid level (Lines 6-9). In this case, the second level is selected, since job 2 does not exist in the job set (1,6). The algorithm starts searching for valid nodes within this level. The node $\langle 2,3\rangle$ is found as the valid node in this level, since none of the jobs in this node exist in the job set (1,6) (Line 12). Therefore, the distance between nodes $\langle 1,6\rangle$ and $\langle 2,3\rangle$ is computed (Line 13), which is 18, and a new job set (1,2,3,6) is created (Line 14). Since job set (1,2,3,6) does not exist in Q, the algorithm creates a new element u for it (Line 16). The job set (1,2,3,6) is assigned to *u.jobset* (Line 17). The distance of 18, path $\langle\langle 1,6\rangle, \langle 2,3\rangle\rangle$ and the level of the node $\langle 2,3\rangle$ are assigned to *u.distance*, *u.path* and *u.level*, respectively (Lines 18-20). The algorithm then removes the job set (1,6) from Q (Line 29). At the end of the current iteration, the algorithm selects the job set (1,5), which is the job set with the smallest weight, from Q (Line 30) and enters the next iteration. In the new iteration, the algorithm repeats the above procedure and the job set (1,2,3,5) with the weight of 20 will be added into Q.

In the following iterations, the algorithm selects the job set (1,2,3,6), adds a new job set (1,2,3,4,5,6) with the distance of 25 into Q and removes job set (1,2,3,6) from Q in the same manner as that described above. After this, the job set (1,2,3,5) is selected from Q, and the algorithm will construct the job set (1,2,3,4,5,6), which already exists in Q. But this time, the distance associated with this job set is 24. Since the job set (1,2,3,4,5,6) has already been stored in Q, the algorithm compares this newly computed distance with the one saved in Q (Line 24). Since this new distance is smaller, the algorithm updates the attributes saved in Q with the new distance (Lines 22-27). The algorithm reaches the end node in the next iteration (Line 5) and returns the shortest valid path, which is $\langle\langle 1,5\rangle, \langle 2,3\rangle, \langle 4,6\rangle\rangle$.

The time complexity of Algorithm 3.1 is $\mathrm{O}\big(\sum\limits_{i=1}^{m} \binom{n-i}{i\cdot(u-1)} \cdot ((n-u+1)+\frac{\binom{n}{u}}{n-u+1}+ log\binom{n}{u})\big)$, where m is the number of u-core machines required to run n jobs.

When the problem is to schedule n jobs on u-core machines, the constructed graph has $\binom{n}{u}$ nodes and $n-u+1$ levels. Therefore, the average number of nodes per level is $\frac{\binom{n}{u}}{n-u+1}$. In each iteration, the worst-case time complexity for finding the next valid level is $\mathrm{O}\big(n-u+1\big)$ (Lines 6-8), while the average time complexity for finding a valid node in a level is $\mathrm{O}\big(\frac{\binom{n}{u}}{n-u+1}\big)$ (Lines 11-14). The time complexity for obtaining an element with the smallest distance from Q is $\mathrm{O}\big(log\binom{n}{u}\big)$ (Line 30). Other operations in each iteration take time $\mathrm{O}\big(1\big)$. Therefore, each iteration is bounded by $\mathrm{O}\big((n-u+1)+\frac{\binom{n}{u}}{n-u+1}+log\binom{n}{u}\big)$. The maximum number of iterations of the outer-most loop (Line 5) equals the maximum length of Q, which in turn equals the number of all possible job sets that can form a valid sub-path in the graph. This can be calculated by $\sum\limits_{i=1}^{m} \binom{n-i}{i\cdot(u-1)}$, where m is the number of u-core machines required. The calculation is explained as follows. All subpaths start from the first level in the graph. The number of job sets that can form a valid subpath whose length is 1 (i.e., the subpath contains only one node in the graph) equals the number of nodes in the first level, which in turn equals the probability of placing $n-1$ jobs into $u-1$ positions and is therefore $\binom{n-1}{u-1}$. Now let us calculate the number of job sets that can form a valid subpath whose length is 2 (i.e., the subpath contain two nodes in the graph). Because of the layout of the graph, the valid subpath whose length is 2 must contain jobs 1 and 2. Therefore, the number of such job sets equals the probability of placing $n-2$ jobs into $2 \cdot (u-1)$ positions, which is $\binom{n-2}{2 \cdot (u-1)}$. Thus, there are in total $\sum\limits_{i=1}^{m} \binom{n-i}{i \cdot (u-1)}$ job sets which can form valid subpaths whose lengths are 1 or 2. The maximum length of a valid path is m (i.e., the number of u-core machines required). Therefore, the total number of job sets that form a valid subpath is $\sum\limits_{i=1}^{m} \binom{n-i}{i \cdot (u-1)}$.

Now we discuss the difference between SVP and the intuitive method from the algorithm's perspective. In SVP, the maximum length of Q is the number of all possible job sets that can form a valid subpath in the graph, which is $\sum\limits_{i=1}^{m} \binom{n-i}{i \cdot (u-1)}$. However, the maximum length of Q in the intuitive method is the number of all valid subpaths in the graph, which is $\sum\limits_{k=1}^{m} \prod\limits_{i=0}^{k-1} \binom{n-(i \cdot u)-1}{u-1}$, while the steps in each iteration of the intuitive method, i.e., the steps used by the intuitive method to find the next valid level and a valid node in the level can be similar as those by SVP. Therefore, essentially, SVP accelerates the solving process by significantly reducing the length of Q in the algorithm, i.e., reducing the number of iterations of the search loop. For example, when u is 4 and n is 8, 12 and 16, the maximum length of Q is 35, 376 and 4174, respectively, in SVP, while the maximum length of Q is 280, 46365 and 5330780, respectively, in the intuitive method.

3.4.2 Further optimization of SVP

One of the most time-consuming steps in Algorithm 3.1 is to scan every node in a valid level to find a valid node for a given subpath *v.path* (Line 11 and 28). By carefully examining the Algorithm 3.1 and the structure of co-scheduling graph. We noticed that once the algorithm locates a node that contains a job appearing in *v.path*, the number of the nodes that follow that node and also contains that job can be calculated since the nodes are arranged in the ascending order of node ID. These nodes are all invalid and can therefore be skipped by the algorithm.

Based on the above discussion, the O-SVP (Optimal SVP) algorithm is proposed to further optimize SVP. The only difference between O-SVP and SVP is that in the O-SVP algorithm, when the algorithm gets to an invalid node, instead of moving to the next node, it calculates the number of nodes that can be skipped and jumps to a valid node. Effectively, O-SVP can find a valid node in the time O(1). Therefore, the time complexity of O-SVP is $O\big(\sum\limits_{i=1}^{m} \binom{n-i}{i\cdot(u-1)} \cdot ((n-u+1) + log\binom{n}{u})\big)$. The outline algorithm for O-SVP is omitted in this chapter.

In summary, SVP accelerates the solving process over the enumerative method by reducing the length of Q in the algorithm, while O-SVP further accelerates SVP by reducing the time spent in finding a valid node in a level.

3.5 Shortest valid path for parallel jobs

The SVP algorithm presented in the last section considers only serial jobs. This section addresses the co-scheduling of both serial and parallel jobs. Subsection 3.5.1 presents how to handle embarrassingly parallel (PE) jobs, while Subsection 3.5.2 further extends the work in Subsection 3.5.1 to handle parallel jobs with inter-process communications (PC).

3.5.1 Co-scheduling PE jobs

In Subsection 3.5.1, the SVPPE (SVP for PE) algorithm is proposed, extending SVP to incorporate PE jobs. Subsection 3.5.1 presents the optimization techniques used to accelerate the solving of SVPPE.

The SVPPE algorithm

When Algorithm 3.1 finds a valid node, it calculates the new distance after the current path extends to that node (Line 13). This calculation is adequate for serial jobs, but cannot be applied to parallel jobs. As discussed in Subsection 3.2.2, the completion time of a parallel job is determined by Eq. 3.10. In order to incorporate parallel jobs, we can treat each process of a parallel job as a serial job (therefore the graph model remains the same) and extend the SVP algorithm simply by changing the means by which we calculate the path distance.

In order to calculate the performance degradation for PE jobs, a number of new attributes are introduced. First, two new attributes are added to an object v in Q. One attribute stores the total degradation of all serial jobs on $v.path$ (denoted by $v.dg\_serial$). The other attribute is an array, in which each entry stores the largest degradation of all processes of a parallel job p_i on $v.path$ (denoted by $v.dg\_p_i$). Second, two similar new attributes are also added to a graph node $node_k$. One stores the total degradation of all serials jobs in $node_k$ (denoted by $node_k.dg\_serial$). The other is also an array, in which each entry stores the degradation of a parallel job p_i in $node_k$ (denoted by $node_k.dg\_p_i$).

SVPPE is outlined in Algorithm 3.2. The only differences between SVPPE and SVP are: 1) Changing the means by which we calculate the subpath distance (Line 13-19 in Algorithm 3.2), and 2) Updating the newly introduced attributes for the case where J is not in Q (Line 28-30) and the case otherwise (Line 38-40).

The maximum number of iterations of all for-loops (Line 14, 28 and 38) is u, because there are at most u jobs in a node. Each iteration takes constant time. Therefore, the worst-case complexity of computing the degradation (the first for-loop) and updating the attributes (the two other for-loops) are $O(u)$.

Algorithm 3.2: The SVPPE algorithm

```
1:      SVPPE(Graph, start, end):
2-12:     ... //same as Line 2-12 in Algorithm 1;
13:         $total\_dg\_serial = v.dg\_serial + node_k.dg\_serial$
14:         for every parallel job, $p_i$, in $node_k$:
15:           if $p_i$ in $v.jobset$:
16:             $dg\_p_i = max(v.dg\_p_i, node_k.dg\_p_i)$;
17:           else
18:             $dg\_p_i = node_k.dg\_p_i$;
19:         $distance = \sum dg\_p_i + total\_dg\_serial$;
20-26:        ... //same as Line14-20 in Algorithm 1
27:         $u.dg\_serial = total\_dg\_serial$;
28:         for every parallel job, $p_i$, in $node_k$ do
29:           $u.dg\_p_i = dg\_p_i$;
30-36:        ... //same as Line21-27 in Algorithm 1
37:           $u'.dg\_serial = total\_dg\_serial$;
38:         for every parallel job, $p_i$, in $node_k$ do
39:           $u'.dg\_p_i = dg\_p_i$;
40-43:      ... //same as Line28-31 in Algorithm 1
```

Therefore, combined with the time complexity of Algorithm 3.1, the worst-case complexity of Algorithm 3.2 is $\mathrm{O}\big(\sum_{i=1}^{m}\binom{n-i}{i\cdot(u-1)}\cdot((n-u+1)+u\cdot\frac{\binom{n}{u}}{n-u+1}+log\binom{n}{u})\big)$.

Process condensation for optimizing SVPPE

An obvious optimization measure for SVPPE is to skip the invalid nodes in a similar way to that given in O-SVP algorithm, which is not repeated in this Subsection. This subsection focuses on proposing another important optimization technique that is only applicable to PE jobs. The optimization technique is based on the following observation: different processes of a parallel job should have the same mutual effect with other jobs. So it is unnecessary to differentiate different processes of a parallel job, treating them as individual serial jobs.

Therefore, the optimization technique, which is called the *process condensation technique* in this chapter, labels a process of a parallel job using its job ID, that is, it treats different processes of a parallel job as the same serial job. We illustrate this below using Figure 3.2. Now assume the jobs labelled 1, 2, 3 and 4 are four processes of a parallel job, whose ID is set to be 1. Figure 3.2 can be transformed to Figure 3.3 after deleting the same graph nodes in each level (the edges are omitted). Compared with Figure 3.2, it can be seen that the number of graph nodes in Figure 3.3 is reduced. Therefore, the number of subpaths that need to be examined and consequently the time spent in finding the optimal solution is significantly reduced.

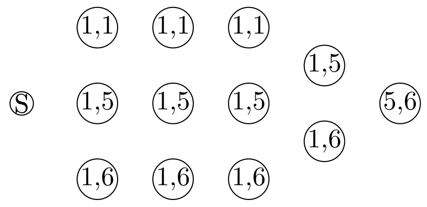


Figure 3.3: The graph model for a mix of serial and parallel jobs

We now present the O-SVPPE (Optimal SVPPE) algorithm, which adjusts SVPPE so that it can find the shortest valid path in the optimized co-scheduling graph. The only difference between O-SVPPE and SVPPE is that a different mechanism is used to find 1) the next valid level and 2) a valid node in a valid level for parallel jobs.

Lines 6-9 in Algorithm 3.1 capture the mechanism used by SVPPE to find the next valid level. In O-SVPPE, for a given level l, if job l is a serial job, the condition of determining whether level l is valid is the same as that in SVPPE. However, since the same job ID is now used to label all processes of a parallel job, the condition of whether a job ID appears on the given subpath can no longer be used to determine a valid level for parallel jobs. The revised method is discussed next.

Several new attributes are added for the optimized graph model. $proc_i$ denotes the number of processes that parallel job p_i has. For a given subpath $v.path$, $v.proc_i$ is the number of times a process of parallel job p_i appears on $v.path$. $v.jobset$ is now a bag (not a set) of job IDs that appear on $v.path$, that is, there are $v.proc_i$ instances of that parallel job in $v.jobset$. As in the case of serial jobs, the adjusted $v.jobset$ is used to determine whether two subpaths consist of the same set of jobs (and parallel processes). A new attribute, $node_k.jobset$, is also added to a graph node $node_k$, where $node_k.jobset$ is also a bag of job IDs that are in $node_k$. $node_k.proc_i$ is the number of processes of parallel job p_i that are in $node_k$. $node_k.serialjobset$ is a set of all serial jobs in $node_k$.

Algorithm 3.3: The O-SVPPE algorithm

```
1:   O-SVPPE(Graph)
2-6:   ... //same as Line 2-6 in Algorithm 1;
7:       if job $l$ is a serial job
8-10:      ...// same as Line 7-9 in Algorithm 1;
11:      else if $v.proc_l < proc_l$
12:        $valid_l = l$;
13:        break;
14-15:   ... //same as Line 10-11 in Algorithm 1
16:      if $node_k.serialjobset \cap v.jobset = \phi$ & $\forall p_i$, $v.proc_i + node_k.proc_i \leq proc_i$
17-48:       ... //same as Line13-44 in Algorithm 2
```

The following condition is used to determine whether a level is a valid level for a given path. Assume job l is a parallel job. For a given subpath $v.path$, level l (l starts from $v.level + 1$) is a valid level if $v.proc_l < proc_l$. Otherwise, level l is not a valid level.

After a valid level is found, O-SVPPE needs to find a valid node in that level. When there are both parallel and serial jobs, O-SVPPE uses two conditions to determine a valid node: 1) the serial jobs in the node do not appear in $v.jobset$, and 2) $\forall$ parallel job p_i in the node, $v.proc_i + node_k.proc_i \leq proc_i$.

O-SVPPE is outlined in Algorithm 3.3, in which Lines 7-13 implement the way of finding a valid level and Line 16 checks whether a node is valid, as discussed above.

3.5.2 Co-scheduling PC jobs

We now extend the SVPPE algorithm to handle PC jobs, which is called SVPPC (SVP for PC jobs). We first adjust SVPPE to handle PC jobs. Moreover, since the further optimization technique developed for PE jobs, i.e., the O-SVPPE algorithm, presented in Subsection 3.5.1 cannot be directly applied to PC jobs, the O-SVPPE algorithm is extended to handle PC jobs in Subsection 3.5.2, and termed O-SVPPC.

Recall that the communication time, $c_{ij,S}$, can be modelled with Eq. 3.10. We now adjust SVPPE to incorporate the PC jobs. In the graph model for serial and PE jobs, the weight of a graph node is calculate by summing up the weights of the individual jobs/processes, which is the performance degradation. When there are PC jobs, a process belongs to a PC job, the weight of a process p_{ij} in a PC job should be calculated by Eq. 3.10 instead of Eq. 3.1. The remainder of the SVPPC algorithm is exactly the same as SVPPE.

Communication-aware process condensation for optimizing SVPPC

The reason why the process condensation technique developed for PE jobs cannot be directly applied to PC jobs is because different processes in a PC job may have different communication patterns and therefore cannot be treated as identical processes. After carefully examining the characteristics of the typical inter-process communication patterns, a communication-aware process condensation technique is developed to accelerate the solving process of SVPPC, which is called O-SVPPC (Optimized SVPPC) in this chapter.

We can construct the co-scheduling graph model as we did in Figure 3.2 for finding the optimal solution of co-scheduling PC and serial jobs. We then define the *communication property* of a parallel job in a graph node as the number of communications that the processes of the parallel job in the graph node has to perform in each decomposition direction with other nodes. In the communication-aware process condensation, multiple graph nodes in the same level of the graph model can be condensed to one node if the following two conditions are met: 1) these nodes contain the same set of serial jobs and parallel jobs, and 2) the communication properties of all PCs in these nodes are the same.

We now present an example to illustrate the communication-aware process condensation for SVPPC. Consider Figure 3.1 again. We construct the co-scheduling graph model as we did in Figure 3.2 for finding the optimal solution for co-scheduling the 9 processes in p_1 and a serial job p_2 on 2-core machines. The IDs for the 9 processes are labelled as 1, ..., 9 and that of p_2 as 10. Figure 3.4 shows all graph nodes in the first level of the graph model (the level is drawn horizontally to save space). Node $\langle 1, 2\rangle$ means that processes p_{11} and p_{12} are co-scheduled to the same physical machine. According to the communication pattern shown in Figure 3.1, node $\langle 1,2\rangle$ will have one x-direction communication with other nodes in the graph model in any possible co-scheduling solution due to the communication between processes p_{12} and p_{13}. Similarly, due to the communications between p_{11} and p_{14} and between p_{12} and p_{15}), node $\langle 1,2\rangle$ will have two y-direction communications with other nodes. We define the *communication property* of a parallel job in a graph node as the number of communications that the processes of the parallel job in the graph node has to perform in each decomposition direction with other nodes. Then the communication property of p_1 in node $\langle 1,2\rangle$, termed (c_x, c_y) (since it is the 2-D decomposition), is (1, 2). Similarly, we can calculate the communication property of p_1 in other nodes, which are shown under the corresponding nodes in Figure 3.4.

According to the communication-aware process condensation technique, the level of nodes shown in the top part of Figure 3.4 can be condensed into the form shown in the bottom part of the figure (i.e., node $\langle 1,7\rangle$ and $\langle 1,9\rangle$ are condensed with $\langle 1,3\rangle$. Consequently, the number of nodes in the graph model is reduced, and therefore the solving process is accelerated.

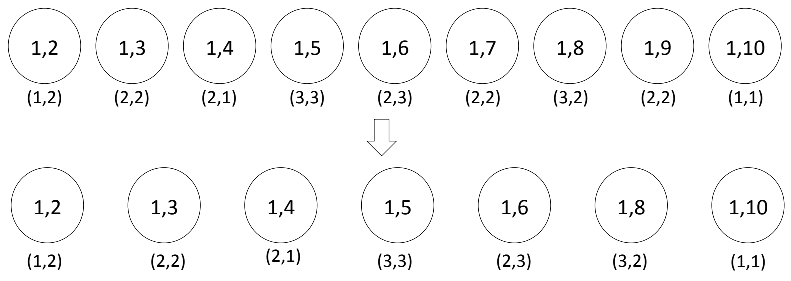


Figure 3.4: The example of communication-aware process condensation

3.6 Co-scheduling jobs on multi-processor computers

In order to add more cores to a multicore computer, there are two general approaches: 1) increase the number of cores on a processor chip and 2) install more processors, with the number of cores in each processor remaining unchanged; both approaches are often simultaneously applied.

The co-scheduling graph previously presented is for multicore machines each of which contains a single multi-core processor, which we term a single processor multicore machine (or a single processor for short). If there are multiple multicore processors in a machine (which we term a multi-processor machine) the resource contention, such as cache contention, is different. For example, only the cores on the same processor share the Last-Level Cache (LLC) on the chip, while the cores on different processors do not compete for cache. In a single processor machine, the job-to-core mapping does not affect the tasks' performance degradation. This is not the case in a multi-processor machine, as illustrated in the following example.

Consider a machine with two dual-core processors (processors p_1 and p_2) and a co-run group with 4 jobs ($j_1, ..., j_4$). Now consider two job-to-core mappings. In the first mapping, jobs j_1 and j_2 are scheduled on processor p_1, while j_3 and j_4 are scheduled on p_2. In the second mapping, jobs j_1 and j_3 are scheduled on processor p_1, while j_2 and j_4 are scheduled on p_2. The two mappings may generate different total performance degradations for this co-run group. In the co-scheduling graph in previous sections, a graph node corresponds to a possible co-run group in a machine, which is associated with a single performance degradation value. This holds for a single processor machine. As shown in the above discussions, however, a co-run group may generate different performance degradations in a multi-processor machine, depending on the job-to-core mapping within the machine. This subsection presents how to adjust the methods presented in previous sections to find the optimal co-scheduling solution in multi-processor machines.

A straightforward method is to generate multiple nodes in the co-scheduling graph for a possible co-run group, with each node having a different weight that equals a different performance degradation value (which is determined by the specific job-to-core mappings). We call this method MNG (Multi-Node for a co-run Group) method. For a machine with p processors with each processor having u cores, it can be calculated that there are $\frac{\prod_{i=0}^{p-1}\binom{(p-i)\cdot u}{u}}{p!}$ different job-to-core mappings that may produce different performance degradations. The algorithms presented in previous sections can be used to find the shortest path in this co-scheduling graph, where the shortest path must correspond to the optimal co-scheduling solution on the multi-processor machines. In this straightforward solution, however, the scale of the co-scheduling graph (i.e., the number of graph nodes) increases $\frac{\prod_{i=0}^{p-1}\binom{(p-i)\cdot u}{u}}{p!}$ fold, and consequently the solving time increases significantly compared with that for the case of single processor machines.

We now propose a method, called the Least Performance Degradation (LPD) method, to construct the new co-scheduling graph. Using this method, the optimal co-scheduling solution for multi-processor machines can be computed without increasing the scale of the co-scheduling graph. The LPD method is explained below.

As discussed above, in the case of multi-processor machines, a co-run group may produce different performance degradation in a multi-processor machine. Instead of generating multiple nodes (each being associated with a different weight, i.e., a different performance degradation value) in the co-scheduling graph for a co-run group, the LPD method constructs the co-scheduling graph for multi-processor machines in the following way: *A node is generated for a co-run group and the weight of the node is set to be the smallest performance degradation among all possible performance degradations generated by the co-run group. The remainder of the construction process is exactly the same as that for the case of single processor machines.*

Assume the jobs are to be co-scheduled on multi-processor machines. Using the LPD method defined above to construct the co-scheduling graph, the algorithms that have been proposed to find the optimal co-scheduling solutions on single processor machines will still find the optimal co-scheduling solutions on multi-processor machines.

3.7 Co-scheduling multi-thread jobs

A parallel job considered so far in this chapter is one consisting of multiple processes, such as an MPI job. In this subsection, we adapt the proposed graph model and algorithms so that they can handle parallel jobs consisting of multiple threads, such as OpenMP jobs. We call the former parallel jobs Multi-Process Parallel (MPP) jobs and the latter Multi-Thread Parallel (MTP) jobs.

In the co-scheduling graph, a thread in an MTP job is treated in the same way as a parallel process in an MPP job. Compared with MPP jobs, however, MTP jobs have the following different characteristics: 1) multiple threads of a MTP job must reside in the same node, and 2) the communication time between threads can be largely ignored. Accordingly, the co-scheduling graph model is adjusted as follows to handle the MTP jobs. For each node (i.e., every possible co-run group) in the co-scheduling graph, we check whether all threads belonging to the MTP are on node. If not, the node is deleted from the graph since it does not satisfy the condition that all threads of a MTP job must reside in the same node. We call the above process the validity check for MTP jobs.

Since the communication time between the threads in MTP jobs can be ignored, the performance degradation of a MTP job can be calculated using Eq. 3.7 that is used to compute the performance degradation of a PE job. Also, since the communication time of an MTP job is not considered, an intuitive method to find the optimal co-scheduling solution in the presence of MTP jobs is to use the algorithm for handling PE jobs, i.e., Algorithm 3.3. However, after closer inspection into the features of MTP jobs, it is apparent that Algorithm 3.3 can be adjusted to improve the performance of managing MTP jobs, a feature which is explained next.

After the validity check for MTP jobs, all threads belonging to a MTP job must only appear in the same graph node. Therefore, there is no need to perform the process condensation as we do in the presence of PE jobs. Consequently, the SVPPE algorithm (i.e., Algorithm 3.2) can be used to handle MTP jobs. Next, when the current path expands to a new node in the SVPPE Algorithm, for each parallel job p_i in the new node, SVPPE needs to check whether p_i appears in the current path. However, all threads in a MTP job only reside in the same node. Therefore, if a new node that the current path tries to expand to contains an MTP job, it is unnecessary to check whether threads of the MTP job appear in the current path.

In order to differentiate this approach from SVPPE, the algorithm for finding the optimal co-scheduling solution for the mix of serial and MTP jobs is denoted as SVPPT (where T stands for thread). The only difference between SVPPT and SVPPE is that Lines 15-17 in SVPPE (i.e., Algorithm 3.2) are removed from SVPPT.

From the above discussions, it is possible to determine if it would be much more efficient to find the optimal co-scheduling solution for MTP jobs than for PE jobs. This is because 1) the number of nodes in the co-scheduling graph for SVPPT is much less than that for PE jobs (because of the validity check for MTP jobs) and 2) SVPPT does not execute Lines 15-17 in SVPPE.

Note that the method discussed above for handling MTP jobs is applicable to both single processor machines and multi-processor machines, as defined previously.

3.8 The A*-search-based algorithm

The dismiss strategy designed for the SVP algorithm in Subsection 3.4.1 and the optimization strategies developed in O-SVPPE and O-SVPPC can avoid unnecessary searches in the co-scheduling graph. It has been shown that the A*-search algorithm is also able to find the optimal solution and during the searching, effectively prune the graph branches that will not lead to the optimal solution. In order to further accelerate the solving process, an A*-search-based algorithm is developed in this section to combine the ability of avoiding the unnecessary searches in the traditional A*-search algorithm and the algorithms presented in this chapter so far (SVP, O-SVP, O-SVPPE and O-SVPPC).

This section presents how to design the A*-search-based algorithm to find the optimal co-scheduling solution in the co-scheduling graph. We only consider the co-scheduling of serial and PC jobs for the sake of generality. The presented A*-search-based algorithm is called SVPPC-A*. SVP-A* (i.e., co-scheduling serial jobs), SVPPE-A* (i.e., co-scheduling both serial and PE jobs) and SVPPT-A* can be developed in a similar way.

3.8.1 Traditional A*-search algorithm

Given a graph in which every node has a weight, the objective of the A*-search algorithm is to find the shortest path (the path with the smallest total weight) from the start to the end node. In A*-search, each node v has two functions, denoted as $g(v)$ and $h(v)$. Function $g(v)$ computes the actual length from the start node to the node v. Function $h(v)$ estimates the length of the shortest path from node v to the end node. Therefore, the sum of $g(v)$ and $h(v)$, denoted as $f(v)$, is the estimated length of the shortest path from the start to the end node that passes the node v. Initially, the priority list contains only the start node. Each iteration, A*-search removes the node with the shortest distance from the priority list and expands that node. The algorithm then uses the $f(v)$ function to estimate the length of all the nodes generated by expansion, and inserts them into the priority list according to their distances, which is $f(v)$. This process terminates when the node with the shortest distance is the end node, which indicates that a shortest path has been found. Note that when setting $h(v)$ to be 0, the A*-search algorithm is equivalent to Dijkstra's algorithm for finding the shortest path.

The A*-search based algorithm has the same structure and logic as the SVP algorithm. However, in the A*-search based algorithm, each node v has two functions, denoted as $g(v)$ and $h(v)$. Function g(v) computes the actual length from the start node to the node v. Function $h(v)$ estimates the length of the shortest path from node v to the end node. Therefore, the sum of $g(v)$ and $h(v)$, denoted as $f(v)$, is the estimated length of the shortest path from the start to the end node that passes node v.

3.8.2 SVPPC-A*

The traditional A*-search algorithm, which is briefly overviewed in the supplementary notes, cannot be directly applied to obtain the optimal co-scheduling solution for the same reasons discussed in the presentation of the SVP and the SVPPE algorithms; namely, i) the optimal co-scheduling solution in the constructed co-scheduling graph corresponds to the shortest *valid* path, not the shortest path, and ii) since the jobs to be scheduled contain parallel jobs, the distance of a path is not the total weights of the nodes on the path, as calculated by the traditional A*-search algorithm.

Three functions are defined in the traditional A*-search algorithm. Function $g(v)$ is the actual distance from the start node to node v and $h(v)$ is the estimated length from v to the end node, while $f(v)$ is the sum of $g(v)$ and $h(v)$. In SVPPC-A*, we use the exactly same methods proposed for the SVP algorithm (i.e., the dismiss strategy) to handle and expand the valid subpaths and avoid the unnecessary searches. Also, we use the method proposed for the SVPPC algorithm to calculate the distance of the subpaths (i.e., Eq. 3.7 and Eq. 3.10) that contain the PC jobs. This technique can be used to obtain the value of $g(v)$. Note that the communication-aware process condensation technique proposed in Subsection 3.5.2 can also be used to accelerate SVPPC-A*.

The estimation of $h(v)$ is one of the most critical parts in designing an A*-search algorithm. The following two properties reflect the importance of $h(v)$

[61]: i) The result of an A* search is optimal if the estimation of $h(v)$ is not higher than the lowest cost to reach the end node, and ii) the closer the result of $h(v)$ is from the lowest cost, the more effective A* search is in pruning the search space.

Therefore, in order to find the optimal solution, the $h(v)$ function must satisfy the first property. In our problem, if there are q jobs on the path corresponding to $g(v)$, then the aim of setting the $h(v)$ function is to find a function of the remaining $n - q$ jobs such as the value of the function is less than the shortest distance from node v to the end node. The following two strategies are proposed to set the $h(v)$ function.

Strategy 1 for setting $h(v)$: Assume node v is in level l, we construct a set R that contains all the nodes from $l + 1$ to the last level in the co-scheduling graph, and sort these nodes in ascending order of their weights. Then, regardless of the validity, the first $(n - q)/u$ (u is the number of cores) nodes are selected from R to form a new subpath; the distance of this subpath is $h(v)$.

Strategy 2 for setting $h(v)$: Assume node v is in level l. We find all valid levels from level $l + 1$ to the last level in the co-scheduling graph. The total number of valid levels obtained must be $(n - q)/u$. We then obtain the node with the least weight from each valid level. $(n - q)/u$ nodes will be obtained. We use these $(n - q)/u$ nodes to form a new subpath and use its distance as $h(v)$.

It is easy to prove that $h(v)$, obtained through the above strategies, must be less than the actual shortest distance from v to the end node; this is the case because it uses the nodes with the smallest weights from all remaining nodes in Strategy 1 or from all valid levels in Strategy 2. We will show in the experiments that Strategy 2 is much more effective than Strategy 1 in terms of pruning unnecessary searches.

3.8.3 Case studies for the A*-search based algorithm

Strategy 1

In this example, we use all nodes from Figure 3.2. First, consider the node $\langle 1, 2\rangle$ in the first level of the graph, the function $g(v)$ corresponding to this node is the actual distance from the start node. In this example, $g(v) = 7$. To compute function $h(v)$, the algorithm first creates a set that contains all nodes from the next level, which is level 2 in this example, to the last level. The algorithm then sorts these nodes in ascending order of their weights. The function $h(v)$ is computed by adding together the weights of the first $(n - q)/u$ nodes from this set. In this example, $(n - q)/u$ is 2 and therefore the first two nodes, which are $\langle 2, 5\rangle$, $\langle 3, 5\rangle$ with weights 1 and 3, are selected to calculate $h(v)$. Therefore, $h(v) = 4$, and $f(v) = g(v) + h(v) = 11$ for node $\langle 1, 2\rangle$.

Strategy 2

In strategy 2, we change the way we compute the function $h(v)$. In this strategy, the $h(v)$ function is computed in the following way. The algorithm first finds all valid levels and then selects the node with the least weight from each valid level and adds their weights together.

Consider the node $\langle 1, 2\rangle$ in Figure 3.2. As in the previous example, $g(v) = 7$. To compute $h(v)$, the algorithm finds the first valid level, which is level 3, and selects node $\langle 3, 5\rangle$, since this node has the smallest weight in level 3. The next valid level is level 4, and the selected node is $\langle 4, 6\rangle$. Therefore, $h(v) = 1 + 4 = 5$, and $f(v) = 7 + 5 = 12$.

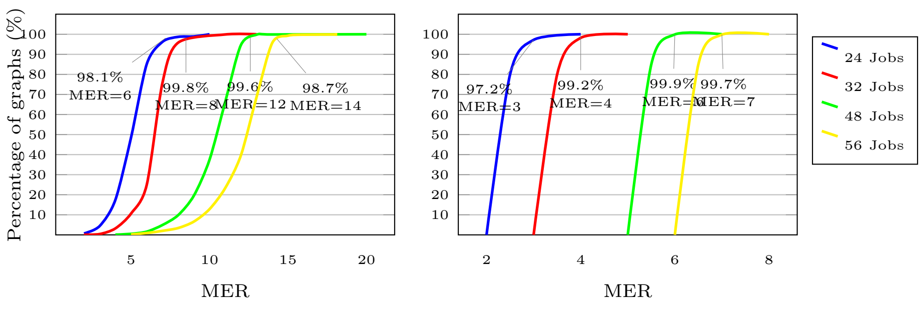
This example also shows that Strategy 2 provides a tighter estimation bound of $f(v)$ than Strategy 1.

3.9 Heuristic A*-search Algorithm

In this section, a heuristic method is proposed to further trim the searching in the co-scheduling graph. The resulting search space is much smaller than the original one and therefore the co-scheduling solution, which is sub-optimal, can be computed more efficiently by order of magnitude than the extended A*-search algorithm proposed in previous section to find the optimal solution, which we now call the *Optimal A*-search* (OA*) method.

The principle of trimming the co-scheduling graph is based on the following insight. We re-arrange the nodes in each level of the co-scheduling graph in the ascending order of node weight. We then apply OA* to find the shortest path from the sorted graph. For each node on the shortest path, we record its rank in the graph level that the node is in (the i-th node in a level has the rank of i). Assume that a node on the computed shortest path has the rank of i in its level. We also record how many invalid nodes the algorithm has to skip from rank 1 to rank i in the level before locating this valid node of rank i. Assume the number of invalid nodes is j. Then (i-j) is the number of nodes that the algorithm has attempted in the level before reaching the node that is on the shortest path. We call this number, $i - j$, the *effective rank* of the node of rank i in the level. We calculate the effective rank for every node on the shortest path and obtain the maximum of them, which we denote by MER (Maximum Effective Rank of the shortest path). If we had known the value of MER, assuming it is k, before we apply the OA*-search algorithm, we can instruct the algorithm to only attempt the first k valid nodes in each level and the algorithm will still be able to find the shortest path of the algorithm.

Given the above insight, we designed the following benchmarking experiment to conduct the statistical analysis for the value of MER. The numbers of jobs we used are 24, 32, 48 and 56 jobs. For each job batch and u-core machines, we randomly generated K different cache misses for a job (the cache miss rate of a job is randomly selected from the range of [15%, 75%]) and construct K different co-scheduling graphs. We then used OA* to find the shortest path of each graph and record the value of MER. Figure 3.5a and 3.5b depict the Cumulative Distribution Functions (CDF) of MER with 1000 graphs (i.e., K=1000) on Quad-core machines and 8-core machines, respectively.



(a) On Quad-core machines

(b) On 8-core machines

Figure 3.5: Cumulative Distribution Function (CDF) of MER

As shown in Figure 3.5a, when the number of jobs is 24, the value of MER is no more than 6 for 98.1% of graphs. Similarly, when the numbers of jobs are 32, 48 and 56, the values of MER are no more than 8, 12 and 14 for 99.8%, 99.6% and 98.7% of graphs, respectively. The corresponding figures for the case where the jobs are co-scheduled on 8-core machines are shown in Figure 3.5b.

From these benchmarking results, we find that we can use the function $MER = \frac{n}{u}$, where n is the number of jobs and u is the number of cores in a machine, to predict the value of MER. With this MER function, the actual value of MER will be no more than the predicted one in almost all cases. The reason why such a MER function generates very good results can be explained in principle as follows, although we found that it was difficult to give rigorous proof. We know that the nodes with too big weights have less chance to appear on the shortest path. Therefore, when OA* attempts the nodes in a level to expand the current path, if a node's effective rank is more than $\frac{n}{u}$, which is the number of machines that is needed to co-run this batch of jobs, the node will not be selected even if a poor greedy algorithm is used to map the node to one of the $\frac{n}{u}$ machines.

Based on the above statistical analysis, we adjust the co-scheduling graph and trim the searching for the shortest path in the following way. In each level of the graph, the nodes are arranged in the ascending order of node weight. When OA* searches for the shortest path, it only attempts $\frac{n}{u}$ valid nodes in each level to expand the current path, if $\frac{n}{u}$ is less than the number of valid nodes in the level. This way, the graph scale and consequently the number of operations performed by the algorithm are reduced by order of magnitude. We call the A*-search algorithm operating in this fashion the *Heuristic A*-search* (HA*) algorithm.

It is difficult to analyze the time complexity of the A*-search algorithm since it depends on the design of the $h(v)$ function. However, the time complexity of our A*-search algorithm mainly depends on how many valid nodes the algorithm have to attempt when extending a current path to a new graph level, which can be used to analyze the complexity difference between OA* and HA*. Assume that OA* is searching for co-scheduling solutions of n jobs on u-core machines and that the current path includes k nodes. When OA* extends the current path to a new level, the number of valid nodes that OA* may have to attempt in the new level can be calculated by $\binom{(n-1)-k\cdot u}{u-1}$, since all nodes that contain the jobs that appear in the current path are not valid nodes. Under the same assumptions, the number of valid nodes that HA* needs to attempt is only $\frac{n}{u}$. The following example is given to show the difference between these two numbers. When n is 100, u is 4 and k is 2, $\binom{(n-1)-k\cdot u}{u-1}$ is 121485 while $\frac{n}{u}$ is 25. $\binom{(n-1)-k\cdot u}{u-1}$ is only less than 25 when k is bigger than 23 ($\binom{(n-1)-k\cdot u}{u-1}$ is 35 when k is 23). But the biggest value of k is 24 since there are total 25 nodes in a complete path when co-scheduling 100 jobs on Quad-core machines. This means that in almost all cases (except the last graph level) $\binom{(n-1)-k\cdot u}{u-1}$ is bigger than $\frac{n}{u}$ by orders of magnitude. This is the reason why HA* is much more efficient than OA*.

3.10 Clustering approximation for finding the shortest valid path

We have presented methods and optimization strategies for solving the graph model for the shortest valid path. In order to further shorten the solving time and strike a balance between solving efficiency and solution quality, this section proposes a flexible technique called the *clustering* technique, to rapidly find an approximate solution. The clustering technique is flexible because the solving efficiency can be adjusted by setting the desired solution quality. It can be applied to both O-SVP, O-SVPPE and O-SVPPC.

As discussed in introduction and related work, the reason why co-scheduling causes performance degradation is because the co-running jobs compete for shared cache. SDC (Stack Distance Competition) is a popular technique for calculating the impact when multiple jobs are co-running; this uses the SDPs (Stack Distance Profile) of the multiple jobs as input. Therefore, if two jobs have similar SDPs, they will have a similar effect on other co-running jobs. The fundamental idea of the proposed clustering technique is to class the jobs with similar SDPs together and treat them as the same job. Reflected in the graph model, the jobs in the same class can be given the same job ID. In so doing, the number of different nodes in the graph model will be significantly reduced. The resulting effect is the same as when different parallel processes are given the same job ID in the O-SVPPE algorithm in Subsection 3.5.1.

We now introduce a method of measuring the SDP similarity between two jobs. Given a job j_i, its SDP is essentially an array, in which the k-th element records the number of cache hits on the k-th cache line (which is denoted by $h_i[k]$). The following formula is used to calculate the Similarity Level (SL) in terms of SDP when comparing another job j_j against j_i.

$$SL = \frac{\sqrt{\sum_{k=1}^{cl}(h_i[k] - h_j[k])^2}}{\sum_{k=1}^{cl} h_i[k]} \tag{3.13}$$

When SL is larger, more jobs will be classed together. Consequently, there will be fewer nodes in the graph model and hence less scheduling time is needed to calculate an accurate solution.

The O-SVP clustering algorithm is the same as the O-SVP algorithm except in the way a valid level is found; finding a valid node in a valid level is the same as that for O-SVPPE (Algorithm 3.3). The clustering technique can also be applied to O-SVPPE and O-SVPPC in a similar way. Detailed discussion of this process is not repeated.

3.11 Evaluation

This section evaluates the effectiveness and the efficiency of the proposed methods: O-SVP, O-SVPPE, O-SVPPC, A*-search-based algorithms (i.e., SVPPC-A* and SVP-A*) and the clustering approximation technique. In order to carry out this evaluation, we compare the algorithms proposed in this chapter with existing co-scheduling algorithms proposed in [61]: Integer Programming (IP); Hierarchical Perfect Matching (HPM), and Greedy (GR).

We conduct the experiments with real jobs. Serial jobs are taken from the NASA benchmark suit NPB3.3-SER [6] and SPEC CPU 2000 [54]. NPB3.3-SER has 10 serial programs and each program has 5 different problem sizes. The problem size used in the experiments is size *C*. The PC jobs are selected from the ten MPI applications in the NPB3.3-MPI benchmark suite. As for PE jobs, 5 embarrassingly parallel programs are used: PI [77], Mandelbrot Set(MMS) [91], RandomAccess(RA) from the HPCC benchmark [89], EP from NPB-MPI [6] and Markov Chain Monte Carlo for Bayesian inference (MCM) [72]. In all these 5 embarrassingly parallel programs, multiple slave processes are used to perform calculations in parallel and a master process reduces the final result after it gathers the results from all slaves. This set of parallel programs are selected because they contains both computation-intensive (e.g, MMS and PI) and memory-intensive programs (e.g, RA).

Four types of machines, Dual core, Quad core, 8 core and 16 core machines, are used to run the benchmarking programs. The dual-core machine has an Intel Core 2 Dual processor and each core has a dedicated 32KB L1 data cache and a 4MB 16-way L2 cache shared by the two cores. The Quad-core machine has an Intel Core i7 2600 processor and each core has a dedicated 32KB L1 cache and a dedicated 256KB L2 cache. A further 8MB 16-way L3 cache is shared by the four cores. The 8 core machine has two Intel Xeon L5520 processors with each processor having 4 cores. Each core has a dedicated 32KB L1 cache and a dedicated 256KB L2 cache, and an 8MB 16-way L3 cache shared by 4 cores. The 16 core machine has two Intel Xeon E5-2450L processors with each processor having 8 cores. Each core has a dedicated 32KB L1 cache and a dedicated 256KB L2 cache, and a 16-way 20MB L3 cache shared by 8 cores. The network interconnecting the dual-core and quad-core machines is a 10 Gigabit Ethernet, while the network interconnecting the 8-core and 16-core Xeon machines is QLogic TrueScale 4X QDR InfiniBand. In the remainder of this section, we label the 8 core and 16 core machines as 2*4 core and 2*8 core machines, to highlight the fact that they are dual-processor machines.

The single-run computation times of the benchmarking programs are measured. Then the method presented in [104] is used to estimate the co-run computation times of the programs, According to [104], CPU_Time is calculated using Eq.3.14.

$$\begin{split} CPU\_Time = (CPU\_Clock\_Cycle + \\ Memory\_Stall\_Cycle) \times Clock\_Cycle\_Time \end{split} \tag{3.14}$$

$Memory\_Stall\_Cycle$ in Eq.3.14 is computed by Eq.3.15, where $Number\_of\_Misses$ is the number of cache misses.

$$\begin{split} Memory\_Stall\_Cycle = Number\_of\_Misses \times \\ Miss\_Penalty \end{split} \tag{3.15}$$

The values of CPU_Clock_Cycle and $Number\_of\_Misses$ for a single-run program can be obtained using *perf* [1] in Linux. Then the value of $Memory\_Stall\_Cycle$ for a single-run program can be obtained by Eq.3.15. CPU_Time for a single-run program can also be obtained by *perf.*

The value of CPU_Time for a co-run program can be estimated in the following way. We use the *gcc-slo* compiler suite [9] to generate the SDP (Stack Distance Profile) for each benchmarking program offline, and then apply the SDC (Stack Distance Competition) prediction model in [19] to predicate $Number\_of\_Misses$ for the co-run programs. Then Eq.3.15 and Eq.3.14 are used to estimate $Memory\_Stall\_Cycle$ and CPU_Time for the co-run programs.

With the single-run and co-run values of CPU_Time, Eq.3.1 is used to compute the performance degradation.

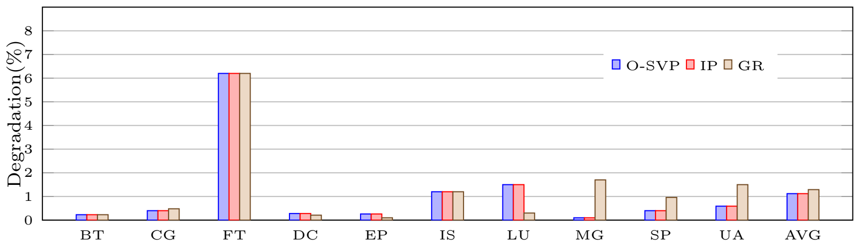
In order to obtain the communication time of a parallel process when it is scheduled to co-run with a set of jobs/processes, i.e., $c_{ij,S}$ in Eq.3.11, we examined the source codes of the benchmarking MPI programs used for the experiments and obtained the amount of data that the process needs to communicate with each of its neighbouring processes (i.e., $\alpha_{ij}(k)$ in Eq.3.11). Then Eq.3.11 is used to calculate $c_{ij,S}$.

3.11.1 Evaluating the O-SVP algorithm

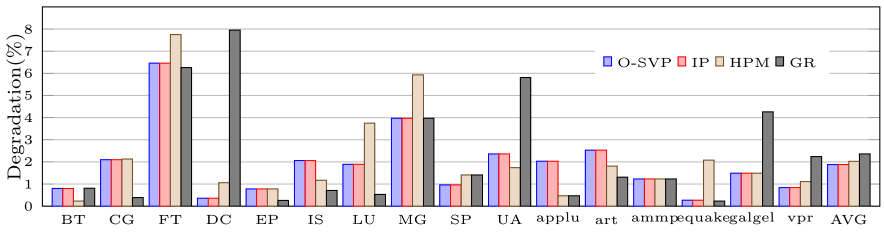
In this subsection, we compare the O-SVP algorithm with the existing co-scheduling algorithms in [61].

These experiments use all 10 serial benchmark programs from the NPB-SER suite. The results are presented in 3.6a and 3.6b, which show the performance degradation of each of the 10 programs plus their average degradation under different co-scheduling strategies on Dual-core and Quad-core machines.

The work in [61] shows that IP generates the optimal co-scheduling solutions for serial jobs. As can be seen from Figure 3.6a, O-SVP achieves the same average degradation as that under IP. This suggests that O-SVP can find the optimal co-scheduling solution for serial jobs. The average degradation produced by GR is 15.2% worse than that of the optimal solution. It can also been seen from Figure 3.6a that the degradation of FT is the biggest among all 10 benchmark programs. This may be because FT is the most memory-intensive



(a) Dual Core



(b) Quad Core

Figure 3.6: Comparing the degradation of serial jobs under O-SVP, IP, HPM and GR

program among all, and therefore endures the biggest degradation when it has to share the cache with others.

Figure 3.6b shows the results on Quad-core machines. In this experiment, in addition to the 10 programs from NPB-SER, 6 serial programs (applu, art, ammp, equake, galgel and vpr) are selected from SPEC CPU 2000. In Figure 3.6b, O-SVP produces the same solution as IP, which shows the optimality of O-SVP. Also, O-SVP finds the better co-scheduling solution than HPM and GR. The degradation under HPM is 7.7% worse than that under O-SVP, while that of GR is 25.5% worse. It is worth noting that O-SVP does not produce the least degradation for all programs. The aim of O-SVP is to produce minimal total degradation. This is why O-SVP produced bigger degradation than GR and HPM in some cases.

3.11.2 The O-SVPPE algorithm

This section first reports the results for validating the optimality of O-SVPPE proposed in this chapter. We first compare O-SVPPE with the IP model developed for co-scheduling mix of serial and PE jobs on Dual-core and Quad-core machines. In our experiments, we employ the IP solver, CPLEX [3], to compute the optimal co-schedules. The experiments use those 5 embarrassively parallel programs and the serial jobs are from NPB-SER plus art from SPEC CPU 2000. The results are listed in Table 3.2. In these experiments, the processes of each parallel application vary from 2 to 4. The detailed combinations of serial and parallel programs are listed below:

- In the case of 8 processes, PI and RA are combined with DC, UA, BT and IS.
- In the case of 12 processes, PI and RA are combined with applu, art, DC, UA, BT and IS.
- In the case of 16 processes, PI and RA are combined with applu, art, ammp, vpr, DC, UA, BT and IS.

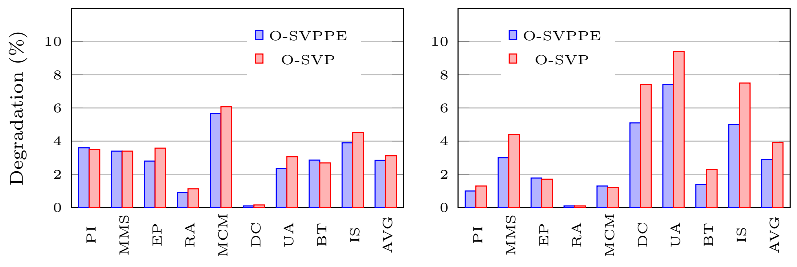
Table 3.2: Comparison of IP and O-SVPpe for serial and parallel jobs

Number of Jobs	Average Degradation (%)			
	Dual Core		Quad Core	
	IP	O-SVPPE	IP	O-SVPPE
8	2.85	2.85	2.89	2.89
12	2.69	2.69	2.94	2.94
16	2.91	2.91	3.15	3.15

As can be seen from Table 3.2, O-SVPPE achieves the same performance degradation as that by the IP model. These results verify the optimality of O-SVPPE.

The reason why we propose O-SVPPE is because 1) none of the existing co-scheduling methods is designed for parallel jobs; 2) we argue that if applying the existing co-scheduling methods designed for serial jobs to schedule parallel jobs, it will not produce the optimal solution. In order to investigate the performance discrepancy between the method for serial jobs and that for PE jobs, we apply O-SVP to solve the co-scheduling for a mix of serial and parallel jobs and compare the results with those obtained by O-SVPPE. In the mix of serial and parallel jobs, the parallel jobs are those 5 embarrassively parallel programs and the serial jobs are from NPB-SER plus art from SPEC CPU 2000. The experimental results are shown in Figure 3.7a (for the Dual-core case) and 3.7b (for Quad-core), in which each parallel program has 10 processes.

As can be seen from the figures, SVPPE produces smaller average degradation than O-SVP in both Dual-core and Quad-core cases. In the Dual-core case, the degradation under O-SVP is worse than that under SVPPE by 9.4%, while in the Quad-core case, O-SVP is worse by 35.6%. These results suggest it is necessary to design the co-scheduling method for parallel jobs.



(a) Dual Core (b) Quad Core

Figure 3.7: Comparing the degradation under SVPPE and O-SVP for a mix of PE and serial benchmark programs

3.11.3 The O-SVPPC algorithm

This section first reports the results for validating the optimality of O-SVPPC proposed in this chapter. We first compare O-SVPPC with the IP model on Dual-core and Quad-core machines. In our experiments, we employ the IP solver, CPLEX [3], to compute the optimal co-schedules. The results are listed in Table 3.3. In these experiments, two MPI applications (i.e., MG-Par and LU-Par) are selected from the NPB3.3-MPI and combined with serial programs chosen from NPE-SER and SPEC CPU 2000. The processes of each parallel application vary from 2 to 4. The detailed combinations of serial and parallel programs are listed below:

- In the case of 8 processes, MG-Par and LU-Par are combined with applu, art, equake and vpr.

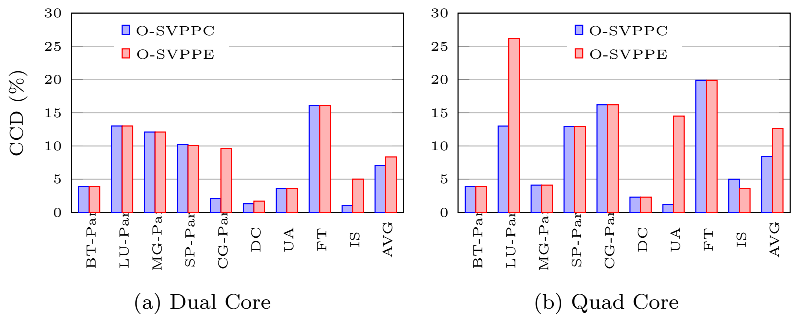
- In the case of 12 processes, MG-Par and LU-Par are combined with applu, art, ammp, equake, galgel and vpr.
- In the case of 16 processes, MG-Par and LU-Par are combined with BT, IS, applu, art, ammp, equake, galgel and vpr.

Table 3.3: Comparison of IP and O-SVPPC for serial and parallel jobs

Number of Jobs	Average Degradation (%)			
	Dual Core		Quad Core	
	IP	O-SVPPC	IP	O-SVPPC
8	0.07	0.07	0.098	0.098
12	0.05	0.05	0.074	0.74
16	0.12	0.12	0.15	0.15

As can be seen from Table 3.3, O-SVPPC achieves the same performance degradation as that by the IP model. These results verify the optimality of O-SVPPC.

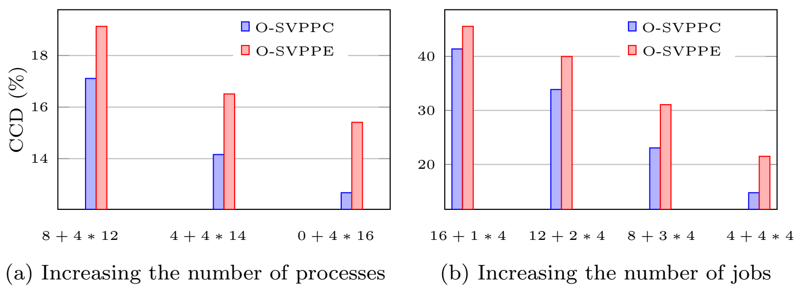
Figure 3.8a and 3.8b show the Communication-Combined Degradation (CCD) (i.e., the value of Eq.3.10) of the co-scheduling solution obtained by the SVPPC algorithm when the applications are run on Dual-core and Quad-core, respectively. In this set of experiments, 5 MPI applications (i.e., BT-Par, LU-Par, MG-Par, SP-Par and CG-Par) are selected from the NPB3.3-MPI suite and each parallel application is run using 10 processes, while the serial jobs remain the same as those used in Figure 3.7b. In order to demonstrate the effectiveness of the SVPPC, SVPPE is also used find the co-scheduling solution for the mix of MPI jobs and serial jobs, by ignoring the inter-process communications in the MPI jobs. We then use Eq.3.10 to calculate CCD of the co-scheduling solution obtained by SVPPE. The resulting CCD is also plotted in Figure 3.8a and 3.8b. As can be seen from these figures, the CCD under SVPPE is worse than that under SVPPC by 18.7% in Dual-core machines, while in Quad-core machines, the CCD obtained by SVPPE is worse than that by SVPPC by 50.4%. These results justify the need to specially develop the algorithm to find the co-scheduling solution for PC jobs.



(a) Dual Core (b) Quad Core

Figure 3.8: Comparing the Communication-Combined Degradation (CCD) obtained by SVPPC and SVPPE

We further investigate the impact on CCD as the number of parallel jobs or the number of parallel processes increases. The experimental results are shown in Figure 3.9a and 3.9b (on Quad-core machines). In Figure 3.9a, the number of total jobs/processes is 64. The number of parallel jobs is 4 (i.e., LU-Par, MG-Par, SP-Par and CG-Par) and the number of processes per job increases from 12 to 16. Other jobs are serial jobs. For example, 8+4*12 represents a job mix with 8 serial and 4 parallel jobs, each with 12 processes.



(a) Increasing the number of processes (b) Increasing the number of jobs

Figure 3.9: Impact of the number of parallel jobs and parallel processes

In Figure 3.9a, the difference in CCD between SVPPC and SVPPE becomes bigger as the number of parallel processes increases. This result suggests that SVPPE performs increasingly worse than SVPPC (increasing from 11.8% to 21.5%) as the proportion of PC jobs increases in the job mix. Another observation from this figure is that the CCD decreases as the proportion of parallel jobs increases. This is simply because the degradation experienced by multiple processes of a parallel job is only counted once. If those processes are the serial jobs, their degradations will be summed and is therefore bigger. In Figure 3.9b, the number of processes per parallel job remains unchanged and the number of parallel jobs increases. For example, 12+2*4 represents a job mix with 12 serial jobs and 2 parallel jobs, each with 4 processes. The detailed combinations of serial and parallel jobs are: i) In the case of 16+1*4, MG-Par is used as the parallel job and all 16 serial programs are used as the serial jobs; ii) In the case of 12+2*4, LU-Par and MG-Par are the parallel jobs and the serial jobs are SP, BT, FT, CG, IS, UA, applu, art, ammp, equake, galgel and vpr; iii) In the case of 8+3*4, BT-Par, LU-Par, MG-Par are parallel jobs and the serial jobs are SP, BT, FT, DC, IS, UA, equake, galgel; iv) In the case of 4+4*4, BT-Par, LU-Par, SP-Par, MG-Par are parallel jobs and the serial jobs are IS, UA, equake, galgel. The results in Figure 3.9b show the similar pattern as those in Figure 3.9a. The reasons for these results are also the same.

3.11.4 Scheduling in Multi-processor Computers

In this section, we investigate the effectiveness of the LPD method proposed to handle the co-scheduling in multi-processor machines. In the experiments, we first use the MNG method discussed in Section 3.6 (i.e., generating multiple graph nodes for a co-run group with each node having a different weight) to construct the co-scheduling graph. As we have discussed, from the co-scheduling graph constructed by the MNG method, the algorithm must be able to find the optimal co-scheduling solution for multi-processor machines. Then we use the LPD method to construct the graph and find the shortest path of the graph. The experimental results are presented in Figure 3.10a and 3.10b, in which a mix of 4 PE jobs (PI, MMS, RA and MCM, each with 31 processes) and 4 serial jobs (DC, UA, BT and IS) are used. It can be seen that the performance degradations obtained by two methods are the same. This result verifies that the algorithms can produce the optimal co-scheduling solutions using the LPD method.

Following the same logic as in Figure 3.7, we conducted the experiments to investigate the performance discrepancy between the method for serial jobs and that for PE jobs on multi-processor machines. The LPD method is used to generate the co-scheduling graphs (therefore, the "LPD" prefix is added to the algorithms in the legends in the figures). In these experiments, we use the same experimental settings as in Figure 3.10a and 3.10b. The results are shown in Figure 3.11a and 3.11b. As can be seen from the figures, LPD-SVPPE produces smaller average degradation than LPD-SVP in both 8-core and 16-core cases. In the 8-core case, the degradation under LPD-SVP is worse than that under LPD-SVPPE by 31.9%, while in the 16-core case, LPD-SVP is worse by 34.8%. These results verify the effectiveness of the LPD method for co-scheduling PE jobs.

Similarly, following the same logic as in Figure 3.8, we conducted the experiments to run PC jobs using SVPPC and SVPPE on multi-processor machines and compare the performance discrepancy in terms of CCD. The same experimental settings as in Figure 3.8 are used and the results are presented in Figure 3.12a and 3.12b. In this set of experiments, 4 MPI applications (i.e., BT-Par, LU-Par, MG-Par, and CG-Par) are selected from the NPB3.3-MPI suite and each parallel application is run using 31 processes, while the same serial jobs as in Figure 3.10a are used. As can be seen from these figures, the CCD under LPD-SVPPE is worse than that under LPD-SVPPC by 36.1% and 39.5% in 2*4-core and 2*8-core machines, respectively. These results justify the necessity of using SVPPC to handle PC jobs and show that the LPD method works well with the SVPPC algorithm.

As discussed in Section 3.6, the reason why we propose the LPD method is because using the MNG method, the scale of the co-scheduling graph increased significantly in multi-processor systems. The LPD method can reduce the scale of the co-scheduling graph and consequently reduce the solving time. Therefore, we also conducted the experiments to compare the solving time obtained by LPD and the MNG method. The experimental results are presented in Figure 3.13, in which Figure 3.13a and 3.13b are for PE and PC jobs, respectively. It can be seen from the figure that the solving time of LPD is significantly less than that of the straightforward method and the discrepancy increases dramatically as the number of jobs increases. These results suggest that LPD is effective in reducing solving time compared with the MNG method.

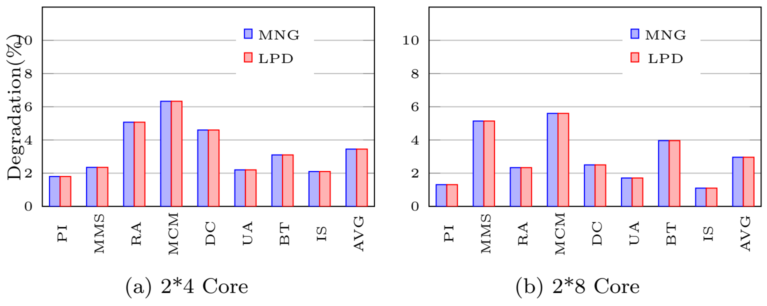


Figure 3.10: Comparing the degradation caused by the straightforward method and the LPD method

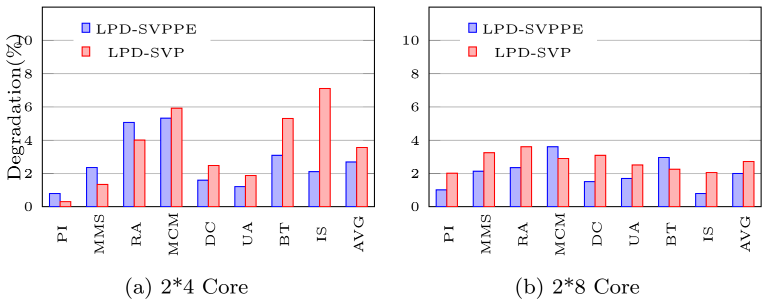


Figure 3.11: Comparing the degradation under LPD-SVP and LPD-SVPPE for a mix of PE and serial benchmark programs

3.11.5 Scheduling Multi-threading Jobs

In Section 3.7, in order to schedule the MTP jobs correctly, we need to guarantee that the threads from the same MTP job are scheduled in the same machine. In order to handle this, the SVPPT algorithm is proposed to construct the co-scheduling graph and find the shortest path. In this subsection, we first conduct

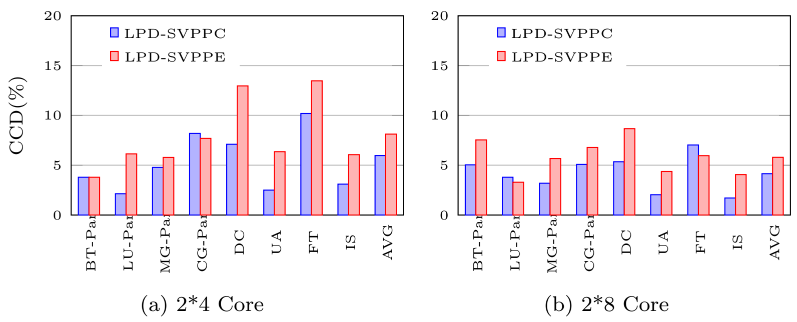


Figure 3.12: Comparing the Communication-Combined Degradation (CCD) obtained by LPD-SVPPC and LPD-SVPPE

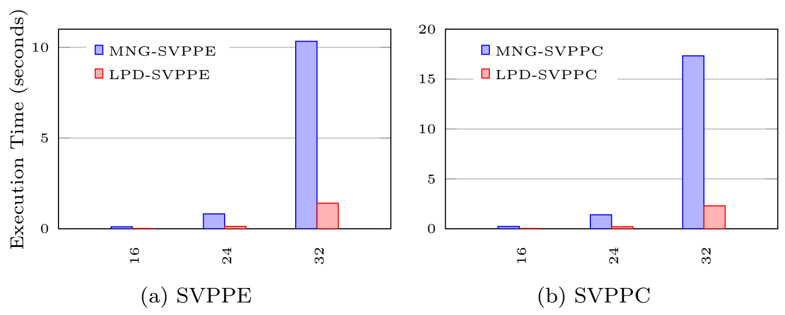


Figure 3.13: Comparing the solving times of the LPD and the MNG method, coupled with SVPPE and SVPPC

the experiments to examine the co-scheduling solution obtained by SVPPT. In the experiments, we chose 4 MTP programs (each with 2 threads on 4 Core and 3 threads on 8 Core) from NPB3.3-OMP (BT, MG, EP and FT) and 4 serial jobs from NPB-SER (DC, UA, LU and SP). The experiments are conducted on two types of processors, Xeon L5520 (4 cores) and Xeon E5-2450L (8 cores). The results are presented in Table 3.4. It can be seen that all threads from the same MTP program are mapped to the same machine, which verifies SVPPT can find correct co-scheduling solutions for MTP jobs.

As discussed in Section 3.7, SVPPT is supposed to be more efficient than SVPPE in finding the shortest path. Therefore, we also conducted the experiments to compare the solving time of SVPPT and SVPPE. The results are presented in Table 3.5. The experiments are conducted on 4-core and 8-core machines. It can be seen that SVPPT spends much less time than SVPPE and the gap increases as the number of jobs/threads increases. These results verify the efficiency of SVPPT.

Table 3.4: Schedule result for Multi-threading program

Processor	Jobs on each chip		
4 Core	bt,bt, ep,ep	mg,mg, lu, sp	ft,ft,dc, ua
8 Core	bt,bt,bt, ep,ep,ep, dc, sp		mg, mg,mg, ft,ft,ft,ua, lu

Table 3.5: Comparing the solving time between MTP and SVPPE

Number of Jobs	Solving Time (seconds)		Number of Jobs	Solving Time (seconds)	
	4 Core			8 Core	
	MTP	SVPPE		MTP	SVPPE
24	0.0011	0.0025	24	0.0013	0.0022
36	0.013	0.034	32	0.004	0.011
48	0.13	0.38	48	0.078	0.15
64	1.11	3.89	64	0.26	1.35

3.11.6 The A*-search-based algorithms

This section reports the results for validating the optimality of the proposed A*-search-based algorithms. We first compare the SVP-A* algorithm with the O-SVP algorithm in terms of the optimality in co-scheduling serial jobs. The experiments use all 10 serial benchmark programs from the NPB-SER suite and 6 serial programs (applu, art, ammp, equake, galgel and vpr) selected from SPEC CPU 2000. The experimental results are presented in Table 3.6. We also compare the SVPPC-A* algorithm with the O-SVPPC algorithm in terms of optimality in co-scheduling a mix of serial and parallel programs. The experiments are conducted on Quad-core machines. The results are listed in Table 3.7. In these experiments, 2 MPI applications (i.e., MG-Par and LU-Par) are selected from the NPB3.3-MPI and mixed with serial programs chosen from NPE-SER and SPEC CPU 2000. The process of each parallel application varies from 2 to 4. The detailed combinations of serial and parallel programs are: i) In the case of 8 processes, MG-Par and LU-Par are combined with applu, art, equake and vpr; ii) In the case of 12 processes, MG-Par and LU-Par are combined with applu, art, ammp, equake, galgel and vpr; iii) In the case of 16 processes, MG-Par and LU-Par are combined with BT, IS, applu, art, ammp, equake, galgel and vpr.

Table 3.6: The optimality of SVP-A*

Number of Jobs	Average Degradation (%)			
	Dual Core		Quad Core	
	O-SVP	SVP-A*	O-SVP	SVP-A*
8	0.12	0.12	0.34	0.34
12	0.22	0.22	0.36	0.36
16	0.13	0.13	0.27	0.27

Table 3.7: The optimality of SVPPC-A*

Number of Processes	Average Degradation (%)			
	Dual Core		Quad Core	
	O-SVPPC	SVPPC-A*	O-SVPPC	SVPPC-A*
8	0.07	0.07	0.098	0.098
12	0.05	0.05	0.074	0.74
16	0.12	0.12	0.15	0.15

As can be seen from Table 3.6 and 3.7, SVP-A* and SVPPC-A* achieve the same performance degradations as those by O-SVP and O-SVPPC, respectively. These results verify the optimality of the A*-search-based algorithms. Indeed, SVPPC-A* combines the functionalities of SVPPC and the A*-search algorithm and is expected to generate the optimal solution.

Table 3.8 and 3.9 show the scheduling efficiency of our A*-search-based algorithms under the two different strategies of setting the $h(v)$ function proposed in Section 3.8. SVP-A*-1 (or SVPPC-A*-1) and SVP-A*-2 (or SVPPC-A*-2) are the SVP-A* (or SVPPC-A*) algorithm that uses Strategy 1 and 2, respectively, to set $h(v)$. Table 3.8 shows the results for synthetic serial jobs, while Table 3.9 shows the results for parallel jobs. In Table 3.9, 4 synthetic parallel jobs are used and the number of processes of each parallel job increases from 10 to 50. Recall that the O-SVP algorithm is equivalent to SVP-A* with the $h(v)$ function being set to 0, while O-SVPPC is equivalent to SVPPC-A* with $h(v)$ being set to 0. Therefore, we also conducted the experiments to show the scheduling efficiency of O-SVP and O-SVPPC, which can be used to demonstrate the effectiveness of the strategies of setting $h(v)$. The underlying reason why SVPPC-A* and

SVP-A* could be effective is because they can further avoid the unnecessary searches in the constructed co-scheduling graph. Therefore, we also recorded the number of paths visited by each algorithm and present them in Table 3.8 and 3.9.

It can be seen from both tables that the strategies used to set $h(v)$ play a critical role in our A*-search-based algorithms. Both Strategy 1 and 2 proposed in Section 3.8 can reduce the number of visited paths dramatically and therefore reduce the solving time compared with the corresponding O-SVP and O-SVPPC. These results suggest that the strategies proposed in this chapter can greatly avoid the unnecessary searches.

Further, as observed from Table 3.8 and 3.9, the algorithms under Strategy 2 visited the less number of paths by orders of magnitude than their counterparts under Strategy 1. Therefore, SVP-A*-2 and SVPPC-A*-2 are more efficient by orders of magnitude than SVP-A*-1 and SVPPC-A*-1, respectively, in finding the optimal co-scheduling solution. This is because the estimation of $h(v)$ provided by Strategy 2 is much closer to the actual shortest path of the remaining nodes than that Strategy 1, and consequently Strategy 2 is much more effective than Strategy 1 in avoiding unnecessary searches.

The scalability of the proposed algorithms can also be observed from Table 3.8 and 3.9. It can be seen that SVPPC-A*-2 (or SVP-A*-2) show the best scalability against SVPPC-A*-1 and O-SVPPC (or SVP-A*-1 and O-SVP). This can be explained as follows. Although the scale of the constructed co-scheduling graph and the possible searching paths increase rapidly as the number of jobs/processes increases, SVPPC-A*-2 can effectively prune the graph branches that will not lead to the optimal solution. Therefore, the increase in the graph scale will not increase the solving time of SVPPC-A*-2 as much as for other two algorithms.

Table 3.8: Comparison of the strategies for setting $h(v)$ with serial jobs

Number of Jobs	Solving time (seconds)		
	SVP-A*-1	SVP-A*-2	O-SVP
16	0.72	0.014	1.01
20	12.88	0.047	17.52
24	190.79	0.14	234.5

Number of Jobs	The number of visited paths		
	SVP-A*-1	SVP-A*-1	O-SVP
16	31868	122	49559
20	546603	436	830853
24	6726131	1300	9601465

Table 3.9: Comparison of the strategies for setting $h(v)$ with parallel jobs

Number of Processes	Solving time (seconds)		
	SVPPC-A*-1	SVPPC-A*-2	O-SVPPC
40	0.43	0.037	0.61
80	2.44	0.17	3.38
120	10.93	0.33	17.93
160	40.05	0.64	66.85
200	99.13	0.88	212.79

Number of Processes	The number of visited paths		
	SVPPC-A*-1	SVPPC-A*-2	O-SVPPC
40	18481	414	27349
80	261329	1952	422025
120	1275799	4452	2105706
160	3990996	7050	6585938
200	8663580	16290	15991561

3.11.7 Heuristic A*-search algorithm

The experiments presented in this subsection aim to verify the effectiveness of HA*. We conducted the experiments to compare the solutions obtained by HA* and those by OA*. We also compared HA* with the heuristic algorithm (denoted by PG) developed in [61] for finding co-scheduling solutions. PG first calculates the politeness of each job based on the degradation that the job causes when it co-runs with other jobs, and then applies the greedy algorithm to co-schedule "polite" jobs with "impolite" jobs.

In the experiments, we choose 12 applications from NPB3.3-SER and SPEC CPU 2000 (BT, CG, EP, FT, IS, LU, MG, SP, UA, DC, art and ammp) and co-schedule them on Quad-core machines using OA*, HA* and PG. We also conducted the similar experiments on 8-core machines, in which 16 applications were used from NPB3.3-SER and SPEC CPU 2000 (BT, CG, EP, FT, IS, LU, MG, SP, UA, DC, art, ammp, applu, equake, galgel and vpr). The experimental results for Quad-core and 8-core machines are presented in Figure 3.14 and Figure 3.15, respectively. Note that the algorithms aim to optimize the average performance degradation of the batch of jobs, which is labelled by "AVG", not to optimize performance degradation of each individual job.

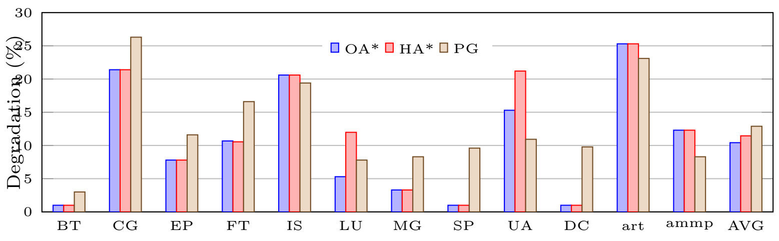


Figure 3.14: Comparing performance degradations of benchmarking applications on Quad-core machines under OA*, HA* and PG

In Figure 3.14 and 3.15, the average performance degradation obtained by HA* is worse than OA* only by 9.8% and 4.6% on Quad-core and 8-core machines, respectively, while HA* outperforms PG by 12.6% and 14.6% on Quad-core and 8-core machines, respectively. These results show the effectiveness

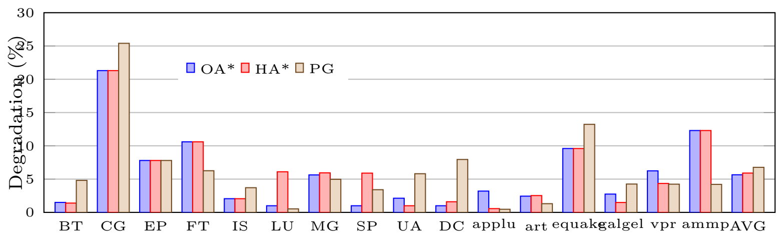
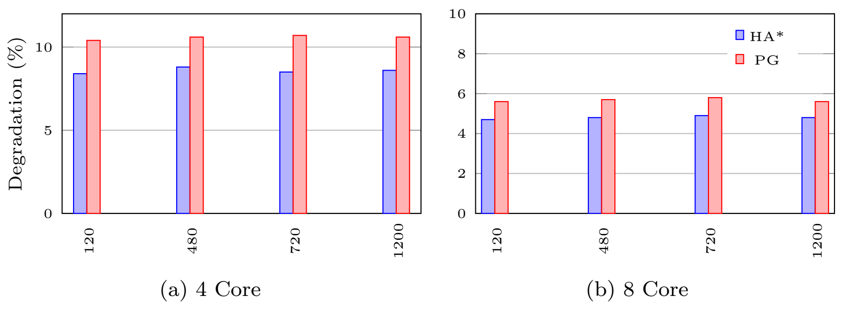


Figure 3.15: Comparing performance degradations of benchmarking applications on 8-core machines under OA*, HA* and PG

of the heuristic approach, i.e., using the MER function, in HA* and that the heuristic method can deliver the near-optimal performance.

We also used the synthetic jobs to conduct larger-scale experiments and compare HA* and PG. The synthetic jobs are generated in the same way as in Figure 3.5a and 3.5b. The results on Quad-core and 8-core machines are presented in Figure 3.16a and 3.16b, respectively. It can be seen from the tables that HA* outperforms PG in all cases, by 20%-25% on Quad-core machines and 16%-18% on 8-core machines).



(a) 4 Core (b) 8 Core

Figure 3.16: Comparing the degradation under HA* and PG algorithms

We further investigated the scalability of HA* in terms of the time spent in finding the co-scheduling solutions. Figure 3.17 shows the scalability for co-scheduling synthetic jobs on Quad-core and 8-core machines. The synthetic jobs in this figure are generated in the same way as in Figure 3.5a and 3.5b.

By comparing the scalability curve for Quad-core machines in Figure 3.17 with that in Figure 3.18b, it can be seen that HA* performs much more effi-

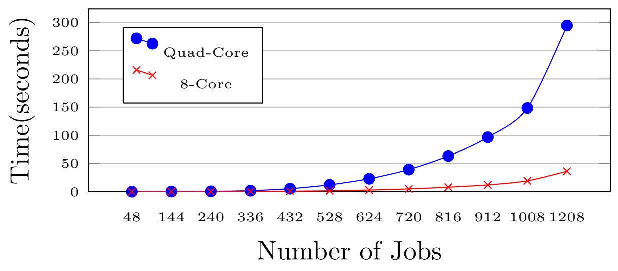


Figure 3.17: Scalability of HA* on Quad-core and 8-core machines

ciently than OA*. Another interesting observation from Figure 3.17 is that HA* spends much less time to find solutions on 8-core machines than on Quad-core machines. This is because we use the MER function, $\frac{n}{u}$, to trim the searching in HA*. When there are more cores in a machine (consequently, less machines are needed to run the same batch of jobs), less number of valid nodes will be examined in each level of the co-scheduling graph and therefore less time is taken by HA*. The scalability trend of OA* is different as shown in Figure 3.18a and 3.18b. In OA*, when the jobs are scheduled on machines with more cores, the co-scheduling graph becomes bigger with more graph nodes. OA* will examine all nodes in the graph in any case. Therefore, the solving time of OA* increases as the number of cores in a machine increases.

3.11.8 Efficiency of OA* and IP

This subsection investigates the efficiency of OA* and IP, i.e., the time spent by the methods in finding the optimal co-scheduling solutions. We used various IP solvers, CPLEX [3], CBC [39], SCIP [41] and GLPK [2], to solve the same IP model. The results are shown in Table 3.10. As can be observed from Table 3.10, CPLEX is the fastest IP solver. Note that when the number of processes is 16, the solving times by SCIP are all around 1000 seconds. This is only because the SCIP solver gave up the searching after 1000 seconds, deeming that it cannot find the final solution. It can be seen that the IP solvers are not efficient in solving the optimal co-scheduling problem. In fact, our records show that none of these IP solvers can manage to solve the IP model for more than 24 processes. We also used O-SVP to find the co-scheduling solutions in the experiments and present the solving times in Table 3.10.

It can be seen that OA* finds the optimal solution much more efficiently than O-SVP and that the time gap becomes increasingly bigger as the number of jobs increases. Note that we did not even use the process condensation technique in OA* for handling parallel processes due to the small problem scale. We also used synthetic jobs to conduct the comparison with more jobs. The experimental results are shown later in Table 3.8 when we compare the effect of different strategies for setting the $h(v)$ function in OA*.

The results in Table 3.10 (and in Table 3.8) show that OA* is much faster than IP. These results show the significant advantage of OA* over IP in terms of efficiency. The reason for this superiority is because 1) the layout of the co-scheduling graph is carefully designed so that it is fast to find the next valid node for node expansion and 2) the design of $h(v)$ can effectively avoid the searching space that cannot lead to the optimal solution.

Figure 3.18a and 3.18b show the scalability of OA* on Dual-core and Quad-core machines, respectively, as the number of serial processes increases.

Table 3.10: Efficiency of different methods on Quad-core machines

Number of Jobs	Solving time (seconds)					
	CPLEX	CBC	SCIP	GLPK	OA*	O-SVP
8(se)	0.086	0.19	0.28	0.049	0.004	0.004
8(pe)	0.33	0.26	0.21	0.041	0.005	0.006
8(pc)	0.48	0.45	0.24	0.038	0.006	0.07
12(se)	3.44	72.74	51.09	51.58	0.15	0.5
12(pe)	0.998	13.56	30.32	15.97	0.24	0.94
12(pc)	2.23	21.09	29.82	16.42	0.2	0.97
16(se)	33.4	704	1000	33042	0.63	1.26
16(pe)	32.52	303	1001	1231	1.52	2.89
16(pc)	11.76	313	1001	1170	1.63	2.93

3.11.9 The optimization techniques

This section tests the efficiency of the communication-aware process condensation techniques and the clustering approximation proposed in this chapter. The experiments are conducted on the Quad-core machines.

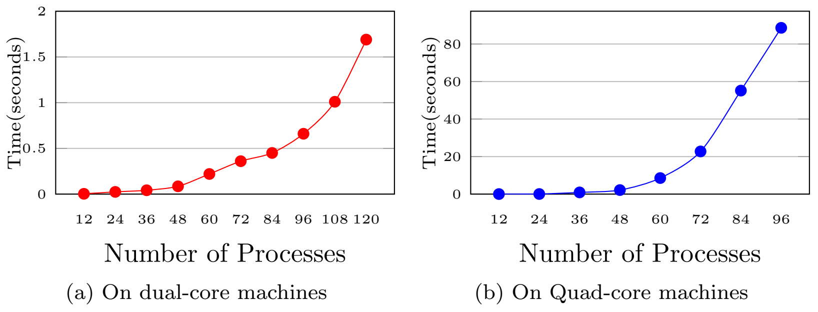


Figure 3.18: Scalability of OA*

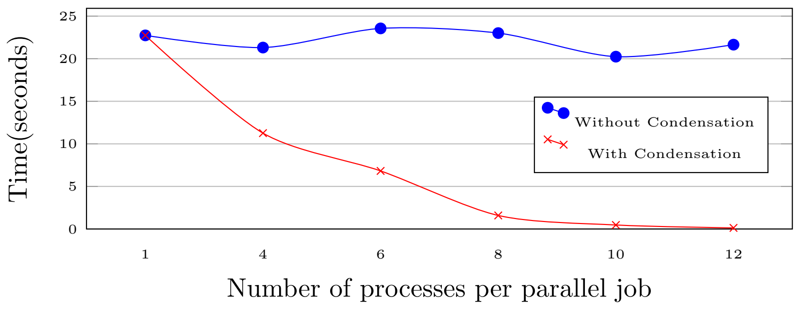


Figure 3.19: Solving time with and without process condensation as the number of processes per parallel job increases. The total number of parallel processes and serial jobs is 72.

We first test the effectiveness of the communication-aware process condensation technique. The experiments are conducted on the Quad-core machines with synthetic jobs. In this set of experiments, the number of total jobs/processes is 72, in which the number of parallel jobs is 6 with the number of processes per job increasing from 1 to 12 and the remaining jobs are serial jobs. These jobs are scheduled using SVPPC-A* with and without applying the process condensation. The solving times are plotted in Figure 3.19.

It can be seen from the Figure 3.19 that after applying the process condensation technique, the solving time decreases dramatically as the number of processes increase. This is because the number of nodes with the same communication pattern in the graph increases as the number of processes increases. Therefore, the condensation technique can eliminate more nodes from the co-scheduling graph and consequently reduce the solving time.

The clustering approximation algorithm is tested with 32 synthetic serial jobs. These jobs are first scheduled using O-SVP. Then these jobs are grouped into 8, 4 and 2 classes by setting the Similarity Level (SL). The experimental results are presented in Table 3.11. It can be observed from Table 3.11 that when the jobs are grouped into 8 classes, the degradation increases slightly, compared with that achieved by O-SVP. But the scheduling time under the clustering technique is reduced significantly. Moreover, as the number of class decreases, the degradation increases further and the scheduling time continue to decrease. These results show that our clustering technique is effective. This table also lists the number of the subpaths visited by the co-scheduling algorithms, which decreases by orders of magnitude as the number of class decreases. This is the underlying reason why the scheduling time decreases after applying the clustering approximation technique.

Table 3.11: Comparing the clustering method with O-SVP

Algorithm	visited path	Degradation (%)	time (seconds)
O-SVP	181267889	19.97	708
8 class	2115716	21.23	14.25
4 class	141508	23.75	1.18
2 class	17691	25.96	0.31

3.12 Summary

In this chapter, we explore the problem of finding the optimal co-scheduling solutions for a mix of serial and parallel jobs on multi-core processors. The co-scheduling problem is first modelled as an IP problem and then the existing IP solvers can be used to find the optimal co-scheduling solutions. Then we proposes a graph-based method to co-schedule jobs in multi-core processors. Then finding the optimal co-scheduling solution is modelled as finding the shortest valid path in the graph. A basic algorithm for finding the shortest valid path for serial jobs is first developed and then the optimization strategy is proposed to reduce the scheduling time. Further, the algorithm for serial jobs is extended to incorporate parallel jobs, so that it can find the optimal co-scheduling solution for a mix of serial and parallel jobs. The algorithm for parallel jobs is also optimized to reduce the scheduling time. An Optimal A*-search algorithm (OA*) is developed to find the shortest valid path efficiently. Based on OA*, we proposes a heuristic A*-search algorithm to find the near-optimal solutions with much less time. Moreover, a flexible approximation technique, called the clustering technique, is proposed to strike the balance between solving efficiency and solution quality. The experiments have been conducted to verify the effectiveness of the algorithms proposed in this chapter.

Chapter 4

WolfPath: Accelerating Iterative Graph Searching Algorithm on GPU

4.1 Introduction

A lot of research have shown that the graph theory can be used to solve scheduling problems [74] [28] [126] [25]. However, due to the NP-Hard nature of the scheduling problem, the size of graph increased exponentially, which leads to poor scalability. When processing large-scale graphs, parallel processing techniques are widely used. Many frameworks have been developed to process large graphs in parallel. According to hardware architecture, these frameworks can be classified into the following three classes: The distributed systems [93] [85] [43], the single machine systems [76] [140] [109] and GPU-accelerated systems [146] [65] [40].

Many of these graph processing frameworks employ iterative processing techniques. Namely, graph processing involves many iterations. Some iterative graph processing algorithms use the threshold value (e.g., in the PageRank algorithm) or the number of vertices/edges (e.g., in the Minimum-cut algorithm) to determine the termination of the algorithms. In these algorithms, the iteration count is known beforehand.

However, in iterative traversing-based graph processing algorithms (such as the connected component and the shortest path algorithms), the algorithm is driven by the graph structure and the termination of the algorithm is determined by the states of vertices/edges. Therefore, these algorithms need to check the state of vertices/edges at the end of every iteration to determine whether to run the next iteration. In each iteration of an iterative traversing-based graph processing algorithm, either synchronous (such as Bulk Synchronous Parallel used by Pregel [93]) or asynchronous (such as Parallel Sliding Window used by GraphChi [76]) methods can be used to compute and update the values of vertices or edges in the graph. The processing terminates when all vertices meet the termination criteria.

The aforementioned termination criteria are application-specific (e.g., the newly computed results of all vertices remain unchanged from the previous iteration). The number of iterations is unknown before the algorithm starts. Such a termination method has the following limitation in GPU-accelerated systems: In GPU-accelerated systems, all the threads in different thread blocks need to synchronize their decision at the end of each iteration. However, current GPU devices and frameworks (CUDA [27] and OpenCL [124]) do not support synchronization among different thread blocks during the execution of the kernel. Therefore, to synchronize between different thread blocks, the program has to exit the kernel, and copy the data back to the host and use the CPU to determine whether the computation process is complete. This frequent data exchange between host and GPU introduces considerable overhead.

To address this problem, we present WolfPath, a framework that is designed to improve the iterative traversing-based graph processing algorithms (such as BFS and SSSP) on GPU. Compare to other graph systems, WolfPath has the following features: First, a graph in WolfPath is represented by a tree structure.

In doing so, we manage to obtain very useful information, such as the graph diameter, the degree of each vertex and the traversal order of the graph, which will be used by WolfPath in iterative graph computations. Second, we design a layered graph structure, which is used by WolfPath to optimize GPU computations. More concretely, for all vertices in the same depth of the graph tree, we group all the edges that use these vertices as source vertex into a layer. All the edges in the same layer can be processed in parallel, and coalesced memory access can be guaranteed. Last but not least, based on the information we gain from the graph model, we design a computation model that does not require frequent data exchange between host and GPU.

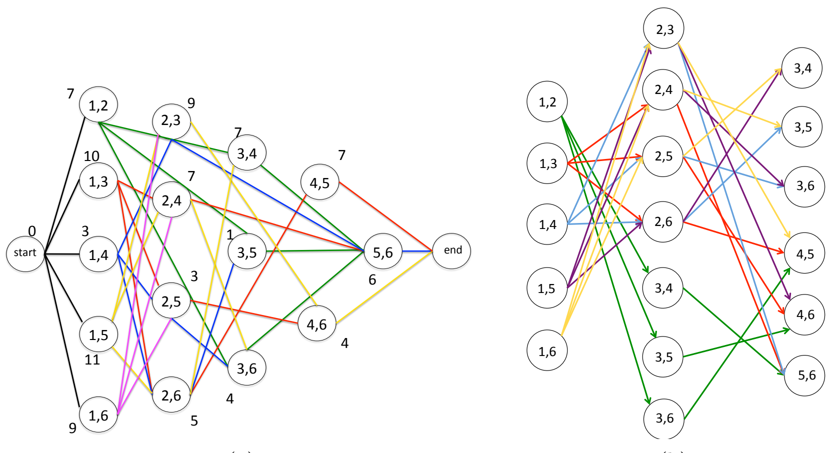
The rest of this Chapter is organised as follows. Section 4.2 presents the data structure used by WolfPath to store the co-scheduling graph in GPU memory. Section 4.3 presents the computation model used by WolfPath. Section 4.4 extends the WolfPath to a general graph processing framework. Experimental evaluation is presented in Section 4.5.

4.2 Representing Co-Scheduling Graph in GPU

Recall the co-scheduling graph we proposed in Chapter 3 (For convenience purpose, we list the co-scheduling graph in Figure 4.1a again). In the co-scheduling graph, the vertices are organised in a layered structure, and the edges are established as the algorithm progress. However, this structure is not suitable for GPU processing due to the following reason: Since there are no edges in the graph, when processing the co-scheduling graph in GPU, the only option is to assign one vertex to each thread, and then each thread iterates over the next valid level, finds all valid vertices and computes the results. However, this strategy not only limits the parallelism degree on GPU but also causes random access to global memory and thread divergence.

Taking co-scheduling graph in Figure 4.1a as an example, there are 5 vertices in the first level (the level with job 1), when processing these vertices in parallel with the strategy we discussed above, the maximum number of threads need to be launched is 5. However, during computation, each vertex will establish 3 edges, which gives 15 edges in total. If all these edges can be processed in parallel, we can improve parallelism degree by 3. In addition, consider the first level again. The thread that is processing vertex $\langle 1,2 \rangle$ will access the memory location start with vertex $\langle 3,4 \rangle$, and thread with vertex $\langle 1,3 \rangle$ will search from the location start with vertex $\langle 2,3 \rangle$. Because only vertices in the same level are stored contentiously in the memory, the coalesced memory access cannot be guaranteed. In addition, each thread needs to determine if a vertex is valid or not (Algorithm 3.1), therefore, the threads diverged after this step which could lead to bad performance [112].

To avoid these problems, we re-organise the co-scheduling graph in following ways: first, we add edges between vertices. Second, for vertices in the same level, we place their destination vertices in the same level as well. With these modifications, the re-organised co-scheduling graph of Figure 4.1a is shown in Figure 4.1b.

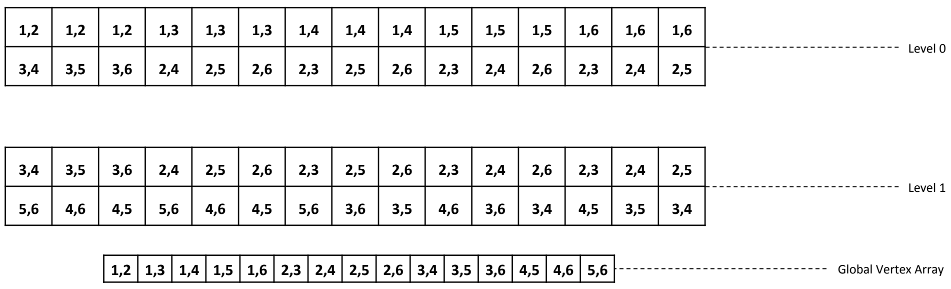


(a)

(b)

Figure 4.1: The example co-scheduling graph from Chapter 3 and after reorganisation

Organising co-scheduling graph in this way provides two advantages: First, because the graph edges are already established, it is possible to process all



1,2	1,2	1,2	1,3	1,3	1,3	1,4	1,4	1,4	1,5	1,5	1,5	1,6	1,6	1,6
3,4	3,5	3,6	2,4	2,5	2,6	2,3	2,5	2,6	2,3	2,4	2,6	2,3	2,4	2,5

3,4	3,5	3,6	2,4	2,5	2,6	2,3	2,5	2,6	2,3	2,4	2,6	2,3	2,4	2,5
5,6	4,6	4,5	5,6	4,6	4,5	5,6	3,6	3,5	4,6	3,6	3,4	4,5	3,5	3,4

1,2	1,3	1,4	1,5	1,6	2,3	2,4	2,5	2,6	3,4	3,5	3,6	4,5	4,6	5,6

Figure 4.2: An Example of layered edge list structure for graph in Figure 4.3a

edges at the same level in parallel. Hence, a better parallelism degree can be achieved. Second, with carefully designed graph data structure, it is possible to achieve coalesced memory access to GPU global memory.

Due to the space efficiency provided by CSR representation, most graphs algorithms/frameworks use this format to represent graphs in memory. However, as shown in [65], CSR will cause random access to graph data. Therefore, when using this format, the memory access to global memory is not coalesced, which will limit the performance. Therefore, it is important to design a data structure to store the graph in the GPU so that the coalesced global memory access can be guaranteed.

Based on the re-organised co-scheduling graph structure, we design a layered edge list structure to store the graph in the memory. In this design, each level in the co-scheduling graph is represented by two arrays, *source array* and *destination array*, which are used to store the source and destination vertexes of each edge in that level respectively. The i-th entry in the source array and the i-th entry in the destination array form an edge in the level. We also create a global vertex array to store the updated values of the destination vertices. An example is shown in Figure 4.2

It provides the following benefits to using this structure to store a graph in memory. First, the consecutive threads can read/write the consecutive elements from/to the source and destination arrays in the global memory. Therefore the coalesced global memory access can be guaranteed. Second, because all the edges in each layer are independent of each other, we can process them in parallel. In addition, if one layer is too big to fit in the GPU global memory, we can split it into multiple smaller edge lists. Third, multiple layers can be combined to fully utilise the computation power of GPU. We will discuss the last two advantages in more detail in a later section.

4.3 WolfPath Framework

In this section, we first discuss the limitation of current graph processing approach, and then we describe how WolfPath executes graph searching algorithm in parallel based on the co-scheduling graph structure. In the end, we demonstrate how to use WolfPath to implement the shortest path algorithm to find the optimal co-scheduling solution in the co-scheduling graph.

4.3.1 Motivation: the limitation of current approach

Many parallel graph processing frameworks are based on the iterative processing model. In this model, the computation goes through many iterations. In each iteration, either synchronous (such as Bulk Synchronous Parallel used by Pregel) or asynchronous (Parallel sliding window used by GraphChi) methods can be used to compute and update the vertex or edge values. The computation terminates when all vertices meet the application specific termination criterion (e.g., the results of all vertices remain unchanged).

In order to determine if all processes/threads meet the termination condition, the processes/threads need to communicate with each other. On CPU based parallel graph processing systems, the processes/threads can communicate through messaging passing or shared memory. On GPU, the threads are organised in thread blocks and the communication can be divided into two types: intra- and inter-block communication.

Intra-communication refers to the communications between the threads within a block, which is achieved via shared memory or global memory in GPU. On the contrary, the inter-block communication is the communications across different thread blocks. However, there is no explicit support for data communication across different thread blocks. Currently, inter-block data communication is realized through the GPU global memory followed by a barrier synchronization in CPU [27] [124]. The barrier is implemented by terminating the execution of the current kernel and re-launching the kernel.

The reason for the lack of support for inter-block communications on GPU is as follows. On GPU, the number of thread blocks launched by an application is normally much larger than the number of Streaming Multiprocessors (SM). However, When a large number of threads try to communicate between different blocks, it can cause the deadlock. An example is given as follows to illustrate the deadlock issue. Suppose that there are 5 thread blocks and only 4 SMs and that each thread block will occupy all resources on a SM. Assume blocks 1 to 4 execute on 4 SMs. When synchronization occurs, blocks 1-4 will wait until block 5 finishes. However, block 5 will never be executed on any SM since all SMs are busy and there are no resources available. Consequently, the deadlock occurs.

Due to the lack of support for inter-block communications, implementing iterative graph computation algorithm on GPU is much more challenging than on CPU. To demonstrate this, let us first consider how the iterative computation is implemented on CPU. Algorithm 4.1 shows the high level structure of the iterative computation on CPU. The loop is controlled by the *flag* variable, which is set to be true at the beginning. Next, all processes/threads execute a user-defined function, *compute()*, and then invoke the *update_condition* function to check if the user-specified termination condition is met. Each process/thread has its own $flag$ variable, which is updated by *update_condition* function. The *update_condition* function returns *false* if the program reaches the termination condition and returns *true* if otherwise. The $flag$ variable is synchronised between all the processes/threads. If its value is $false$, the iteration terminates.

Algorithm 4.1: Iterative Computation on CPU

```
  flag = true
  do
  {
    for all processes/threads in parallel:
    compute();
    flag = update_condition();

    synchronise flag between all process
  }while(flag == true)
```

Algorithm 4.2: Iterative processing kernel

```
update_kernel(bool flag){
  for all thread in parallel:
    d_flag = update_condition()
    synchronise d_flag with thread block;
    if d_flag for thread block i is false
      flag = false;
}
```

When executing the above code in parallel on a CPU-based system, the synchronization of the $flag$ variable can be easily achieved through shared memory or message passing. However, due to the fact that the current GPUs do not support synchronisation between different thread blocks, it takes more efforts to achieve the synchronization on GPU. The only solution that can ensure that the shared variable is properly synchronized across all thread blocks is to exit the kernel. To implement the iterative processing on GPU, many state-of-the-art graph processing frameworks use the following approach. The $flag$ variable is stored in the global memory of GPU. Each thread also has a local d_flag variable. If a thread meets the termination condition in the current iteration, it sets its own d_flag to *false*. Then d_flag is synchronised between all the threads within a thread block. One thread in each thread block updates the global $flag$ variable if the value of d_flag in this thread block is $false$. Next, the program exits the kernel and copies the value of $flag$ back to the host, which is used by the host program to determine whether another kernel should be launched (i.e., continuing to perform the computation). This technique is outlined Algorithms 4.2 and 4.3.

Algorithm 4.3: Iterative processing host

```
flag = true;
do{
  copy_flag_to GPU;
  update_kernel();
  copy_flag_to_CPU;
}while(flag==true)
```

Table 4.1: Real world graphs used in the experiments

GPU In memory Graph		
Graph	Vertices	Edges
RoadNet-CA [80]	1965206	5533214
amazon0601 [80]	403394	3387388
Web-Google [80]	875713	5105039
LiveJournal [80]	4847571	68993773

Clearly, in order to decide whether to launch the next iteration, the value of $flag$ needs to be exchanged between host and GPU frequently. These operations incur a significant overhead. If the number of iterations needed to complete the graph computation is known beforehand, the exchange of $flag$ between host and GPU can be eliminated, which can potentially improve the performance.

We conduct the experiments to investigate the extra overhead caused by exchanging the $flag$ variable. Four real world graphs, which are listed in Table 4.1, are used in this experiment. For each graph, we run the BFS algorithm implemented on the CuSha framework [27]. We record the computation time and the number of iterations it takes to complete the algorithm. Then, we determine the diameter of the graph using the BFS algorithm. We can prove that the number of iterations the iterative graph computation has to perform must be less than the graph diameter. Instead of using the $flag$ value to determine the termination condition of the graph computation, we use the graph diameter as the termination condition, i.e., we terminate the computation when the number of iterations equals to the graph diameter. We re-run the modified program and record the computation time. The results are listed in Table 4.2.

As can be seen from Table 4.2, the computation time of modified program is much faster than the original version. We also noticed that the original

Table 4.2: The computation time comparison of Original CuSha and Modified CuSha

Graph	Original		Modified	
	Time (ms)	Iteration	Time(ms)	Iteration
RoadNet-CA	195.97	551	1.98	554
amazon0601	22.3	35	0.15	35
web-Google	12.83	21	0.14	32
LiveJournal	70.96	9	0.076	14

Table 4.3: Memory to Computation

Graph	Memory copy (ms)	Computation
RoadNet-CA	0.36	0.004
amazon0601	0.63	0.004
Web-Google	0.6	0.004
LiveJournal	7.9	0.005

program takes a fewer number of iterations than the modified version in some cases. This is because the computation converges fast in those cases. Therefore, using the graph diameter as the termination condition in those cases causes the extra overhead of performing unnecessary computations. In order to compare these two types of overhead, we measured the average time for performing data copying and the average computation time per iteration in the experiments. The results are listed in Table 4.3:

The results from Table 4.3 show that the data copy time is greater than the computation time by 90 to 1500 times. These results indicate that it is worth performing the excessive iterations, comparing with copying data. We observe another interesting point in Table 4.3. No matter which graph the algorithm is processing, only one integer (i.e., the value of the $flag$ variable) is copied between GPU and host. However, the average memory copying time per iteration is different for different graphs. This is because the synchronisation cost between thread blocks is different for different graphs. More threads are involved in synchronisation, the longer the synchronization takes.

All these results support our proposed strategy. Namely, it can improve performance to use the maximum number of iterations as the termination condition so as to eliminate the need of data copying between GPU and CPU. However, a question arises from the strategy: how to know the number of iterations needed for different graphs before the graph processing algorithm starts? This question motivates us to design a new graph representation that helps determine the graph diameter and further develop the novel and GPU-friendly graph processing methods.

4.3.2 Computation model of WolfPath

The experimental results shown in thw last section suggest that using the number of iterations can improve the performance in graph processing. However, because graph processing algorithms are data driven, it is difficult to determine the number of iterations for different graph inputs before the program starts. Much research [93] [62] [78] [63] have been conducted to tackle this problem. The research shows that when processing graph algorithms iteratively, the upper bound of the number of iterations is the diameter of a graph, i.e., the number of the nodes on the path corresponding to the graph diameter [106].

Determine the graph diameter is challenging on general graphs. However, in the co-scheduling graph, the graph diameter is number of layers in the graph, which can be computed easily. Hence, we can explicitly define the iteration number of the algorithm.

The computation process of WolfPath is as follows. In WolfPath, the graph is processed layer by layer. For each layer, three operations are performed by GPU in parallel: the read, compute and write operations. For i-th level in the graph, the read operation reads the updated vertex value from the global vertex array. The compute operation acts on each edge and uses the data gathered from the read operation to compute the value for its edge/vertices. The write operation writes the updated value to the global vertex value array. So the updated values can be used in next iteration by the read operation. Hence, The computation model in WolfPath is synchronous and guarantees that all updates from a previous compute phase are seen only after the write operation is completed and before the next read operation starts. The whole process terminates when all the levels have been processed, that is, the number

Algorithm 4.4: Computation process of WolfPath

```
processing(graph G)
{
  i = 0;
  v_rt = DEFAULT_VALUE;
  while(i<G.level)
  {
    parallel for all vertices in level $i$:
      read vertex value from global memory
    parallel for all edges in level $i$:
      compute update value;
    parallel for all vertices in level $i$:
      write update value to global memory;
  }
}
```

of iterations is equal to the number of levels of the graph. This process is outlined in Algorithm 4.4

4.3.3 Finding Optimal Co-scheduling solution with WolfPath

The reason for us to design WolfPath is to accelerate the co-scheduling algorithm we proposed in the last Chapter. In this Section, we show how to use WolfPath to find the optimal co-scheduling solution.

In the last Chapter, finding the optimal co-scheduling solution has been modelled as finding the shortest path in the co-scheduling graph. Therefore, in this Section, we demonstrate how to implement the shortest path algorithm with WolfPath on re-organised co-scheduling graph. We assume that entire graph can fit into GPU memory. The host program is listed in Algorithm 4.5 and the kernel function is listed in Algorithm 4.6

In host function, the global vertex array $d[V]$ is first created to store the updated value of each vertex. The size of this array is equal to the number of vertices in the graph. At the beginning of the algorithm, the source vertex is set to 0, and the rest vertices are set to infinite (Line 3-4). In order to reduce data transfer time between GPU and host, we copy the entire graph data G and $d[V]$ to GPU (Line 5-6). Then the program enters the main loop (Line 8). In each

Algorithm 4.5: Host function of SSSP implementation with WolfPath

```
 1 SSSP_host(graph G, v_rt)
 2 {
 3   d[v] = INF;
 4   d[v_rt] = 0;
 5   cudaMemCpy(copy G to GPU);
 6   cudaMemCpy(copy d[V] to GPU);
 7   i = 0;
 8   while(i<G.level)
 9   {
10     SSSP_Kernel(G, i, v_rt, d[V]);
11   }
12   cudaMemCpy(copy d[V] to host)
13 }
```

Algorithm 4.6: Kernel function SSSP implementation with WolfPath

```
 1 SSSP_kernel(graph G, l, v_rt, d[V])
 2 {
 3   parallel for all edges e in G[l] do:
 4   {
 5     source = e.v;
 6     destination = e.u;
 7     weight = e.weight;
 8     update_value = d[source]+weight;
 9     atomicMin(d[u], update_value);
10   }
11 }
```

iteration, it invokes the kernel function, which does the actual computation. The number of iterations is guided by the level of the co-scheduling graph $G.level$. (Line 8-11). Once the computation is done, the result is copied back to the host.

The kernel function takes graph data G, the level number l, root vertex and global vertex array as input. In each kernel call, the kernel processes all edges in lth level in parallel. Each edge in level l is assigned to a thread. The thread first reads the source and destination vertex ID of edge e from global memory, then it fetches the edge weight (Line 5-7). The thread then uses *source* ID to read the source vertex value from the array $d[]$, and compute the update value for destination vertex (Line 8). At the end of the kernel, it uses $atomicMin$ function provide by CUDA to write the results back to the array $d[]$ (Line 9).

4.4 General Graph representation in WolfPath

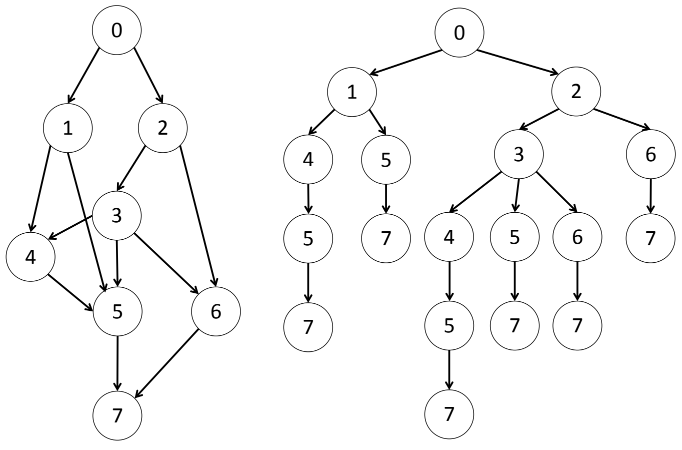
As discussed before, the layered structure of the co-scheduling graph can help to eliminate the time consuming memory operations in iterative graph processing. However, most real world graphs are stored in an unordered manner. In order to use WolfPath to process general graphs, these graphs need to be converted into a layered structure that is similar to the co-scheduling graph. In this section, we first describe how to convert general graphs into a co-scheduling graph like structure. Then we discuss how WolfPath handles the GPU under utilisation and how to partition the graph if it cannot fit in the GPU memory.

In WolfPath, we model the graph as a layered tree structure. That is, we first represent the graph as a tree-like structure, then group the vertices that in the same depth into one layer. By modelling the graph this way, the diameter of the graph is the distance from the root vertex to the deepest leaf vertex.

If some vertices in the same layer are connected, we duplicate these vertices in the next level. By duplicating these vertices, the vertex value updated in the current level can be sent to the next level. The reason for this design is as follow. The vertices in the current level and the next level form a group of edges. If a vertex is both source and destination of different edges, the calculated values of their neighbouring vertices may not settle (i.e., the calculated values are not final values of the vertices) after one iteration. Therefore, by duplicating these vertices in the next level, the updated value of their neighbouring vertices will be recomputed.

Based on the above description, given a graph $G = (V, E)$, a layered tree $T = (V_t, E_t)$ is defined as follows. $V_t \subseteq V$ and $E_t \subseteq E$. The root vertex of the tree, denoted by $v_{rt} \in V_t$, is the vertex which does not have in-edges. $degree_{in}(v)$ denotes the in-degree of vertex v. Then $degree_{in}(v_{rt}) = 0$.

$\forall v_t \in V_t$, if $degree_{in}(v_t)$ is greater than 0, then $\exists v \in V_t$ s.t. $(v, v_t) \in E_t$. If the out-degree of vertex v_t, denoted by $degree_{out}(v_t)$, is 0, then v_t is called a leaf vertex of T. Given a level L_i, $\forall v_t \in L_i$, if $\exists v \in L_i$ s.t. $e_t = (v_t, v)$, then v



(a) Example Graph (b) Layered Tree representation

Figure 4.3: An Example graph and its Layered tree representation

is also in the level L_{i+1}.

Figure 4.3b gives the tree structure of the graph shown in Figure 4.3a.

4.4.1 Preprocessing

In this work, a preprocessing program is developed to transform a general graph format into a layered tree. This program first reads the general graph format into the CPU memory and converts it into the CSR format. Next, it uses Algorithm 4.7 to build the layered tree, and then writes the layered tree back to a file stored in the disk. It is worth noting that this program is only run once for a graph. When processing a graph, WolfPath will first check if there exists a corresponding layered tree file. If the file exists, WolfPath will use it as the input. Otherwise, it will convert the graph into this new format. Algorithm 4.7 is used to build the layered tree.

Algorithm 4.7 is based on the breadth-first algorithm (BFS). The algorithm constructs a layered tree T for graph G (Line 3). It also creates Queue Q (Line 4). The algorithm starts with adding the vertex v_{rt} into Q (Line 5). In order to quickly retrieve the level information of each node, the algorithm also maintains an array called *node_level* (Line 6), which is used to store the level information of

Algorithm 4.7: Building layered tree

```
1  build_tree(graph G, v_rt)
2  {
3    Tree T = NULL;
4    Queue Q= NULL;
5    Q.enqueue(v_rt)
6    node_level[V] = {0};
7    while(Q != NULL)
8    {
9      v= Q.dequeue();
10     level = node_level[v];
11     for all neighbors $u$ of $v$:
12       if(u not in T)
13         T.add_node(v, u);
14         Q.enqueue(u);
15         node_level[u]+=1;
16       else if( u in T && u in $w \in T | w.level = level$)
17         T.add_node(v, u);
18         Q.enqueue(u);
19         node_level[u]+=1;
20   }
21 }
```

each node. The size of this array is equal to the number of vertices (denoted by V) in the graph G. This array is indexed by vertex id. The algorithm initialises all vertices in this array to 0 (Line 6). Then the algorithm performs the following steps iteratively (Line 7): it pops out the first vertex v from the queue (Line 9), and reads its level information from *node_level* (Line 10). $\forall e : v \to u \in E$ (Line 11), if $u \notin T$ (Line 12), or u has already been added into T but is in the same level as v (Line 16), the algorithm adds edge $\langle v, u \rangle$ in T (Line 13, 17). Next, the algorithm puts u in the queue by performing $enqueue(u)$ (Line 14, 18), sets the level of u to the current level plus 1 (Line 15 and 19). This process repeats until Q becomes empty.

4.4.2 Edge List Combination

By representing the graph in the layered tree format, we can gain the knowledge about how many layers there are in the graph and use it to determine the number of iterations needed for most graph algorithms. However, because WolfPath processes the graph layer by layer and the graphs have the nature of irregular data structure, such representation may cause the under-utilisation of GPU.

Algorithm 4.8: Processing Combinaed Edge list

```

processing(Tree T)
{
  CES = build_CES(T);
  i = 0;
  v_rt = DEFAULT_VALUE;
  while(i<CES.count)
  {
    levle = CES[i].level;
    while(j<level)
      parallel process all edges in CES[i]
  }
}
```

Each layer in the graph may have different numbers of edges. This number varies dramatically between levels. For instance, consider the graph shown in Figure 4.3. The first layer has 2 edges only. on the other hand, the second and third layer have 4 and 6 edges respectively. Hence, the number of threads required to process the first level is far less than the computing power (i.e., the number of processing cores) of GPU.

To overcome this problem, we propose the combined edge list, which is a large edge list that is constructed from multiple layers of edge lists. The combined edge list is constructed in the following way. We first define a number ME, which is the minimum amount of edges to be processed by GPU. Then we add the number of edges level by level starting from the first level of the layered tree. Once the total number of edges is greater or equal to ME, we group these levels together and then re-count the edges from the next level. This process repeats until all levels have been processed.

The way of building the combined edge list ensures that each combined edge list is formed by consecutive edge lists from the layered tree. Hence, each combined edge list can be treated as a sub-graph of the original graph. Therefore, the number of iterations required to process a combined edge list is equal to the number of levels used to form this tree. So Algorithm 4.4 is re-designed as Algorithm 4.8.

It is very important that we group the consecutive levels together to form a combined edge list. Because the result of the vertices in level i depends on those in level $i-1$, the results from the previous iteration need to be passed to the next iteration, which can be easily achieved by grouping the consecutive levels together. If we group the non-consecutive levels into a list, passing results between different levels requires lot of data transferring between host and GPU memory, which will harm the performance.

There remains one question: how do we choose the value for ME? If the number of ME is too small, the resulting edge list may not fully occupy the GPU. On the contrary, if it is too large, the size of the edge list may exceed the size of the GPU memory. Since it is desired that the GPU is fully occupied, the maximum active threads can be used to determine the minimum number of edges per combined edge list. The maximum number of active threads is the number of threads that a GPU can run simultaneously in theory, which can be found by Equation 4.1, where N_{sm} is the number of multiprocessors (SMs), MW_{psm} is the maximum number of resident warps per SM and the T_{pw} is the threads per warp.

$$N_{sm} * MW_{psm} * T_{pw} \tag{4.1}$$

4.4.3 Out of GPU memory processing

Comparing with the host platform, the GPU has a limited memory space. The size of real world graphs may be from few gigabytes to terabytes, which are too large to fit in the GPU global memory. Therefore, in this section, we develop an out-of-GPU-memory engine that can process such large scale graphs.

The general process of developing an out-of-GPU-memory engine is to first partition the graph into sub-graphs that can fit into GPU memory, and then process these sub-graphs in GPU one at a time. Therefore, our first objective in designing such an engine is to achieve good performance in graph partitioning. Ideally, the performance of this process should be as close as possible to the performance of loading the graph into memory. The second objective is that after partitioning the graph, the graph framework has to deal with the frequent data exchange between GPU and host memory. Otherwise, the performance will take hit.

Based on the design of Layered Edge List, these two goals can be achieved. The process of partitioning the graph is similar to building the combined edge list. We start with the first layer and accumulate the vertices in the layers until the size of accumulated vertices becomes larger than the GPU globa memory. We group all the accumulated vertices as a sub-graph and start the accumulating process again from the current layer. The partitioning process is complete when we have processed all layers in the graph.

The complexity of such partitioning method is $\mathrm{O}\big(N\big)$, where N is the number of layers in the graph. Given the fact that most real world graphs, especially the social network graphs, do not have a big diameter, the number of N will not be very large.

After partitioning the graph, each sub-graph is processed in order based on their positions in the graph. That is, the processing starts with the sub-graph that contains the first layer. The next sub-graph to be processed is the one that follows the first sub-graph. In addition, the updated values of the vertices in the last layer of the current sub-graph need to be passed to the next sub-graph. This can be achieved by retaining the updated vertex values in the global memory after the computation is finished. Therefore, when next sub-graph is loaded into the GPU global memory, the data needed by the sub-graph is in the global memory. To process each sub-graph, we use the same method as that for in-GPU-memory processing. Combining multiple layers into a sub-graph enables us to fully utilise the GPU.

It is possible that the size of one layer is larger than the GPU memory. in this case, we can split this layer into multiple parts and compute one part at a time. This works because all the edges in a layer are independent to each other and it is therefore safe to partition a layer into multiple small chunks and process them separately.

Table 4.4: Co-scheduling graphs used in the experiments

GPU In memory Graph		
Graph	Vertices	Edges
24	10626	2677118
36	58905	64238082
48	194580	564309075

4.5 Experimental Evaluation

In this section, we first compare the performance of the shortest path algorithm implemented using WolfPath with A* algorithm we developed in last Chapter. Then we evaluate the performance of WolfPath with general graphs. The experiments were conducted on a system with a Nvidia GeForce GTX 780Ti graphic card, which has 12 SMX multiprocessors and 3GB GDDR5 RAM. On the host side, we use the Intel Core i5-3570 CPU operating at 3.4 GHZ with 32 GB DDR3 RAM. The WolfPath were implemented using CUDA 6.5 on Fedora 21. All the programs were compiled with the highest optimisation level flag (-O3).

4.5.1 Performance comparison with CPU based A* algorithm

In this section, we evaluate the performance of the shortest path algorithm implemented in Section 4.3.3. In this experiment, we use this algorithm to find the optimal co-scheduling solution in the co-scheduling graph generated from synthetic jobs. The co-scheduling graphs are generated from 24, 36, 48 jobs on a quad-core machine and re-organised into the format we proposed in this chapter. The graph size is listed in Table 4.4. We compare the execution time of WolfPath's implementation with the A*-search algorithm proposed in the last chapter.

Figure 4.4 shows the execution time of both algorithms. In these experiments, we record both data transfer and computation time of WolfPath. As shown in this Figure, though WolfPath suffers from the unavoidable data transfer overhead, it still much faster than the A*-search algorithm. In this exper-

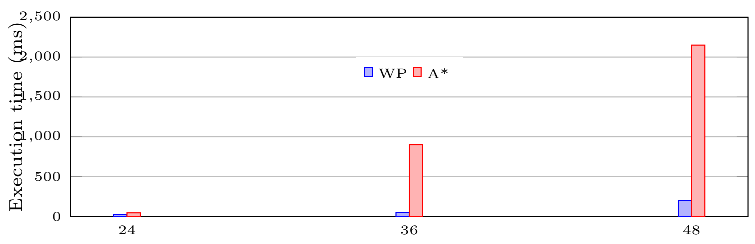


Figure 4.4: Execution time comparison between WolfPath and A*-Search

iments, WolfPath achieves nearly 20X speedup over A* on 36 jobs and 10X speedup on 48 jobs. These performance gain achieved by WolfPath is due to the massive parallelism degree provided by the GPU.

4.5.2 Performance evaluation with general graphs

We evaluate the performance of WolfPath on general graphs using two types of graph dataset: small sized graphs that can fit into the GPU global memory (called in-memory graphs in the experiments) and large scale graphs that do not fit (out-of-memory graphs). The size of a graph is defined as the amount of memory required to store the edges, vertices, and edge/vertices values in user-defined datatypes.

Small graphs are used to evaluate WolfPath's performance in in-memory graph processing against other state-of-the-art in-memory graph processing systems (e.g., CuSha [65] and Virtual-Warp-Centric [56], and the large graphs are used to compare WolfPath with other out-of-core frameworks that can process large graphs on a single PC (e.g., GraphChi and X-Stream).

The eight graphs listed in Table 4.1 and Table 4.5 are publicly available. They cover a broad range of sizes and sparsity and come from different real-world origins. For example, *Live-Journal* is directed social networks, which represent friendship among the users. *RoadNetCA* is the California road network, in which the edges represent roads and the vertices represent the intersections. *WebGoogle* is a graph released by Google, in which vertices represent web pages and the directed edges are links. *orkut* is an undirected social network, in which

Table 4.5: Real world graphs used in the experiments

GPU Out-of-memory Graph		
Graph	Vertices	Edges
orkut [80]	3072441	117185083
hollywood2011 [13]	2180653	228985632
arabic2005 [14]	22743892	639999458
uk2002 [13]	18520486	298113762

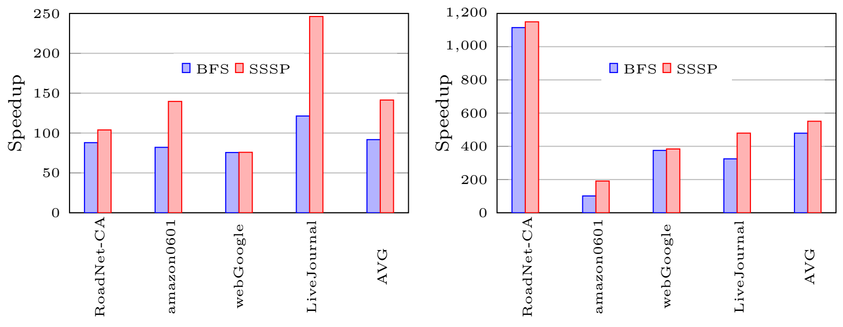
vertices and edges represent the friendship between users. *uk-2002* is a large crawl of the .uk domains, in which vertices are the pages and edges are links.

We choose two widely used searching algorithms to evaluate the performance, namely Breadth First Search (BFS) and Single Source Shortest Paths (SSSP).

Comparison with existing In-memory Graph Processing Frameworks

In this section, we compare WolfPath with the state-of-the-art in-memory processing solutions such as CuSha [65] and Virtual Warp Centric [56]. In the experiments, we use the CuSha-CW method, because this strategy provides the best performance in all CuSha strategies. Both CuSha and Virtual Warp Centric apply multi-level optimisations to the in-memory workloads.

We first compare the computation times among WolfPath, CuSha and VWC. Figure 4.5a and Figure 4.5b show the speedup of WolfPath over CuSha and VWC.



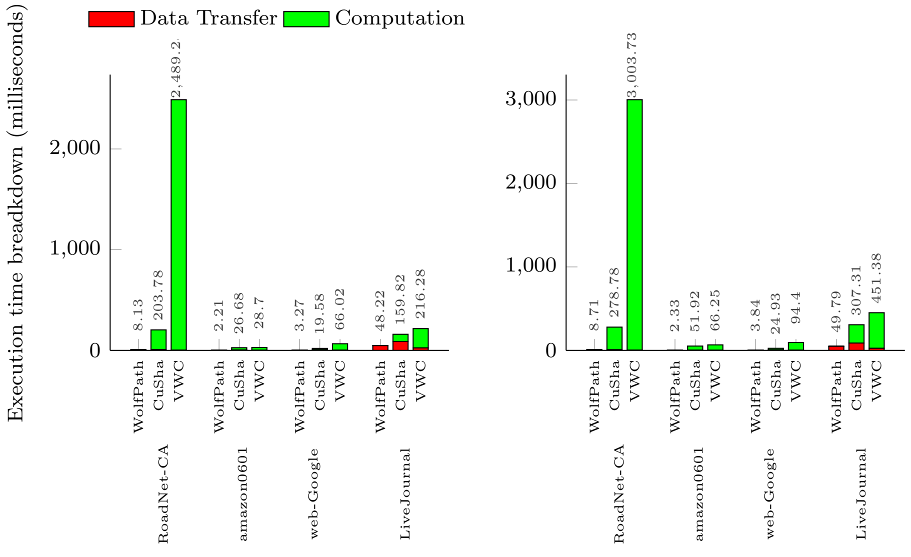
(a) Speedup over CuSha (b) Speedup over VWC

Figure 4.5: Speedup of WolfPath over CuSha and VWC

We also list the break down performances in Figure 4.6. In these experiments, the Data Transfer is time taken to move data from host to GPU; the computation refers to the time taken for actual execution of the algorithm. The number above each bar indicates the total execution time (data transfer + computation).

As can be seen from these results, WolfPath outperforms CuSha and Virtual Warp Centric. The average speedup of WolfPath over CuSha is more than 100X, and 400X over VWC. TThis is due to the elimination of memory copy operations. On the other hand, the other two systems rely on frequent data exchange between GPU and host to process the algorithms. These experiments also suggest that when processing graph algorithms with GPU, the execution is bounded by memory operations rather than computation. Therefore, reducing the memory operations can improve the performance significantly.

Also, the performance of VWC is the worst among 3 systems, because VWC does not guarantee the coalesced memory access. On the other hand, with carefully designed data structures, both WolfPath and CuSha can access graph edge in a sequential manner, hence memory performance is much better.



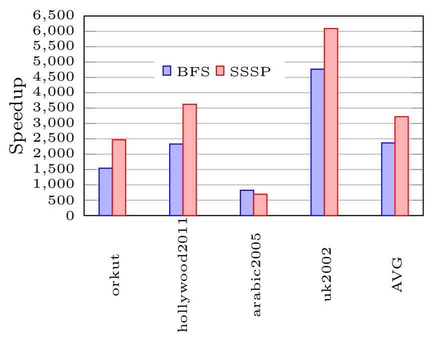
(a) BFS (b) SSSP

Figure 4.6: Execution time breakdown of WolfPath, CuSha and VWC on different algorithms and graphs. Reported times are in milliseconds.

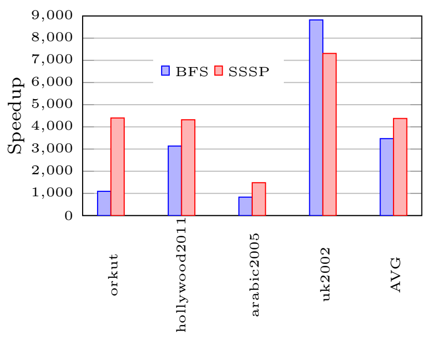
Comparison with GPU Out-of-memory Frameworks

The results shown in the last section demonstrate WolfPath's performance in processing in-memory graphs. However, many real-world graphs are too large to fit in GPU memory. In this section, we examine the WolfPath's ability to process large graphs which cannot fit into GPU memory. To the best of our knowledge, the state-of-the-art GPU-based graph processing frameworks [65] [146] [40] assume that the input graphs can fit in the GPU memory. Therefore, in this work, we compare WolfPath (WP) with two CPU-based, out-of-memory graph processing framework: GraphChi (GC) [76] and X-Stream (XS)[109]. To avoid disk I/O overhead in systems such as GraphChi and X-Stream, the dataset selected in the experiments can fit in host memory but not in GPU memory.

As shown in Figure 4.7a and 4.7b, WolfPath achieves an average speedup of 3000X and 4000X over GraphChi and X-Stream (running with 4 threads), respectively, despite its need to move the data between GPU and CPU. We also list the computation time and iteration counts of three systems in Table 4.6. Since X-Stream does not require any pre-processing and the computation is overlapped with I/O operations, we use the total execution time of the system as the comparison.



(a) Speedup over GraphChi



(b) Speedup over X-Stream

Figure 4.7: Speedup of WolfPath over GraphChi and X-Stream

As can be seen from the Table 4.6, although WolfPath performs more iterations than GraphChi and X-stream, it still outperforms them. This performance improvement is due to the massive parallel processing power provided by GPU, while GraphChi and X-Stream are CPU-based and their degrees of parallelism are limited.

Table 4.6: Execution time WolfPath, GraphChi and X-Stream on different algorithms and graphs. Reported times are in seconds.

BFS	Computation			Iteration		
	WolfPath	GraphChi	X-Stream	WolfPath	GraphChi	X-Stream
orkut	0.02	30.88	21.88	7	2	2
hollywood2011	0.09	209.86	282.17	16	8	13
arabic2005	0.2	164.92	166.18	51	3	3
uk2002	0.15	715.38	1323.59	49	28	48

SSSP	Computation			Iteration		
	WolfPath	GraphChi	X-Stream	WolfPath	GraphChi	X-Stream
orkut	0.02	49.38	88.02	7	3	3
hollywood2011	0.1	362.56	432.24	16	9	16
arabic2005	0.23	160.56	340.94	51	3	7
uk2002	0.16	974.83	1170.32	49	31	42

4.5.3 Memory occupied by different graph representation

From Figure 4.6, we can see that VWC has the shortest data transfer time. This is because it represents the graph in the CSR format, which is memory efficient. However, in order to have sequential access to the edges, both WolfPath and CuSha represent graphs with edges, which consume more memory space than CSR. In this section, we evaluate the cost of using the Layered Edge list representation in terms of required memory space against CSR and CuSha's CW representation.

Figure 4.8 shows the memory consumed by WolfPath's Layered Edge List, CuSha-CW and CSR. The Layered Edge List and CuSha-CW need 1.37x and 2.81x more space on average than CSR. CuSha uses 2.05x more memory than WolfPath, because it represents each edge with 4 arrays.

4.5.4 Pre-processing time

Table 4.7 shows the preprocessing time of WolfPath, CuSha, and GraphChi. The preprocessing time refers to the time taken to convert the graph from the raw data to the framework specified format (e.g., the layered tree in WolfPath

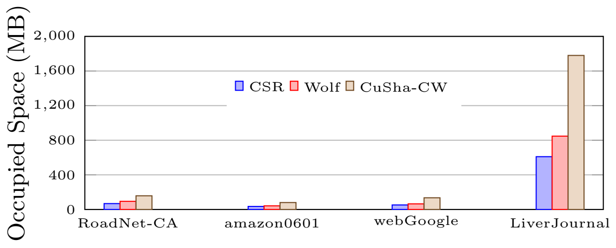


Figure 4.8: Memory occupied by each graph using CSR, CuSha-CW, WolfPath

or Shard in GraphChi). It consists of the graph traversing time and the time to write the data into the storage. Because CuSha is unable to process the graph larger than GPU memory, the corresponding cells in the table are marked as NA.

The first observation from the table is that 1) for in-memory graphs, CuSha preprocesses the data faster than other two systems, and 2) WolfPath is the slowest system. This is because CuSha does not write the processed data back to the disk. GraphChi will only write a copy of data into a shard. In contrast, WolfPath traverses the data using the BFS-based algorithm and then writes the data into a temporary buffer before it writes the data to the hard disk. Therefore, the workload of WolfPath is heavier than two other systems.

For graphs larger than GPU memory, WolfPath performs better than GraphChi when processing uk2002 and arabic2005. This is because GraphChi generates many shard files for these two graphs, and hence it takes longer to write to the disk.

From this experiment, we argue that in WolfPath, although the preprocessing is time-consuming, the pre-processing is worthwhile because of the following reasons: First, for each graph, WolfPath only needs to convert it once. Second, the resultant format provides better locality and performance for iterative graph computations.

Table 4.7: Preprocessing Time (Seconds)

	WolfPath	CuSha	GraphChi
Amazon0601	0.79	0.24	0.58
LiverJournal	14.68	3.8	10.15
WebGoogle	1.46	0.44	0.79
orkut	21.4	NA	22.07
hollywood	38.6	NA	34
uk2002	59.78	NA	69
arabic2005	120.3	NA	151.5

4.6 Summary

In this chapter, we develop WolfPath, which is a GPU-based graph processing framework designed to process iterative traversing-based graph processing algorithms efficiently. In ordered to accomplish the GPU memory access pattern, we reorganised the co-scheduling graph to make it suitable for GPU processing. By benchmarking the state-of-the-art GPU-based graph processing systems, we observed that in iterative graph processing systems, the operation that consumes most time is the data movement between GPU and host memory. Therefore, a new data structure called Layered Edge List is introduced to represent the graph. With this structure, the graph diameter is known before the graph processing starts, which is used to guide the iterative process and hence eliminates the frequent data exchange between host and GPU. We also propose a graph preprocessing algorithm that can convert an arbitrary graph into the layered structure. The experimental results show that WolfPath achieves the significant speedup over the state-of-the-art in-GPU-memory and out-of-memory graph processing frameworks.

Chapter 5

WolfGraph: an Edge-Centric graph processing framework on GPUs

5.1 Introduction

In the last chapter, we proposed WolfPath graph processing framework. To achieve high performance, WolfPath first converts the input graph into a layered graph, and then use this layered graph as input. However, this conversion consumes a significant amount of time, and this is because it requires running an algorithm that is based on BFS algorithm. Similar to WolfPath's approach, in most existing works, the graph is pre-processed before the graph processing algorithm is applied. The idea is that although it takes the time to pre-process a graph, the execution of the graph processing algorithm will take much less time than without pre-processing and therefore the overall execution time will be reduced significantly. However, this approach has following drawbacks. First, as pointed out by Guo et al. [46], in the state-of-the-art GPU-based graph processing systems, the time spent in reading a graph from hard disk to CPU memory and constructing the necessary data structure in memory for processing the graph, which is called the pre-processing time, constitutes a big proportion of the total processing time for a large graph. Reducing this pre-processing time will significantly improve the overall performance of graph processing frameworks. Second, existing work of GPU-based graph processing assumes the entire graph can fit into the global memory of GPU. However, some large-scale graphs are even bigger than GPU memory, which makes these work infeasible to process those graphs.

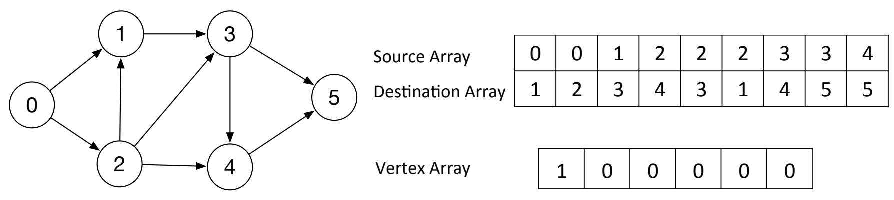
In this chapter, we present WolfGraph, a framework for processing large graphs on GPUs, to address the above problems. WolfGraph adapts a recently introduced graph processing model, known as edge-centric processing [109] to efficiently process the graphs on GPUs. In this model, the graph is represented as an unordered list of edges. The processing iterates over edges rather than vertices. More specifically, after the graph is split into edge blocks, each edge block contains an unordered list of edges that are contiguously stored in memory and a set of vertices associated with the edges of this block. Each edge block is processed by a thread block in GPU, and multiple thread blocks are processed by edge blocks in parallel. Within each thread block, the edges are processed in parallel by the threads. Such allocation of edge blocks to thread blocks enables the coalesced memory access to the edges in GPU global memory. In WolfGraph, the access to vertices is still random. However, the data structure for holding vertices is placed in the shared memory of GPU, the access to which is much faster than to the global memory. Moving the data structure of vertices to the shared memory also significantly reduces the synchronization overhead among threads during the graph processing.

The above processing handles the graph that can fit into the GPU global memory, which is called "in-memory" graph processing. For a graph larger than the global memory of GPU, we develop the "out-of-memory" processing. We first partition the graph into sub-graphs with minimal efforts. Each sub-graph can fit into the global memory. Each sub-graph is processed in the edge-centric manner by GPU. The advantage of such method is its minimal pre-processing time. However, it is at the expense of potential frequent exchange of sub-graphs between GPU global memory and host memory. To address this problem, we develop a new method called the Concatenate Edge List (CEL), which is inspired by GraphChi. When the CEL method loads a sub-graph into GPU global memory, for each vertex that is the destination vertex in this sub-graph, it also loads into GPU the edges that have this vertex as a source. With the CEL method, we only need load each sub-graph into GPU once.

The rest of this chapter is organised as follows. Section 5.2 presents the edge-centric processing model on GPU for the graphs that can fit into GPU memory. Section 5.3 presents the details of the WolfGraph framework including how to split the graph into edge blocks, the allocation of edge blocks to thread blocks and the core APIs provided by WolfGraph. Section 5.4 presents the graph partition and CEL methods for the graphs larger than GPU memory. Section 5.5 describes how to process graph that is larger than host memory. Experimental results are presented and analyzed in Section 5.7.

5.2 An Overview of Edge Centric Processing on GPU

It has been shown that because it allows sequential access to the edges, the edge-centric approach can improve I/O performance for disk-based graph processing, which requires frequent disk accessing during the execution [109]. Similarly, the sequential access to the GPU memory is critical to the performance of GPU applications [65]. This is because the sequential access guarantees the coalesced access to the global memory on the GPU device [65]. This section first gives an overview of edge centric graph processing on GPU and introduces the data



(a) Graph structure

Source Array	0	0	1	2	2	2	3	3	4
Destination Array	1	2	3	4	3	1	4	5	5

Vertex Array	1	0	0	0	0	0

(b) Edge centric representation

Figure 5.1: An exemplar graph and its edge centric representation

structure used to represent the graph, and then further presents the computation model used in WolfGraph.

5.2.1 Edge centric Graph data structure

The input data of the edge centric processing is an unordered set of directed edges (an edge in undirected graphs can be represented by a pair of directed edges, one in each direction). A graph is represented in memory by an edge list, which consists of three one-dimensional arrays: the *vertex array*, the *source array* and the *destination array.* The *vertex array* is used to store the state of each vertex. This array is indexed by the vertex ID, that is, ith entry of the vertex array contains the state of the vertex with ID i (e.g., the distance of the path connecting to vertex i). The *source array* and the *destination array* are used to store the source and destination vertices of each edge respectively. The ith entries in the source array and the destination array form an edge in the graph. In addition, if it is needed, an array of edge weights can be added to this graph representation structure. Figure 5.1 illustrates the edge-centric representation of a sample graph, in which there are 6 vertices and 9 edges.

Unlike other data structure for graphs, the edge-centric graph representation does not require any pre-processing of the data because of the way the edge list is constructed in the memory. When constructing the edge list, an edge is read from the raw data in the disk each time. The edge is stored in the same order in the edge list as it is in the raw data. Therefore, the raw data is only read sequentially once to construct the edge list in the memory. Consequently, the time spent in constructing the graph data structure in memory is minimized.

5.2.2 Computation model

The read-compute-write iterative processing model has been used in literature to process graphs. The advantage of this model is that the graph edges can be processed in any order without affecting the correctness of the final result. Therefore, the graph processing can be parallelized.

The read-compute-write model works in the following way. The model runs a loop of iterations to update the vertex/edge values until none of the vertex/edge values in the graph changes in an iteration. Each iteration consists of three phases: read, compute and write. The read phase first gathers the source and destination vertex for each edge (stored in the source and destination arrays in the edge list), and then uses the IDs of the fetched vertices to obtain the corresponding vertex values from the vertex array. The compute phase uses the data gathered in the read phase to update the values of corresponding edges/vertices. The write phase writes the updated values back to the vertex array so that the updated values can be used in next iteration.

In each iteration of the read-compute-write model, all edges in the graph need to be processed and the edge/vertex values are updated. The computations of the edges are unordered, i.e., independent of each other. Therefore, the edge computations in each iteration can be performed in parallel. It is straightforward to evenly distribute the workload across threads in such an unordered model. Note that on the contrary, the vertex-centric model, which visits the graph by vertex and represents the graph by the adjacency list, is inherently difficult to be load-balanced among threads, because each vertex is connected to a different number of edges.

5.3 In-memory processing Engine in WolfGraph

The in-memory engine is designed for processing the graphs which can be fitted in the global memory of GPU. When designing the in-memory engine, the key is to achieve a good degree of parallelism. Therefore, in this section, we first describe how to map the workload to the GPU threads and parallelize the computation process, and then discuss how to exploit the memory hierarchy of GPU to improve the performance. We also present the APIs provided by Wolf Graph and demonstrate how to program with these APIs at the end of the section. The in-memory processing engine will serve as the core component for processing the graphs whose size are bigger than the global memory of GPU, which we call out-of-memory graph processing and will be discussed in Section 5.4.

5.3.1 Parallel processing in WolfGraph

As discussed in the previous section, WolfGraph is based on the read-compute-write iterative processing model, and the edge computations in each iteration can be processed in parallel on GPU. To facilitate efficient graph processing, it is crucial to develop a suitable strategy to allocate the workload to GPU threads.

In WolfGraph, an edge is allocated to a thread and the continuously indexed threads process the edges that are stored in the contiguous memory space. This way, the coalesced memory access to the edges can be achieved in GPU. Once a thread completes the computation in an iteration (i.e., updates the value of a vertex), it writes the updated vertex values to the corresponding locations in the vertex array.

We identify two problems that need to be addressed in the write phase. First, since a single vertex array is shared by all GPU threads, multiple threads may write to the same memory location during the write phase. To address this problem. WolfGraph uses the CUDA *atomic* operation to synchronize potential simultaneous writes by threads.

Second, the vertex array is constructed in the CPU memory and copied to the GPU global memory at the beginning of the graph processing. When the threads write the newly computed data to the corresponding locations in the vertex array in each iteration, these locations may not be contiguous, which causes the random access to the vertex array in global memory. Our benchmarking experiments show that this is a factor that impairs the performance. To mitigate the performance degradation caused by the random access to global memory, WolfGraph does not write the newly updated data directly to the global memory, but write them (and synchronize, if necessary) to the shared memory first and then launch a separate kernel to write the new data to the global memory. It has two benefits by doing so. First, the number of random accesses to the global memory is significantly reduced. Although the writes to the shared memory is still random access, they are much faster than the random access to the global memory. Therefore, the overall performance is significantly improved compared with writing the new data directly to the global memory. This benefit is analyzed in detail later in this section and is also supported by our experiments in the later part of this chapter. Second, by writing the new data first to the shared memory, some of the necessary data synchronizations are moved from the global memory to the shared memory. We have conducted the benchmarking experiments about this. We made two observations from the experimental results: 1) synchronization in shared memory is faster than synchronization in global memory *when the degree of synchronization (i.e., the number of threads that write the data simultaneously to the same location in memory) is less than a threshold*; 2) caching the new data in the shared memory can reduce both the degree of synchronization and the number of synchronizations in global memory.

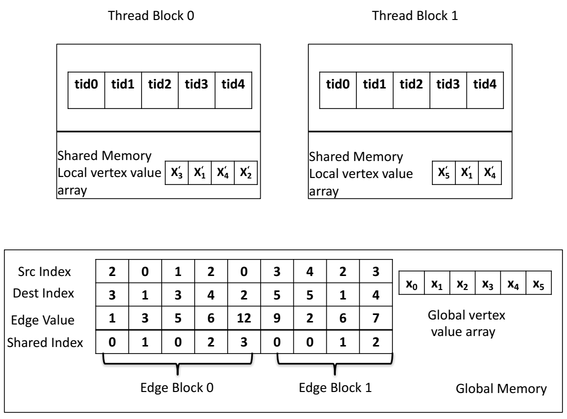
In memory data structure of WolfGraph

Our research shows that the graph representation presented in Section 5.2.1 does not exploit the GPU memory hierarchy effectively. In this section, we present the extension to the data structure and also discuss the memory .

One aim of this work is to minimize the pre-processing time of a graph. We have discussed in Section 5.2.1 that the time spent in constructing the edge list is minimized. We also discussed that the read-compute-wirte iterative model enables us to process the edges in an unordered way, i.e., the processing of the edges can be parallelized.

Once the size of the edge block (i.e., the number of edges in an edge block) is known, the edge list can be split with minimal effort. We will present the method of determining the edge block size later in this chapter.

In the hard disk, the graph is stored as a list of edges. When WolfGraph reads the graph from the hard disk, it constructs an edge list (containing three arrays: src index, dest index and edge value arrays) and a vertex-value array in the CPU memory as shown in Section 5.1 (Figure 5.1), which will be copied from CPU memory to global memory of GPU. In a GPU, WolfGraph splits the edge list into smaller *edge blocks* (We will introduce the method of determining the size of edge blocks later in this chapter), each of which consists of a set of edges. The threads that are processing the graph are organized in thread blocks. A thread block processes an edge block. Multiple thread blocks process the edge blocks in parallel. Within a thread block, an edge is processed by a thread in parallel. When a thread in a thread block processes an edge, it obtains its global thread index (assuming the thread index is i) in GPU and reads from the i-th element in the edge-value array to obtain the edge value. Then the thread obtains the index of the source node of the edge by reading the i-th element of the src index array and reads the value of the source node from the vertex-value array using the source index. After applying the user-defined graph processing algorithm (e.g., shortest path algorithm), it will generate an updated value for the destination node of the edge. In order to achieve the benefits discussed at the beginning of section 5.3 (i.e., reducing the number of random access to GPU global memory and also reducing the number of and the degree of data synchronization in global memory), the thread writes the updated value of the destination node first to the shared memory. In order to facilitate this, WolfGraph adds two new data structures: a *shared-index* array for the whole graph and a *local-vertex-value* array for each thread block. The local-vertex-value array is stored in local memory and used to hold the updated values of the destination nodes after processing the edges, while the shared-index array is in global memory and used to indicate to the thread which position the



Src Index	2	0	1	2	0	3	4	2	3
Dest Index	3	1	3	4	2	5	5	1	4
Edge Value	1	3	5	6	12	9	2	6	7
Shared Index	0	1	0	2	3	0	0	1	2

Figure 5.2: The Edge block representation of graph in Figure 5.1a

updated value of the corresponding destination node should be written into in the local-vertex array. As an example, the data structure and memory allocation for the graph in Figure 5.1 are shown in Figure 5.2. In the figure, the graph is divided into two edge blocks with 5 edges in block 0 and 4 edges in block 1, which are processed by thread block 0 and thread block 1, respectively (tidx in the thread blocks represents the thread id). The first element of the Share-index array is 0, which means that the updated value of the destination node of the edge $< 2, 3 >$ (the updated value is denoted by x'_3) should be written in the location with the index of 0 in the local vertex array (i.e., the first element of the array) in the shared memory.

Analysis of thread synchronization

As discussed above, our design requires the thread synchronisation in global memory. The thread synchronisation is implemented by the atomic operations. When designing applications on GPU, shared memory has been widely used to improve the performance due to its higher access speed than global memory. However, executing atomic operations on shared memory is rather uncommon. There has been the research indicating that thread synchronisation in shared memory is slower than that in global memory [27]. We studied this phenomenon and found that although Nvidia does not reveal the implementation details of the atomic operations in global memory and shared memory, the underlying reason for this performance discrepancy may be because the atomic operations in global memory and shared memory are implemented in different ways. In shared memory, the atomic operations are implemented using the explicit lock and unlock. When multiple threads access the same location in shared memory, they are put in a mechanism like a loop and their atomic operations are processed in sequence. When a thread invokes the atomic operation to access the data, it locks the memory location, accesses the data and unlocks it after the operation is completed. In the global memory, however, the atomic operation is optimised and implemented with a single hardware instruction [27].

We reason that since multiple threads that are calling atomic operations are put in a loop and processed in sequence, the number of these threads should have the impact on synchronisation performance. Based on this reasoning, we conducted the benchmarking experiments, and we have new findings.

We wrote a benchmarking program. The key kernel of the program is shown in Algorithm 5.1. In the kernel, each thread performs the $atomicMin$ operation to write data into shared memory or global memory. The $atomicMin$ operation takes two parameters as input. The second parameter is the data that the operation is writing while the first is the memory location which the data is written into. The $atomicMin$ operation compares the data of the first parameter with the data in the memory location of the second parameter, and then the memory location will store the smaller value between them two. There are two arrays, *result* and *location*, in the program. An element in the *location* array holds a location that the data should be written in the *result* array. When the arrays are defined in global memory (or shared memory), the program accesses the global memory (or shared memory). The kernel is run with a single block of 1024 threads, which is the maximum number of threads that a thread block can support in our GPU device (GTX 780TI). The synchronisation degree (i.e., the number of threads that are storing the data in the same location in the *result* array) is controlled by setting the element values of the *location* array. The

Algorithm 5.1: Mini Benchmark

```
__device__ void benchmark(int *location, int *values, int *result)
{
  int v = values[thread_id];
  int l = location[thread_id];
  atomicMin(&result[l], v);
}
```

synchronisation degree varies from 2 to 512. When the degree is 2, every two threads write to the same location of the *result* array. When the conflict degree is 512, a half of threads in the 1024 threads all write to a same location of the *result* array and the other half write to another same location. The kernel was run 10 times for both accessing shared memory and accessing global memory. The average time for running the kernel with different synchronisation degrees is plotted in 5.3.

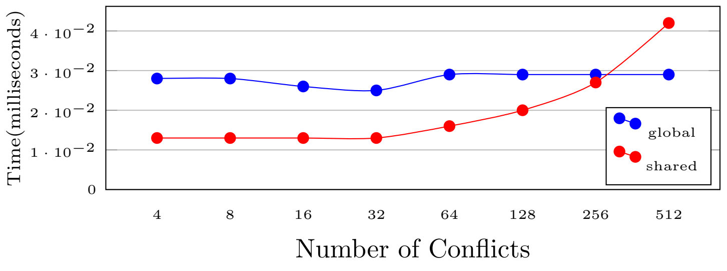


Figure 5.3: The run-time of the benchmark program with different synchronization degree in shared memory and global memory

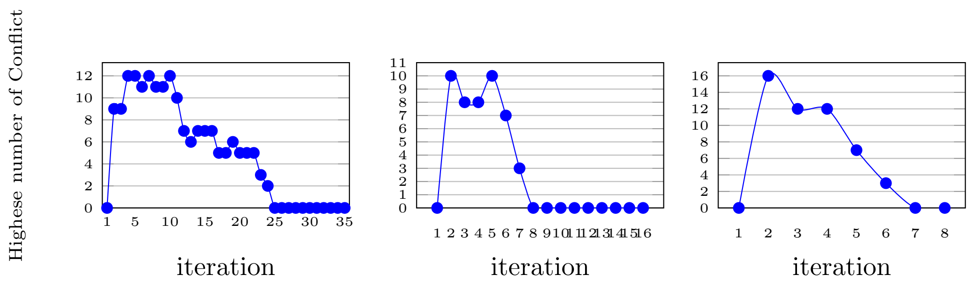
As can be observed from Figure 5.3, the average time spent in writing the data remains approximately unchanged in global memory as the synchronisation degree increases, while the time for writing data to shared memory increases as the synchronisation degree increases. When the synchronisation degree is higher than a certain value, writing data in shared memory takes longer than writing in global memory. This result can be explained as follows. The CUDA kernel runs in groups of 32 threads, which is called a warp. When they invoke the atomic operation for accessing shared memory, their operations are processed in sequence by the CUDA library. If some threads are accessing the same memory location, they need to wait for the lock of the memory location to be released. When more threads are accessing the same memory location (i.e., the synchronisation degree is higher), the longer the threads potentially have to wait and therefore delay the processing of other threads' atomic operations in the same warp. Although threads access the shared memory faster than the global memory, the longer delay caused by higher synchronisation degree will eventually cancel the speed advantage of share memory.

Our GPU device, GTX 780TI, supports running maximum 2048 threads and 32 thread blocks in a SM. Since a thread block is allocated with the separate shared memory space, running a GPU kernel with a higher number of thread blocks will lead to a lower synchronisation degree. If the maximum number of thread blocks is used, the number of threads in each thread block is $2048/32 = 128$, which means that the maximum synchronisation degree is 128. As shown in Figure 5, only when the synchronisation degree is more than approximately 256, will the synchronisation overhead in shared memory become higher than that in global memory. Therefore, we hypothesise that caching data access (and therefore synchronisation) in shared memory will benefit the performance when 32 thread blocks are used to run the GPU kernel in a SM. The number 128 is the maximum synchronisation degree in theory. We also conducted the following experiments to gain the insight into the realistic synchronisation degree in graph processing. In these experiments, we are running the single source shortest path (SSSP) algorithm with following 3 real world graphs, amazon0601, webGoogle and liverJournal. The number of thread blocks is set to be 32.

In the first experiment, we record for each graph the highest number of synchronisation among all thread blocks in each iteration, i.e., the highest number of threads that write the data simultaneously to the same shared memory location. The results are shown in Figure 5.4. It can be seen from the figure that the highest synchronisation degrees are only 12, 10 and 16 for the three graphs. This result suggests that when processing the three graphs, the actual synchronisation degree is much less than the threshold (256) shown in Figure 5.3.

After caching the data access and synchronisation in shared memory, the synchronised data access to global memory should be reduced. We conducted the experiments to show the reduction of the synchronised writes to the global memory. In the experiments, we processed the three graphs by using the shared memory and also by only using the global memory, and then recorded the number of writes in each iteration that refer to the same memory location in both cases. The results are shown in Figure 5.5. It can be seen from the figure that by caching the data access in shared memory, the synchronised writes to global memory, which is random writes, are significantly reduced.

When we ran the experiments for Figure 5.5, we also recorded the execution time of each iteration in both cases. The results are plotted in Figure 5.6. As can be seen from the figure, caching the data access in the shared memory leads to much less execution time, compared with performing atomic operations entirely in global memory. These results verify our earlier hypothesis, i.e., caching the data access and synchronisation in shared memory can improve the performance.



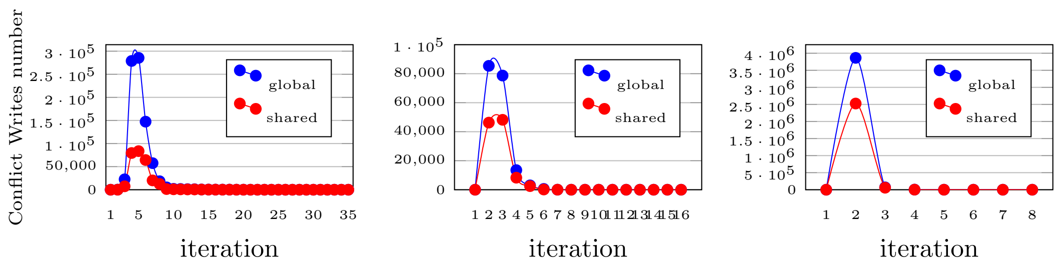
(a) amazon0601 (b) web-Google (c) LiverJournal

Figure 5.4: Maximum conflict among all thread blocks

5.3.2 Two-level GPU processing and memory access pattern

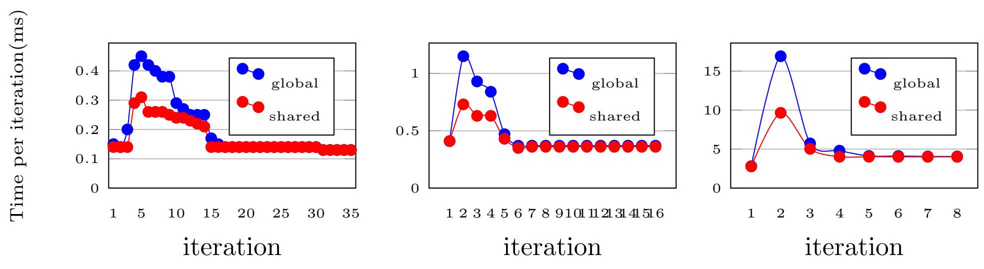
Based on the finding above, we propose a two-level execution mode as follows to reduce random access and the synchronisation overhead in global memory.

The iterative graph processing goes through a number of iterations to calculate the vertex values. In each iteration, a kernel, called the edge-processing



(a) amazon0601 (b) webGoogle (c) liverjournal

Figure 5.5: Conflict writes to global memory with and without using shared memory



(a) amazon0601 (b) webGoogle (c) Liverjournal

Figure 5.6: Execution time per iteration. (ms)

kernel, is launched with multiple thread blocks. A thread block processes an edge block. In a thread block, each thread processes one edge and calculates the value of the vertex associated with the edge in parallel. When a thread in a thread block obtains a new value for the vertex in the edge block, it accesses the shared index array and applies the atomic operation to write the new data to the local vertex-array of the edge block in shared memory. The atomic operation will perform synchronisation if more than one thread updates the data in the same location. Then the edge-processing kernel exits and another kernel, called global-updating kernel, is invoked to write the new values of the vertices in each local vertex array to the corresponding locations of the global vertex array in the global memory. In the global updating kernel, multiple thread blocks are generated, each block is used to write the data in a local vertex array to the global vertex array. In a thread block, a thread calls the atomic operation to write a data item to the global memory. Synchronisation is performed when the

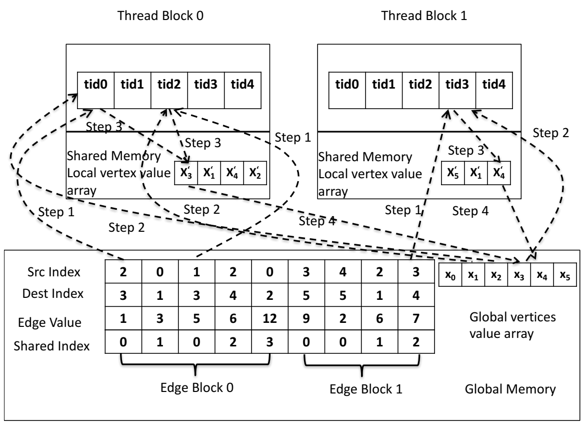


Figure 5.7: The process procedure inside edge blocks of graph shown in Figure 5.2

threads in different thread blocks update the data simultaneously to the same location in the global memory. After the updating is completed, the global updating kernel exits. Note that the synchronisation in global memory is not guaranteed before the global updating kernel exits.

With this two-level execution mode, each edge block is processed by a thread block in the GPU in the following 4 steps, which is shown in Figure 5.7. First, threads within each thread block read all information in the edge block in parallel. The threads with consecutive thread ID in a thread block read the edge information residing in contiguous global memory locations, thus providing the coalesced global memory access. In the second step, based on the vertex information fetched in the first step, the threads fetch the vertex values from the global vertex array to the shared memory of the thread block. Here, the access to the Global-Vertex array is random. In Step three, the threads compute the updated value for the destination vertex of each edge and write the result back to the shared memory. Synchronisation is performed when multiple simultaneous data writing refers to the same location. The last step performs the synchronisation between different thread blocks by writing the local vertex values to the global vertex value array. As can be seen, the memory access in the steps except step 1 is not coalesced. Although the random memory access can be eliminated by more sophisticated graph partitioning methods, these partitioning methods will increase the pre-processing time significantly. For example, in CuSha, the graph is first partitioned into disjoint sets of vertices. Each partition (shard) stores all the edges whose destinations are in that set. Then, the edges in a partition are sorted based on the ID of source vertices. By partitioning the graph in this way, the consecutive threads can access the consecutive memory locations in the global memory when reading/writing the values of destination vertices into/from the shared memory. However, such partitioning method incurs much longer pre-processing time due to the activities of sorting the edges and finding the appropriate partition sizes.

5.3.3 Implementing GPU-based graph processing algorithms using WolfGraph

Users can implement a broad range of graph processing algorithms with WolfGraph. In this section, we take the Single Source Shortest Path (SSSP) algorithm as an example to show how to write a GPU-based graph processing program using WolfGraph.

The edge-centric approach to implementing SSSP is to iteratively update the value of the destination vertex of every edge by adding the value of an edge to the value of the source vertex of the edge. The calculation repeats until the values of the destination vertices of all edges do not change anymore. Algorithm 5.4 presents the pseudo code of the SSSP functions.

Algorithm 5.2 shows the part of the pseudo code that runs on the host/CPU. The host iteratively launches the GPU kernels until the results converge (i.e., *not_converge* is true). In SSSP, convergence means that the path distance from the source to every vertex does not change anymore. At the end of each iteration, the device copies the value of the *not_converge* variable back to the host memory (line 8) and the CPU then determines whether the graph processing is completed according to the value of *not_converge*.

Algorithm 5.3 shows the kernel functions in WolfGraph. Each edge block is

Algorithm 5.2: Pseudo code of host function

```
/*host function*/
not_converge = true;
while(not_converge){
  not_converge = false;
  copy not_converge to GPU;
  process_edge();
  update_vertex();
  copy not_converge back to CPU;
}
```

processed by a thread block. In the first kernel *process_edge*, the consecutive threads read the edge information and use this information to initialize the vertex data. (Line 5-12). The access pattern to the Global memory in these steps is shown in Figure 5.7. The computation is performed by invoking the *compute* method defined by the user. Since multiple threads within the block may simultaneously update the same location in the shared memory, the atomic function is used to update the destination vertex. Because the order of function invocations is non-deterministic, the compute function must be both commutative and associative.

Then a second kernel *update_vertx* is launched. In this kernel, a flag called value_updated is used to indicates whether or not the vertex values are updated. It is initially set to false (line 23). Each thread invokes the *is_update* method (lines 24), which is another user defined function to check whether or not to write the updated value back to global memory. The threads will update the contents of global memory and set values_updated to true if the not_update method returns true. If values_updated flag is set to true, the vertices in the global vertex array are updated atomically with the newly computed value and the not_converge flag is set to true. Even though the memory accesses to global vertex array are not fully coalesced in this case, it requires less number of memory transactions than directly write to global vertex array.

Algorithm 5.4 presents the functions required to compute SSSP on a graph. In SSSP, every vertex holds a value (initially set to a very large number representing ∞) standing for the shortest distance from the source. Source vertex

Algorithm 5.3: Pseudo code of kernel function

```
__global__ process_edge(src_index, dest_index,shared_index,
    edge_values){
    /*parallel for edge blocks:*/
    shared share_local_vertex[N];
    /* step 1 fetch edge information,  coalesced access */
    source_vertex = src_index[tid];
    destination_vertex = dest_index[tid];
      shared_index = shared_index[tid];
      edge_value = edge_values[tid];

      /* step 2 initialise vertex value, non-coalesced access*/
      share_local_vertex[shared_index] = global[destination_vertex
          ];
        src_value = global_vertex[source_vertex];
    __synchronize;

    /* step 3 compute the update vertex value*/
    parallel for each thread invoke:
      compute(src_value, share_local_vertex[shared_index],
          edge_value);
}

__global__ update_vertex(global_vertex, shared_local_vertex);
    /* step 4 write back and check if the computation is converged
        */
    /*parallel for vertex in shared_local_vertex:*/
      value_updated = false;
      if(is_updated(shared_local_vertex[tid], global_vertex[
          destination_vertex])
      {
        atomicMin(&global_vertex[destination_vertex],
            shared_local_vertex[tid]);
        value_updated = true;
      }

      if(value_updated == true)
      {
        not_converge = true;
      }
```

value is set to 0. At the beginning of each iteration, the *init_shared_vertex* method loads the most updated vertex values into the block's shared memory. The *compute* function is act on every edge, it first computes the new distance for a destination vertex, then atomically choosing the minimum distance between the current and the calculated distances. The *is_update* function notifies the caller to execute the next iteration if the new distance of the destination vertex is smaller than its old value. As we can see, the user only need to provide the *compute* and *is_update* functions; hence making it easier to code graph processing algorithms using WolfGraph.

Algorithm 5.4: Pseudo code of SSSP implementation in WolfGrahp

```
__device__ void compute(src_value, share_local_vertex,  edge_value)
{
  if(share_local_vertex != INF)
  {
    atomicMin(&(share_local_vertex, src_value+edge_value));
  }
}

__device__ bool is_updated(shared_local_vertex,  global_vertex)
{
  if(shared_local_vertex < global_vertex)
  {
    return true;
  }
  return false;
}
```

5.4 Out of GPU memory processing

In the last section, we presented the design and implementation of in-memory processing of WolfGraph, i.e., the case where the entire graph can fit into the global memory of GPU. However, the size of GPU global memory is much smaller than the host memory. The sizes of real world graphs can vary from few gigabytes to terabytes, which are too large to be loaded into GPU global memory all at once. Therefore, in this section, we design an out-of-memory graph processing framework that can process such large-scale graphs.

5.4.1 Graph Partition and Computation

The general idea of designing an out-of-memory graph processing framework is to partition the graph into sub-graphs that can fit into the GPU memory and process these sub-graphs in GPU one at a time. There are two key issues that need to be addressed properly. First, we still want to reduce the pre-processing time as we do for in-memory graph processing. So the graph partition should not incur many pre-processing efforts. Second, a common problem of processing sub-graphs one at a time is that when a sub-graph is being processed or after it has been processed, the graph processing algorithm needs to access the previously processed sub-graph to update the newly calculated results. In the GPU environment, however, this requires the data exchanges between GPU global memory and CPU main memory, which will incur high overhead and should be minimised as much as possible. Next, we present the methods that we develop to address the above two issues.

As the graph data is read into the CPU memory, the data structure such as the one in Figure 5.2 is constructed in the CPU memory, including the edge list and the global-vertex-value array. To address the first issue, i.e., to minimise the time spent in graph partitioning, the out-of-memory graph processing framework splits the edge list into chunks (each chunk is a subgraph) in the similar way as we split the graph into edge blocks in the in-memory graph processing. Namely, a graph is partitioned into the equal-sized subgraphs in each of which the edges are in the same order as they are in the graph raw data. Therefore, there is no need to pre-process the graph. In the out-of-memory graph processing, we do not partition the global-vertex-value array. The reasons are two folds. First, to obtain the vertices for a subgraph, we have to search the global-vertex-value array to construct the vertex sub-array that contains the vertices in a subgraph, which incurs the pre-processing. Second, we argue there is no practical need to partition the global-vertex-value array. Since compared to the memory space occupied by edges, the memory space required by vertices is small. For example, in the graph arabic2005 [81], the edges take 10.24 GBytes memory space while the vertices only occupy 90.98 MBytes space (each vertex value is stored as an integer). The vertices of an entire graph can be easily accommodated in GPU global memory, even for very large-scale graphs.

The size of each subgraph is determined in the following way. Assume the size of GPU global memory size is G, and the entire graph has $|N|$ vertices and $|E|$ edges. Because we put the whole vertex array into the GPU global memory, the remaining memory space for holding the edge list of a subgraph is then $G - |N| * sizeof(vertex)$ (a vertex is represented as an integer or a floating point number depending on the graph processing algorithms). As shown in Figure 5.2, an edge is represented by 4 elements: source index, destination

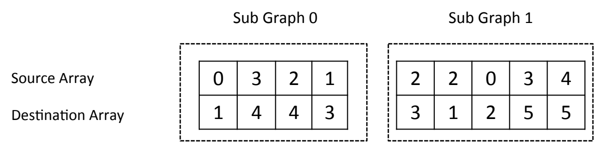


Figure 5.8: An example of partitioning graph in Figure 5.1a into 2 sub-graphs

index, edge value and shared index value. Therefore the size of an edge is $3 * sizeof(int) + sizeof(edge\_value)$ (an edge value is an integer or a floating point number depending on graph processing algorithms). The number of edges that GPU global memory can hold, denoted by e, can then be calculated by

$$e = \lfloor \frac{G - |V| * sizeof(vertex)}{3 * sizeof(int) + sizeof(edge\_value)} \rfloor \tag{5.1}$$

Therefore, this graph partition method does not require any pre-processing. It simply puts the first e edges in the graph data in the first sub-graph, next e edges in the second sub-graph, etc., as WolfGraph reads the graph data from the hard disk into the CPU memory. This process requires only one sequential read of the graph from the hard disk. An example is shown in Figure 5.8.

This graph partition method does not require any pre-processing. It simply puts the first e edges in the edge list in the first subgraph, next e edges in the second subgraph, etc., as Wolfgraph reads the raw graph data from the hard disk into the CPU memory. This process requires only one sequential read of the graph from the hard disk. As an example, the graph in Figure 5.1a can be partitioned into subgraphs shown in Figure 5.8.

Although the graph partition method presented above does not require pre-processing, the partition may cause frequent data exchange between GPU memory and host memory during the graph processing of the graph, which will hamper the performance. For example, Consider Figure 5.8. Let us focus on consider the edges $\langle 1, 3\rangle$ in subgraph 0 and edge $\langle 2, 1\rangle$ in subgraph 1.

WolfGraph first loads sub-graph 0 into GPU global memory and processes it. A thread is responsible for processing one edge, i.e., taking the value of the source vertex of the edge to update the value of the destination vertex. Since it is an iterative processing, the threads may need to go through a number of iterations to update the destination vertices until the values do not change anymore. In this example, when every thread processes its edge in the first iteration, thread 0, which processes edge $\langle 0, 1 \rangle$, uses the value of vertex 0 to update the value of vertex 1, thread 1 updates vertex 4, thread 2 updates vertex 4 and thread 3 updates vertex 3. Since vertex 1 is updated in the first iteration and its new value will be different from the one that is used by thread 3 to update vertex 3 in the first iteration, thread 3 needs to go into another iteration to update vertex 3 again. After vertex 3 is updated, thread 1 needs to run another iteration to update vertex 4 again. After that, the value of all vertices will remain unchanged. Then, the processing of sub-graph 0 is regarded complete. The global vertex array will have the latest values of all vertices in sub-graph 0. Next, WolfGraph clears the memory space occupied by the edge list of sub-graph 0 and loads sub-graph 1 into from CPU memory to GPU global memory. WolfGraph goes through the same process to update the values of the destination vertices in sub-graph 1. During the process, thread 3 takes the latest value of vertex 3, which is updated when processing sub-graph 0, to update vertex 5. Similarly, thread 4 takes the most recent value of vertex 4. Since vertex 1 is the destination vertex of edge $\langle 2, 1 \rangle$ in sub-graph 1, vertex 1 will have new value after the processing of sub-graph 1 is completed. Therefore, we need to load sub-graph 0 into GPU again and re-process sub-graph 0. Such a process repeats until the values of all vertices in the entire graph do not change.

WolfGraph first loads subgraph 0 into GPU global memory and processes it. A thread is responsible for processing one edge, i.e., taking the value of the source vertex of the edge to update the value of the destination vertex. Since it is an iterative processing, the threads may need to go through a number of iterations to update the destination vertices until the values do not change anymore. In this example, when every thread processes its edge in the first iteration, thread 0, which processes edge $\langle 0, 1 \rangle$, uses the values of vertex 0 and the edge to update the value of vertex 1. Thread 1 updates vertex 4, thread 2 updates vertex 4 and thread 3 updates vertex 3. Since vertex 1 is updated in the first iteration and its new value will be different from the one that is used by thread 3 to update vertex 3 in the first iteration, thread 3 needs to go into another iteration to update vertex 3 again. After vertex 3 is updated, thread 1 needs to runs another iteration to update vertex 4 again. After that, the values of all vertices will remain unchanged. Then, the processing of subgraph 0 is regarded complete. The global-vertex-value array will have the latest values of all vertices in subgraph 1. Next, WolfGraph clears the memory space occupied by subgraph 0 and loads subgraph 1 from CPU memory to GPU global memory. Wolfgraph goes through the same process to update the values of the destination vertices in subgraph 1. During the process, thread 3 takes the latest value of vertex 3, which is updated when processing subgraph 0, to update vertex 5. Similarly, thread 4 takes the most recent value of vertex 4 to update vertex 5. Since vertex 1 is the destination vertex of edge $\langle 2, 1 \rangle$ in subgraph 1, vertex 1 will have new value after the processing of subgraph 1 is completed. Vertex 1 is the source vertex of an edge in subgraph 0, and therefore, we need to load subgraph 0 to GPU again and re-process subgraph 0. Such a process repeats until the values of all vertices in the entire graph do not change anymore.

The fundamental reason behind the repetitive loading of a sub-graph is because the graph partitioning method essentially partitions the graph randomly and it may put an edge and its child edges (Edge A is called the child of edge B if B's destination vertex is A's source vertex) in different sub-graphs. If an edge's child edges are processed before the edge, then the sub-graph that contains the child edges needs to be loaded and re-processed again after the edge is processed.

5.4.2 Concatenate Edge List representation

After identifying the reason behind the repetitive loading of sub-graphs, a method called Concatenated Edge List (CEL) is designed in WolfGraph to reduce the frequent data exchange between host and GPU's memory. With CEL method, instead of transferring the sub-graph into GPU, we transfer the concatenate edge list into GPU.

For each graph being processed, we first build the sub-graphs with the method described in the last section. During the computation, before transferring a sub-graph into GPU, we build the CEL for this sub-graph. The CEL is built in the following way, for each destination vertex in the sub-graph, we search all the sub-graphs that have been processed already since we process sub-graphs according to their sub-graph ID, the processed sub-graphs are those sub-graphs with smaller ID than the current one. Then from these sub-graphs, we select all child edges of the current destination vertices set and append these edges to the current sub-graph. Then we transfer this concatenate edge list to the GPU, and process it with the in-GPU-memory engine.

As discussed in the last subsection, if a destination vertex in the current sub-graph has child edge in another sub-graph, and if this sub-graph has been processed already, it needs to be re-processed because the value of some edges in it may change after processing the current graph. On the other hand, if the sub-graph with the child edges has not been processed, there is no need to process it immediately after processing the current sub-graph since it will be computed later and the updated vertex value can be used directly. Therefore, when constructing the CEL, there is no need to consider the sub-graphs that haven't been computed.

The CEL can be built with Algorithm 5.5. The function takes a list of sub-graphs and the ID of the sub-graph that currently being processed as input. It first adds the size of each sub-graph that has been computed together to find out the *pe*, which is the number of edges that has been processed. Then a temporary array *offset* is created, the size of this array is the number of sub-graphs has been computed plus 1. In this array, each element corresponds to a sub-graph, so the sub-graph ID is used to index this array. The content of this array is the offset of the first edge in the corresponding sub-graph to the first edge in the first sub-graph. An extra element is needed to store the offset of the last sub-graph being processed (Line 3-7). Consider graph in Figure 5.9 as an example. When processing sub-graph 1, the size of *offset* array is 2, the content of two elements is 0 and 4 respectively, because the offset of the first edge to itself is 0, and the size of first sub-graph is 4, the offset of the first edge in the second sub-graph is 4.

After building the *offset*. The algorithm iterates over the destination vertices in the current sub-graph. For each destination vertex, the algorithm first locates its corresponding link-list from the mapping array (Line 8-9). Then the algorithm iterates the elements stored in the link-list (Line 10). For each element in link-list, the algorithm compares its value *gi* with *pe*, if *gi* is larger than *pe*, that means the edge corresponds to this index is in the sub-graph that hasn't been processed, so there is no need to processed this link-list (Line 11). If the global index is smaller than *pe*, the algorithm searches the offset array (Line 12). For location i in offset array, it compares value *gi* with value stored in location $i-1$ and $i+1$ (except for first and last element, we only compare $i+1$ and $i-1$ respectively). If the value *gi* is greater than $i-1$ and smaller than $i+1$, then the edge corresponds to the *gi* value is in sub-graph i, and location of this edge in sub-graph i can be computed with *gi-offset[i-1]*. In the end, the edge stored in this location is appended to the CEL (Line 13-18).

For example, consider the destination vertex 3 in sub-graph 1 in Figure 5.9, by looking at mapping array, we can get its corresponding link-list with element $[1, 7]$, then we start comparing 1 with first element in array *offset*, which is 0. Since 1 is greater than 0 and this is the first element, we know the corresponding edge with *gi* value 1 is in sub-graph 0. Hence, we can locate the edge in sub-graph 0 with index $1 - 0 = 1$, which is edge $\langle 3, 4\rangle$, and append this edge to CEL.

The complete example is shown in Figure 5.9. In this example, the graph from Figure 5.1a is partitioned into two sub-graphs. The sub-graph 0 is processed in the first iteration. Because it is the first sub-graph, there is no need

Algorithm 5.5: Build CEL with Mapping array

```
build_cel(sub-graphs s, id)
{
  $pe = 0$;
  $offset[] = malloc(id + 1 * sizeof(int))$;
  for $i < id$ do:
    $pe += s[id].size$;
    $offset[i + 1] = s[i].size$;
  for destination vertices $u$ in $s[id]$ do:
    link list $l = mapping\_array[u]$
    $gi = l.head.value$;
    while $gi < pe$:
      for $i$ in $offset[]$:
        if $i$ == 0 and $gi < offset[i + 1]$:
          l.append($s[0][gi]$);
        else if $i$ == id-1 and $gi > offset[i - 1]$:
          l.append($s[id - 1][gi - offset[i - 1]]$);
        else if $gi > offset[i - 1]$ and $gi < offset[i + 1]$
          l.append($s[i][gi - offset[i - 1]]$);
      $gi$=$l.next.value$;
}
```

to build the CEL. The sub-graph 1 is processed in the second iteration, the destination vertices in sub-graph 1 are 1,2, 3 and 5. Since the sub-graph 0 does not have vertex 5, the algorithm appends all edges start with vertices 1,2 and 3 from sub-graph 0 to it, which are edges $\langle 3, 4\rangle$, $\langle 2, 4\rangle$ and $\langle 2, 4\rangle$.

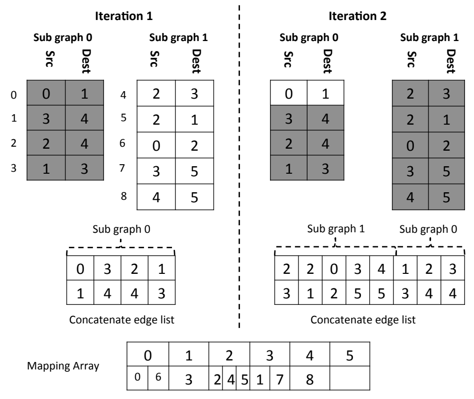


Figure 5.9: An example of building the concatenate edge list from 2 sub-graphs. The gray area represent the edges used to build the concatenate edge list in each iteration.

If the constructed CEL is too large to fit into GPU memory, we evenly split the sub-graph into two sub-graphs, and build the CEL with these two small sub-graphs. If the CEL is still too larger to fit in GPU, we split the small sub-graph into two even smaller sub-graphs to build CEL. This process will be repeated until constructed CEL can fit into GPU memory.

5.5 Out-of-Core Graph processing

In the last few sections, we assumed that the host memory is large enough to store the entire graph. However, nowadays, the size of real-world graphs increasing at an exponential rate, and this increasing rate is much faster than the increasing of memory size. Hence, we cannot assume the graph can always fit in the system memory.

The solution to this problem is to use the hard disk as an extension of the memory. In this section, we extend the WolfGraph with an out-of-core processing engine, so it can be used to process such graphs.

5.5.1 Out-of-Core graph partitioning

The idea to process large scale graph from disk is to partition the graph into sub-graphs, then store the sub-graphs on the hard disk, and process one sub-graph at a time. To achieve good performance when processing from hard disk, we made the following design requirements. First, the partitioning algorithm should only iterate graph once. Moreover, the access to the graph data has to be in the sequential order; this is because hard disk operation is time-consuming, and therefore, it should be minimised. Also, random access to the hard disk is much slower compare to the sequential access. Second, as we discussed in the last section, after partitioning the graph, each sub-graph should only be processed once in the memory. Therefore, both I/O and processing time can be reduced.

To meet these requirements, we propose the following algorithm to partition the graph. We first compute the number of partitions based on the graph size and GPU's memory size. We use GPU's memory size instead of host memory size is due to following reasons: First, in WolfGraph, the GPU is used to accel-

Algorithm 5.6: Graph partitioning

```
 1 Graph_partition(Graph $G$)
 2 {
 3     $partition\_no = \frac{G.size}{GPU\_memory\_size}$
 4     $v\_sets$ = partition_vertics($G.v$)
 5     $buffers[][] = new[partition\_no][v\_sets.size]$
 6     while(read edge $e$ from $G$)
 7     {
 8         $i = find\_partition(e.src)$;
 9         $j = find\_partition(e.dest)$;
10         write_to_buffer($i, e$);
11         write_to_buffer($j, e$);
12     }
13 }
14
15 write_to_buffer($location$, $edge$)
16 {
17     if $buffers[location]$ not full
18         $buffers[location]$.append($edge$);
19     else
20          flush_to_disk($buffers[location]$);
21          $buffers[location]$.append($edge$);
22 }
```

erate the computation. Hence, if the size of each sub-graph can fit into GPU, it can be transferred into GPU directly. Otherwise, the engine will build the concatenate edge list to split the sub-graph, which will introduce overhead to execution time. Once the number of sub-graph has been decided, we can evenly split the vertices into disjoint sets, and assign one set to a sub-graph. After this step, for each vertex, we assign both in and out edges of this vertex to the same sub-graph. Once the partition is complete, each partition will be written to a binary file. Also, to reduce the memory usage during the graph partitioning, we create a buffer for each sub-graph and write the edge into the buffer during the main iteration, once the buffer is full, we flush its content to the hard disk. The whole algorithm is outlined in Algorithm 5.6.

It is clear that our algorithm only needs to iterate the graph once (line 6-11). Hence, the disk accessing time is minimised. Also, by putting the edges that have either a destination or a source vertex in the same sub-graph, the resultant sub-graph is effectively a big concatenated edge list. As we discussed in the last section, each concatenated edge list only need to be processed once in the memory.

5.5.2 Out-of-Core processing

In WolfGraph, the out-of-core processing is done through invoking the in-memory processing engine introduced in Section 5.2. If the sub-graph cannot fit into GPU memory, we can use the method described in Section 5.4 to process it. The major difference between out-of-core processing and in-memory processing is how we synchronise updated vertex values between different sub-graphs.

When synchronising between the various sub-graphs, there are two scenarios need to be considered. In the first situation, after loading one sub-graph into the memory, the memory still has enough space to store all vertices from this graph. In this case, we load the entire vertices into the memory, which will be shared between all sub-graphs. Each sub-graph reads the updated vertices value from this shared memory space before processing and writes the newly computed result back to the memory when the computation is done.

In the second case, when all vertices cannot fit into the memory, we create a vertex file for each sub-graph, which is used to store the update vertex values of this sub-graph. During the execution, we load both sub-graph and vertex file into memory, fetching the updated vertex values from the vertex file and assign these values to the corresponding edges in the sub-graph. Once the processing is complete, we keep the updated vertex values in the memory, then iteratively load the vertex file of unprocessed sub-graphs into the memory, updates the vertices in these files and writes the result back to the disk. Once vertex files from all unprocessed sub-graphs have been updated, the current results are written back to the disk. However, not all unprocessed sub-graphs need to be updated in each iteration. For example, some sub-graphs may not share any vertices with the sub-graph that currently processed. In such case, it should not be loaded. To solve this problem, another file is created for each sub-graph. This file keeps track of other sub-graphs that has common vertices with it associated sub-graph. Hence, by reading this file, only the sub-graphs that have the common vertices will be loaded into memory.

Because the sub-graphs are partitioned based on the GPU memory size,

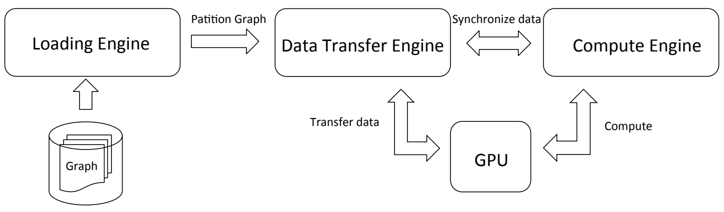


Figure 5.10: Architecture of WolfPath framework.

so one or more partitions can fit in host memory. Hence, the disk I/O and the in-memory computation can be overlapped. To achieve this, the execution engine loads a sub-graph first, after this sub-graph is loaded into the memory, the engine will launch another thread to load the next sub-graph. At the mean time, the current thread is transferring the data to the GPU, waiting for the GPU to finish its job, and then updating the vertex values. This process will repeat until all the sub-graphs have been computed.

5.6 Implementation of WolfGraph

The WolfGraph consists of three main components: Loading Engine, Data Transfer Engine, and Compute Engine. Figure 5.10 shows the general software architecture of WolfGraph. In this section, we describe selected details of these components.

5.6.1 Loading Engine

The Loading Engine is responsible for (1) load-balanced edge block creation and (2) providing graph partitioning logics.

Designing an efficient format for storing the Edge Block is paramount for good performance. In WolfGraph, we store the Edge Block in the following way.

We first create an array called Edge Block List, which is a flat array of pointers, each pointer points to an Edge Block. The Edge Block consists of four arrays as described in Section 5.3.1. The size of Edge Block List array and each Edge Block are determined by the number of edges in the graph and number of thread blocks used in graph processing.

WolfGraph uses user defined thread block size T to determine the size of Edge Block List array and each Edge Block. Assume the input graph has E edges, the size of Edge Block List B is $\lceil \frac{E}{T} \rceil$, and the size of each Edge Block are $\lceil \frac{E}{B} \rceil$. By computing the size of each array, we can statically allocate memory for them, which can reduce the significant amount of time that used in resizing and reallocating in dynamic memory allocation. Then WolfGraph reads the graph data from the hard disk sequentially, and stores the edge information in each Edge Block in the same order. Once it fills all edges in one Edge Block, it moves to the next Edge Block. Therefore, we only need to iterate the entire graph once to construct the Edge Blocks.

5.6.2 Data Transfer Engine

The data transfer engine aims to transfer data between GPU/host memory and construct the Concatenate Edge List if necessary.

In data transfer engine, the data is transferred through the memory-copy operation provided by CUDA. We use CUDA stream operation to overlap the data transfer and the computation. For different Edge Block, each Stream created by the *StreamCreator* inside the Data Transfer Engine typically issues multiple $MemcpyAsync()$ operations and graph computation kernels asynchronously. Therefore, the data transfer and the computation is overlapped.

When building the Concatenate Edge List, each Edge Block needs to access edges stored in other sub-graphs. Hence, to enable fast access to edges from other Edge Block, we give a global index to each edge and use a mapping array to build the relationship between each destination vertex and the edges that start with this vertex. The mapping array is constructed while loading the graph from disk to memory. We first construct an array indexed by the vertex ID, each array element points to a link list. Because we read edges from disk sequentially and write them in Edge Blocks in exact same order, we use a counter to keep track its global index, that is, every time we add an edge to the Edge Block, we increment the counter by 1. For each edge added to the Edge Block, we use its source vertex ID to locate its position in mapping array and append its global index value to the corresponding link list. Therefore, the global index value stored in each link list is in ascending order.

5.6.3 Computation Engine

The Computation Engine is mainly responsible for GPU in-memory computation and to send feedback information to the Data Transfer Engine about the results and termination condition used for the next iteration. The detail of Computation Engine is discussed in Section 5.3.2.

5.7 Evaluation

In this section, we evaluate the performance of WolfGraph using three types of graph dataset: small graphs that can fit into the GPU memory (called in-GPU-memory graphs in the experiments), middle size graphs that can fit into the CPU memory, but cannot fit in the GPU memory (called out-of-GPU memory graphs) and large graphs that cannot fit into the CPU memory (called out-of-core graphs). The size of a graph is defined as the amount of memory required to store the edges, vertices, and edge/vertices values in user-defined datatypes.

In-GPU-memory graphs are used to evaluate WolfGraph against other state-of-the-art GPU-based in-memory graph processing systems, including Cusha [65] and Virtual-Warp-Centric [56]. Out-of-GPU memory and out-of-core graphs are used to compare WolfGraph with other out-of-core frameworks that can process large graphs on a single PC (e.g., GraphChi and X-Stream).

The twelve graphs listed in Table 4.1 are publicly available. They cover a broad range of sizes and sparsity and come from different real-world origins. For example, *Live-Journal* is directed social networks, which represent friendship among the users. *RoadNetCA* is the California road network, in which the edges represent roads and the vertices represent the intersections. *WebGoogle*

Table 5.1: Real world graphs used in the experiments

In-GPU-memory Graph		
Graph	Vertices	Edges
RoadNet-CA [80]	1965206	5533214
amazon0601 [80]	403394	3387388
Web-Google [80]	875713	5105039
LiveJournal [80]	4847571	68993773

Out-of-GPU memory Graph		
Graph	Vertices	Edges
orkut [80]	3072441	117185083
hollywood2011 [13]	2180653	228985632
arabic2005 [14]	22743892	639999458
uk2002 [13]	18520486	298113762

Out-of-Core Graph		
Graph	Vertices	Edges
twitter [73]	41652230	1468365182
FriendSter [80]	65608366	1806067135
sk2005 [14]	50636154	1949412601
uk2005 [14]	39459925	936364282

is a graph released by Google, in which vertices represent web pages and the directed edges are links. *orkut* is an undirected social network, in which vertices and edges represent the friendship between users. *uk-2002* is a large crawl of the .uk domains, in which vertices are the pages and edges are links.

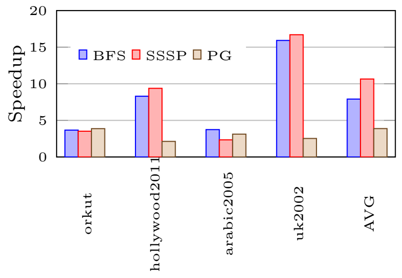
We choose three widely used graph processing algorithms to evaluate the performance, namely Breadth First Search (BFS), Single Source Shortest Paths (SSSP) and PageRank(PG). The PageRank algorithm was set to run 10 iterations.

The experiments were conducted on a system with a Nvidia GeForce GTX 780Ti graphic card, which has 12 SMX multiprocessors and 3GB GDDR5 RAM. On the host side, we use the Intel Core i5-3570 CPU operating at 3.4 GHZ with 32 GB DDR3 RAM. The benchmarks were evaluated using CUDA 6.5 on Fedora 21. All the programs were compiled with the highest optimisation level (-O3).

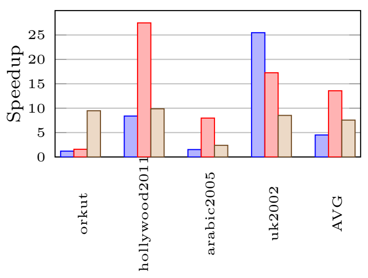
5.7.1 Performance Evaluation

Comparison with Out-of-GPU memory Frameworks

In this section, we examine the WolfGraph's ability to process large graphs which cannot fit into GPU memory. To the best of our knowledge, the state-of-the-art GPU-based graph processing frameworks [65] [146] [40] assume that the input graphs can fit in the GPU memory. Therefore, in this work, we compare WolfGraph (WG) with two CPU-based, out-of-memory graph processing framework: GraphChi (GC) [76] and X-Stream (XS)[109]. To avoid the disk I/O overhead in systems such as GraphChi and X-Stream, the dataset selected in the experiments can fit in host memory but not in GPU memory.



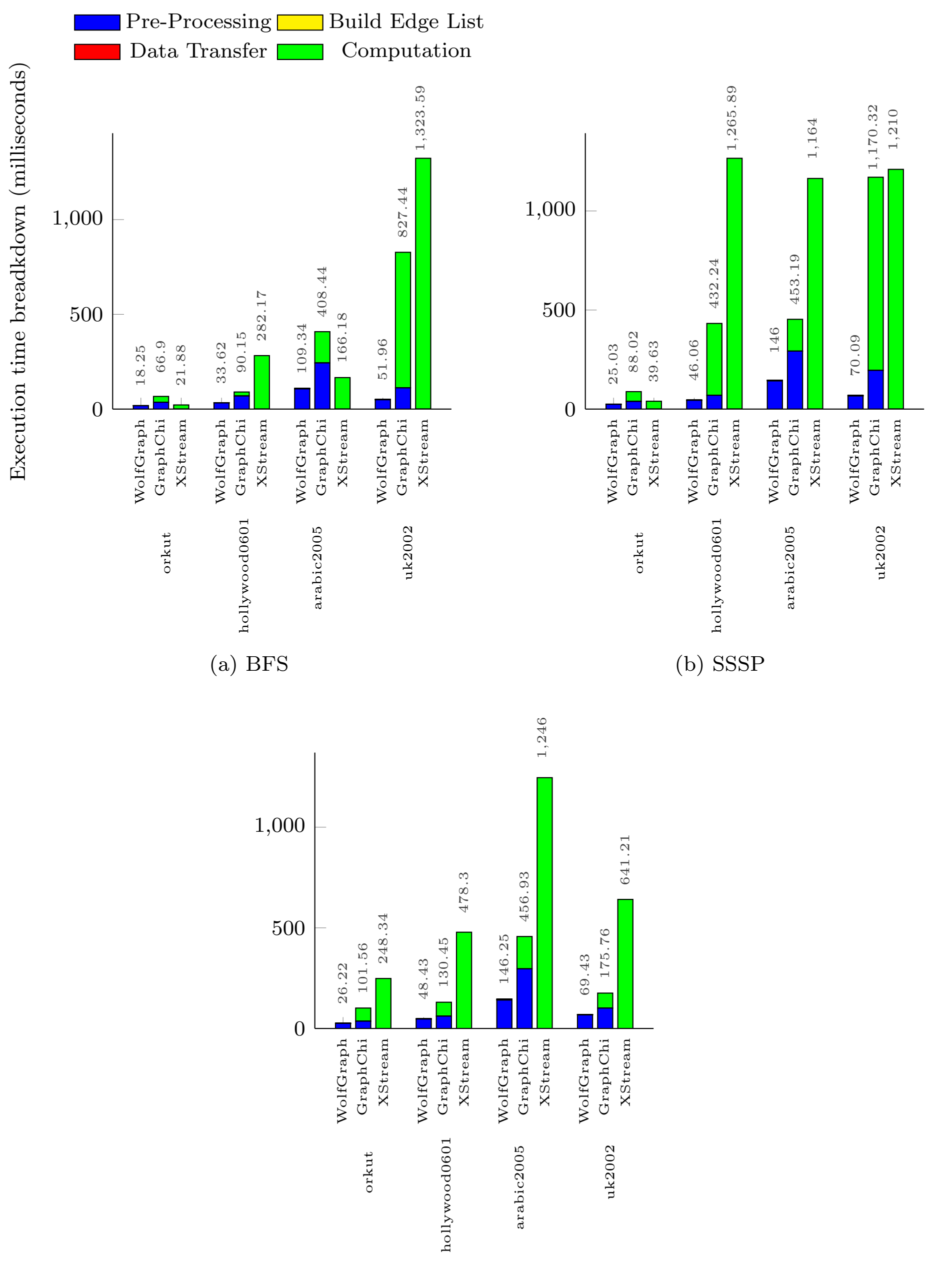
(a) Speedup over GraphChi



(b) Speedup over X-Stream

Figure 5.11: The speedup of WolfGraph over GraphChi and X-stream

Figures 5.11a and 5.11b show the speedup of WolfGraph over GraphChi and X-Stream, respectively. It can be seen from these figures that WolfGraph achieves an average speedup of 7.48 and 8.55 over GraphChi and X-Stream (running with 4 threads), respectively, despite its need to move the data between GPU and CPU. To analyse the performance in detail, Figure 5.12 shows the detailed time breakdown of three frameworks. We define the pre-processing time as the time spent in loading the graph from the hard disk to the CPU memory and constructing the designed data structures. Computation time refers to the time taken in actual execution of the algorithm. The time of building CEL records the time spent in building the concatenated edge list. Data Transfer is time taken by WolfGraph in transferring the data between host and GPU.



(a) BFS

(b) SSSP

(c) PAGERANK

Figure 5.12: Execution time breakdown of WolfGraph, GraphChi and X-Stream on out-of-GPU-memory graphs. Reported times are in seconds.

Since X-Stream does not require any pre-processing and the computation is overlapped with I/O operations, we use the total execution time of the system as the comparison.

As can be observed from the time breakdown shown in the Figure 5.12, in most cases, pre-processing time dominates the execution time in both WolfGraph and GraphChi. Therefore, reducing the pre-processing can improve the overall performance significantly. In detail, the performance improvement of WolfGraph over GraphChi and X-Stream is attributed to the following factors. First, the computation time of WolfGraph is shorter than that of GraphChi by orders of magnitude. The edge-centric processing model used by WolfGraph can fully utilise the massive parallel processing power provided by GPU, while the GraphChi and X-Stream are CPU-based, which limits the degree of parallelism. Second, the pre-processing time in WolfGraph is less than half of the pre-processing time in GraphChi. The pre-processing time of WolfGraph is almost equal to the graph loading time, and no other pre-processing operations are needed while GraphChi needs to convert the raw graph into the graph stored in the "shard" structure and sort the edges in each shard.

Comparison with Out-of-Core memory Frameworks

We then evaluate the WolfGraph (WG)'s out-of-core performance with GraphChi (GC) and X-Stream (XS). In these experiments, we use four out-of-core graphs from Table 5.1. These graphs have billions of edges, which make them unable to fit into host memory. In these experiments, we recorded the total execution time of both frameworks. The results are listed in Table 5.2.

As shown in these Tables, compare with GraphChi and X-Stream, the WolfGraph achieves 2-3X average speedup, this is because when processing large-scale graphs from secondary storage, the execution time is dominated by the disk I/O. For example, when running BFS algorithm on graph twitter with GraphChi, the execution time of BFS algorithm is around 600 seconds out of 2715 total execution time. Therefore, the performance gain due to parallelization is not as significant as processing small/middle size graphs.

Table 5.2: Execution time of WolfGraph, GraphChi and X-Stream on out-of-core graphs. Reported times are in seconds.

BFS	WolfGraph	GraphChi	X-Stream
Twitter	1415.69	2501.2	1843.45
FriendSter	2418.99	3415.58	7743.9
sk-2005	869.05	2159.11	11178.4
uk-2005	1332.46	24360	15033.7
SSSP	WolfGraph	GraphChi	X-Stream
Twitter	1589.32	2715.47	3672.7
FriendSter	2613.29	3593.23	13980.3
sk-2005	983.35	2347.24	16896.6
uk-2005	1732.54	24960	21534.8
PageRank	WolfGraph	GraphChi	X-Stream
Twitter	1654.67	3036.8	1900.83
FriendSter	2952.48	4594.75	3296.23
sk-2005	1034.38	2847.15	2653.49
uk-2005	729.4	996.32	1241.8

Comparison with GPU In memory Frameworks

The results shown in the last two sections demonstrated the WolfGraph's ability to process large graphs that do not fit into GPU memory and CPU memory. Recall that the other design objective of WolfGraph is that it should perform as good as other existing In-GPU-memory graph processing frameworks. In this section, we examine the WolfGraph's in-memory performance for small graphs by comparing it with the state-of-the-art in-GPU-memory processing solutions like CuSha [65] and Virtual Warp Centric [56]. In the experiments, we use the CuSha-CW method, because this strategy provides the best performance in all CuSha strategies. Both CuSha and Virtual Warp Centric apply multi-level optimisations to the in-memory workloads.

The breakdown performances are listed in Figure 5.13. As can be seen from these figures, WolfGraph outperforms CuSha and Virtual Warp Centric; this is due to following reasons: First, in all three frameworks, WolfGraph has the shortest pre-processing time. Because in WolfGraph, the pre-processing is only responsible for reading the data from hard disk and stores these data into Edge List structure. On the other hand, CuSha and VWC require more complicated pre-processing, such as sorting the data, counting the edge degree, etc. Second,

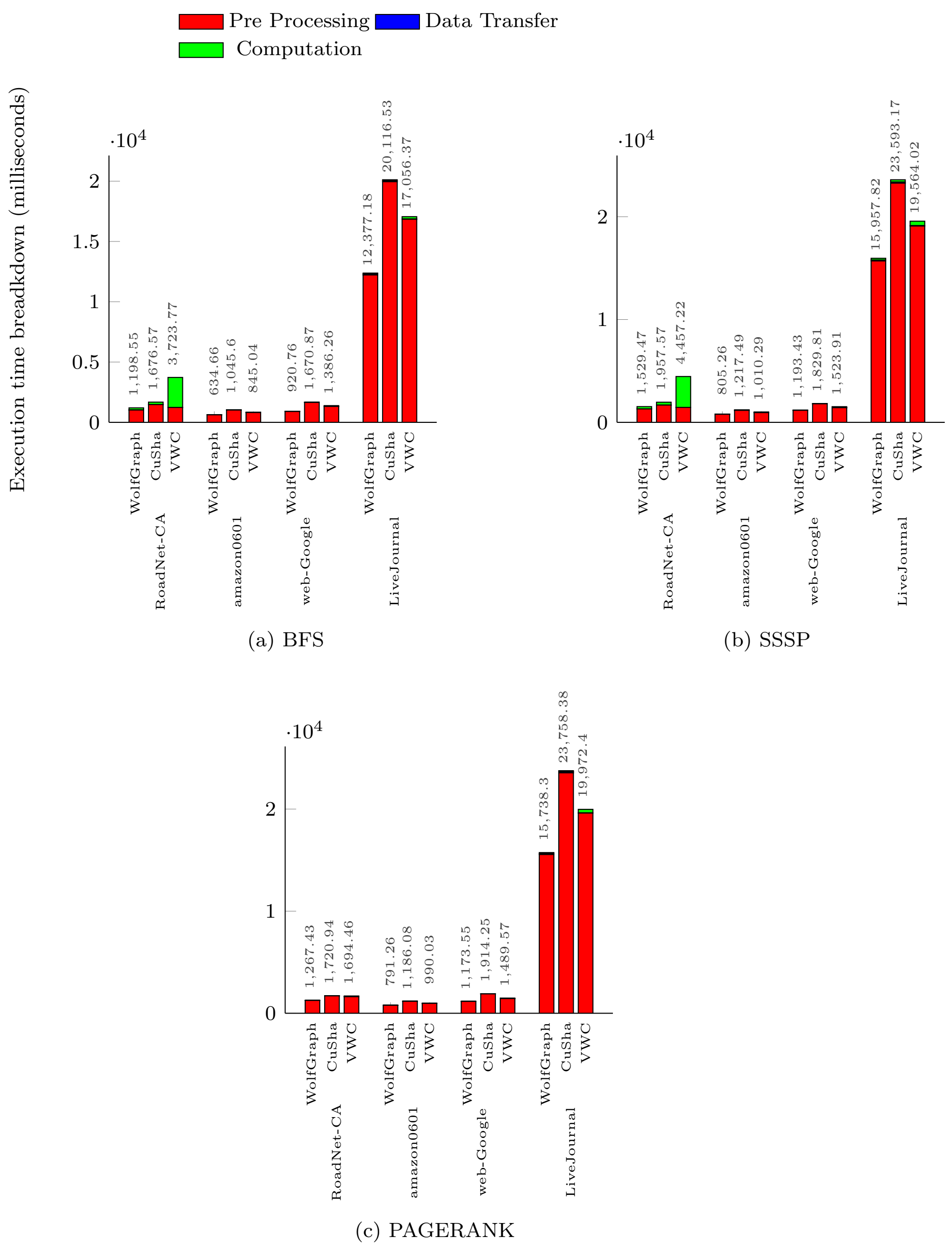


Figure 5.13: Execution time breakdown of WolfGraph, CuSha and VWC on in-GPU-memory graphs. Reported times are in milliseconds.

the data structure designed in WolfGraph can guarantee the coalesced memory access to the global memory, and maximise the GPU utilisation due to the use of edge-centric processing model. Third, due to the two-level synchronisation strategy used in WolfGraph, the computation time between WolfGraph and CuSha are very similar. However, the graph representation employed by CuSha incurs the longest pre-processing time among three solutions. Hence, the overall execution time of WolfGraph is much shorter than CuSha.

All three sets of experiments conducted above show an interesting fact, compare with pre-processing time, the computation time and data transfer time are much shorter. Based on this finding, we argue that when designing modern parallel graph processing system, especially on GPU, it is more important to improve the performance of pre-processing than computation.

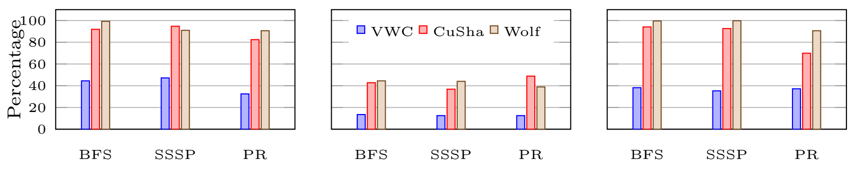
5.7.2 Global Memory efficiency

The reason we adopt edge-centric computation and representing the graph as edge list is to enforce the coalesced access to the global memory. This section evaluates this design decision by comparing the average global memory load efficiency, the average global memory store efficiency and warp execution efficiency of WolfGraph with CuSha and Virtual Warp Centric while processing *LiveJournal* graph. The algorithms used in these experiments are BFS, SSSP and PageRank. The results are shown in Figure 5.14.

The global memory load efficiency is the ratio of requested global load throughput to required global load throughput. It indicates how well the threads within a kernel read from the global memory: a high value shows that more read operations are performed. As can be seen from Figure 5.14a, VWC has the lowest load efficiency (41.4% on average); this is due to the non-coalesced access to the global memory. CuSha and WolfGraph achieve 89.6% and 93.6% average global memory load efficiency respectively, this is because both of these two frameworks provide coalesced access to the global memory.

The global memory store efficiency indicates the ratio of the global memory write throughput achieved by the kernel to the global memory store throughput that is needed by the kernel. This value shows how well the threads within a kernel write to the global memory. As we can see from Figure 5.14b, the average store efficiency achieved by VWC is 12.8%. On the other hand, the store efficiency achieved by CuSha and WolfGraph are 42.8% and 42.5% respectively. In both WolfGraph and CuSha, the store operation is performed in parallel, however, in VWC, within each virtual warp, only one thread is used to update the vertex value, which results in a lower store efficiency. For both three solutions, the average global memory store is lower than load mainly because of the store operation is not fully coalesced.

Warp execution efficiency is defined as the ratio of average active threads in a warp to the maximum possible active threads per warp supported by multiprocessor. It indicates how well the GPU hardware resources are utilised. As shown in Figure 5.14c, compare to CuSha (85.5% on average) and WolfGraph (96.7% on average), the VWC has much lower warp execution efficiency (36.9%), this is due to the imbalance work distribution among the thread blocks in VWC. On the other hand, both CuSha and WolfGraph evenly distribute the workload to thread blocks, hence improve the warp execution efficiency. Also, in CuSha, the warp execution efficiency is bounded by the window size (the set of edges in shard j that are involved during processing of shard i [65]). WolfGraph does not have such limitation because of each thread block only responsible for the edge block assigned to it.



(a) Load Efficiency (b) Store Efficiency (c) Warp Efficiency

Figure 5.14: Average profiled efficiencies of WolfGraph with CuSha and VWC on liveJournal graph.

5.7.3 Memory occupied by different graph representation

We evaluate the cost of using Edge list representation in terms of memory requirement over CSR and CuSha's CW representation.

Table 5.3: The execution time of out-of-GPU memory WolfGraph and in-GPU memory VWC on different algorithms and graphs.

Graph		BFS	SSSP
orkut	Wolf	18.25	25.03
	VWC	31.38	35.26
hollywood	Wolf	33.62	46.06
	VWC	45.98	59.58
uk-2002	Wolf	51.96	70.1
	VWC	58.42	74.58

The Edge List representation takes $3 * |E| * sizeof(index) + (2 * |E| + |V|) * sizeof(Value)$, which is same as the CuSha's CW representation. However, the CSR representation only requires $(|E| + |V|) * sizeof(index) + (|E| + |V|) * sizeof(value)$. The Figure 5.15 shows the actual memory consumed by WolfGraph Edge List, CuSha-CW and CSR. WolfGraph and CuSha-CW take 2.45x and 2.46x more space on average than CSR. The CuSha uses a little bit more memory than WolfGraph, which is due to the dynamic data structure overhead over the ordinary array.

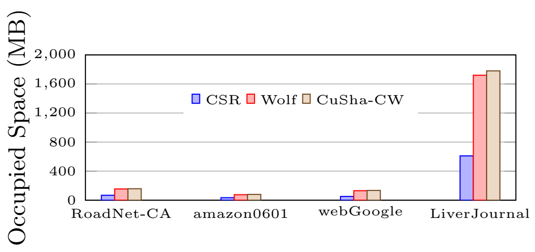


Figure 5.15: Memory occupied by each graph using CSR, CuSha-CW, WolfGraph Edge list representations over all benchmarks.

The increased memory consumption leads to the following situation: it is possible that some graphs can fit into GPU memory with CSR representation, but unable to do so with WolfGraph or CuSha's representation. To test the WolfGraph's performance in this scenario, we conduct the following experiment. We choose three graphs, orkut, hollywood2001 and uk-2002, all these graphs can fit into GPU with CSR representation but need to split into two Concatenate Edge List in WolfGraph's representation, the benchmark algorithms we use are BFS and SSSP, the results are list in Table 5.3.

As can be seen from the Table, although build concatenate edge list and transfer data to GPU add extra overhead, WolfGraph still outperformed the VWC in both cases. This is because WolfGraph has shorter pre-processing time, and faster computation performance due to the GPU friendly graph representation.

5.7.4 Sensitivity Analysis of WolfGraph

In this section, we examine the sensitivity of WolfGraph across different graph characteristics. We first examine the performance of WolfGraph under different graph size. The synthetic graphs are created with SNAP graph library [81] and the RMAT [18] model are used to ensure the generated graphs are scale free and resemble the characteristics of real-world graphs.

We conduct this experiment with BFS algorithm on WolfGraph with ten synthetically created RMAT graphs across the range of different sizes and sparsities. The experiment result is shown in Figure 5.16. In this experiment, we only record the kernel computation time.

Figure 5.16 shows the execution time when increasing the number of edges and vertices in the graph. In this experiment, the degree of the graphs is fixed to 16. As can be seen from the figure, increasing the graph size will cause longer computation time, this is because as the graph size increase, it increases the amount of data that need to be fetched from and written back to global memory. Therefore, the number of conflicts writes to both shared and global memory is increased as well, which will reduce the performance.

We then examine how block size can affect the performance of WolfGraph. We conduct two experiments with BFS algorithm on two real world graphs *amazon0601* and *hollywood-2011*. In these experiments, we change the block size from 32 to 1024, the breakdown execution time of these experiments are listed in Table 5.4 and 5.5.

As shown in these results, changing the block size does not affect the pre-processing time, data transfer time and time spend on building CEL, this is

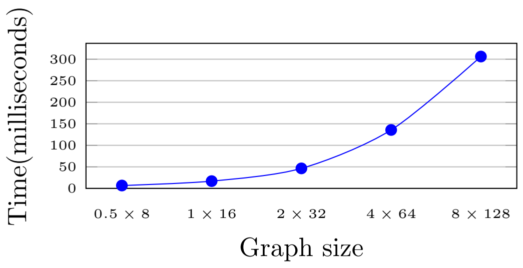


Figure 5.16: Execution time of WolfGraph when Changing graph size with graph degree equal to 16. A $x \times y$ graph has around x million vertices and y million edges

Table 5.4: The break down processing time with different thread block size on amazon0601 measured in miliseconds

Block Size	Preprocessing	Data Transfer	Computation
32	626.6	4.45	14.06
64	625.29	4.45	8.37
128	624.47	4.45	7.36
256	623.66	4.43	7.48
512	623.48	4.49	7.91
1024	625.36	4.44	8.31

Table 5.5: The break down processing time with different thread block size on Hollywood-2011 measured in seconds

Block Size	Preprocessing	Build CEL	Data Transfer	Computation
32	33.42	0.74	0.31	0.36
64	33.35	0.73	0.31	0.21
128	33.43	0.73	0.32	0.14
256	33.49	0.74	0.33	0.15
512	33.41	0.73	0.32	0.16
1024	33.23	0.73	0.32	0.18

because the CPU performs these operations. On the other hand, the block size affects the computation time. In both graphs, the computation time first decreases as we increase the block size, this is because the block size 32 and 64 cannot fully utilise the GPU's computation power. For example, on GTX 780Ti, the maximum number of thread blocks supported by one SM is 16 and the maximum number of threads that the SM can execute is 2048. With 32 thread per block, the GPU can launch 16 thread blocks per SM. However, the total threads number per SM is $32 * 16 = 512$, which is only a quarter of the maximum computation power a SM can provide.

When block size is larger than 128, the computation time starts increasing, this is because increasing block size will also increase the edge block size. Therefore, the number of conflicts within a thread block will increase, which will decrease the performance.

5.8 Summary

In this chapter, we introduce a new graph framework called WolfGraph for processing large graphs that are unable to fit into memory. The core design principle of WolfGraph is to reduce the pre-processing time. To achieve this objective, the framework uses edge-centric computation model, and edge list is used to represent the graph in memory. Through the experiments, we argue that in modern parallel graph processing systems, the performance bottleneck is reading data from hard disk and pre-processing the data to meet the system's requirement. We also demonstrated that reducing the pre-processing time can improve the overall execution time significantly.

Chapter 6

Conclusions and Further Work

The work described in this thesis has been concentrated on developing co-scheduling strategies in multicore systems. Multicore processors have now become a mainstream product in the CPU industry. In a multicore processor, running multiple applications on different cores could cause performance degradation. One key technique applied in reducing performance degradation in a multicore system is contention-aware co-scheduling. In this thesis, we extend existing scheduling model to find the optimal solution when both serial and parallel jobs exist in the system.

We construct a graph to represent the co-scheduling problem and convert the problem of finding good co-scheduling solutions to the problem of finding the shortest valid path in the graph. A set of algorithms and optimization techniques are proposed to find either optimal or near-optimal co-scheduling solutions. We also proposed WolfPath, a graph processing framework that uses GPU to accelerate the graph processing algorithms. In order to reduce pre-processing time in WolfPath, we developed WolfGraph, a general purpose GPU-based graph processing framework that aims to minimize the graph pre-processing.

The key contributions of this thesis are summarised in the first three sections of this chapter. Discussion and Further works are then presented in section 6.4 and 6.5.

6.1 Developing Graph-based Methods to Find Optimal or Near-optimal Co-Scheduling solutions

In chapter 3, a graph-based method is developed to find the optimal co-scheduling solution for serial jobs, and then extended to incorporate parallel jobs. A number of optimization measures are developed to accelerate the solving process. It has been shown that the A*-search algorithm can effectively avoid the unnecessary searches when finding the optimal solution. In this thesis, an A*-search-based algorithm is developed to combine the ability of the A*-search algorithm and the proposed optimization measures in terms of accelerating the solving process. Further, a heuristic method, called heuristic A*-search algorithm, is developed to find the near-optimal solutions more efficiently. Finally, a flexible approximation technique is proposed so that we can control the scheduling efficiency by setting the requirement for the solution quality. We conducted the experiments to evaluate the effectiveness of the proposed co-scheduling algorithms. The results show that i) the proposed algorithms can find the optimal co-scheduling solution for both serial and parallel jobs, ii) the proposed optimization measures can significantly increase the scheduling efficiency, and iii) the proposed approximation technique is effective in the sense that it is able to balance the scheduling efficiency and the solution quality.

6.2 WolfPath: Accelerating the graphs with layered structure by GPU

In chapter 4, we investigate the use of GPU to accelerate our co-scheduling algorithms. Most GPU-based parallel graph processing frameworks employ iterative processing model. However, by benchmarking the state-of-the-art GPU-based graph processing frameworks, we observed that the performance of iterative traversing-based graph algorithms on GPU is limited by the frequent data exchange between host and GPU. In order to tackle the problem, we develop a GPU-based graph framework called WolfPath to accelerate the processing of iterative traversing-based graph processing algorithms. In WolfPath, the iterative process is guided by the graph diameter to eliminate the frequent data exchange between host and GPU. To accomplish this goal, WolfPath proposes a data structure called Layered Edge list to represent the graph, from which the graph diameter is known before the start of graph processing. In order to enhance the applicability of our WolfPath framework, a graph preprocessing algorithm is also developed in this work to convert any graph into the format of the Layered Edge list. We conducted extensive experiments to verify the effectiveness of WolfPath. The experimental results show that WolfPath achieves significant speedup over the state-of-the-art GPU-based in-memory and out-of-memory graph processing frameworks.

6.3 WolfGraph: A General Purpose GPU-based Large-Scale Graph Processing Framework

Similar to WolfPath, most existing graph processing frameworks require graph pre-processing before the graph processing algorithm can be applied. The idea is that although it takes the time to pre-process a graph, the execution of the graph processing algorithm will take much less time than without pre-processing and therefore the overall processing time will be reduced significantly. However, in the state-of-the-art GPU-based graph processing systems, the time spent in reading the graph from hard disks to memory and in constructing the data structure in memory constitute a big proportion of the total processing time for a large graph. Reducing this pre-processing time will improve the overall performance of graph processing frameworks. Based on this observation, we proposed WolfGraph in chapter 5. The main novelty and contribution of WolfGraph are designing a GPU-based graph processing framework that endures minimal pre-processing. The WolfGraph adopts the edge-centric processing model, which iterates over the edges rather than over vertices. The data structure and graph partition are carefully crafted in WolfGraph so as to reduce the pre-processing time and to avoid the irregular access to graph edges and allow the fully coalesced memory accesses. Moreover, WolfGraph fully utilizes the GPU power by processing all edges in parallel. We also developed a new method, called Concatenated Edge List (CEL) to process a graph that is bigger than the global memory of GPU or host memory. Comparing with the existing GPU-based framework, Wolf Graph can achieve similar execution time while minimal pre-processing is carried out. Therefore, WolfGraph reduces the overall processing time significantly.

6.4 Discussion

The co-scheduling work presented in this thesis can benefit the practice of co-scheduling in two ways. First, it can be used to evaluate various co-scheduling systems. Most current evaluation of a co-scheduling system compares the system only to random schedulers. But in the design of a practical co-scheduling system, it is important to know the room left for improvement. The optimal solution provides the engineer with a unique insight into how much performance can be extracted if the system were best tuned. Additionally, knowing the gap between current and optimal performance can help the scheduler designers to weight the trade-offs between efficiency and quality. Through the techniques presented in this work, the optimal schedules can be either attained precisely or approximated accurately. Second, proactive co-scheduling may directly benefit from the algorithms proposed in this work. With accurate predictions, the proactive schedulers may benefit from this work by applying the co-scheduling algorithms to determine the optimal or near-optimal schedules.

In this thesis, we assume that all co-run performance is given. This assumption can be met in following ways: First, when using our algorithm in evaluating other co-scheduling systems. This is because the evaluation process can typically afford the time for collecting all co-run performance. Second, proactive co-schedulers can predict co-run performance. Although obtaining co-run performance is a time consuming process, but we argue that the time for brute-forth search for optimal schedules will dominate for large-size problems, because the search time grows exponentially. On the other hand, the number of co-runs grows polynomially as the number of jobs increases.

Many programs contain multiple phases during the execution, and each phase may have different performance (e.g., MapReduce program). In this work, we assumed that the program execution contains only one phase. But when using our algorithms for evaluation purpose, the user can simply use one phase of the programs and consider the co-run parts of their executions only.

The graph processing frameworks presented in this thesis represent two different research directions. WolfPath focuses on improving the computation performance through data pre-processing. On the other hand, WolfGraph tends to improve the overall execution time by reducing the pre-processing time.

As demonstrated in this thesis and many other researches [76] [146] [90], pre-processing can improve the computation performance significantly. This is because through pre-processing, the system designer can enhance the data locality, which can improve the memory access performance. During the pre-processing, useful information can be extracted from the data to optimise computation performance. In addition, in distributed system, pre-processing can help reduce the communication cost. However, the pre-processing will introduce the extra overhead to the system. As shown in chapter 4, WolfPath achieved 100X speedup over other state-of-the-art graph processing systems through pre-processing, but other systems beat the WolfPath in overall execution time. This is because in WolfPath, pre-processing takes much longer than other systems.

Therefore, extra care needs to be taken when using pre-processing. In fact, we argue that the pre-processing should only be used when it can improve the overall performance except following two cases: First, when the data only need to be pre-processed once. In such situation, the pre-processing is affordable even it is time consuming, because once it is done, all the subsequently computation can benefit from it in the future. Second, when the computation cannot proceed without pre-processing.

The WolfGraph is designed based on the above principle. In WolfGraph, the pre-processing is limited to read and store the input graph from hard disk to memory. On the other hand, WolfGraph replies on the carefully designed data structure, optimised parallel processing model specific to GPU and take advantage of GPU hardware features to ensure its computation performance is competitive with other state-of-the-art graph processing systems.

The biggest advantage of the second strategy is that it can guarantee the sequential access to the hard disk, and fully utilise the storage bandwidth. However, such strategy is only suitable for shared memory system or distributed system with low communication cost. This is because the data locality cannot be guaranteed in this strategy. Therefore, it requires frequent data exchange/synchronization to ensure correctness of the result. On shared memory system, the overhead of data exchange/synchronization is relatively low and can be optimised through the carefully designed data structure. For example, X-Stream [109] uses shuffle buffer to reduce the communication cost in shared memory systems. On the other hand, in distributed systems, the communication cost is bounded by network bandwidth, and is much slower than memory access speed. To our knowledge, Chaos [110] is the only distributed graph processing system that adapts the second strategy, and it is designed for the small cluster. This is because the network bandwidth in the small cluster is greater than hard disk's I/O bandwidth.

6.5 Further Work

The co-scheduling strategies developed in chapter 3 assume that the underlying system is homogeneous. However, nowadays the heterogeneous architectures that bring together CPUs and multiple domain-specific GPUs and FPGAs provide dramatic speedup for many applications. In such systems, the applications not only compete for multicore resources, but also compete for accelerators. For example, when multiple GPU accelerated applications are running on the same node, the GPU is time shared among these applications, because GPU cannot execute the kernels from different context concurrently. In such situation, the challenge lies in utilizing these heterogeneous processors to optimize overall application performance by minimizing workload completion time [8]. In the future, we plan to explore the possibility to extend our models to tackle such systems. Also, given the popularity of cloud systems, we plan to extend our co-scheduling methods to solve the optimal mapping of virtual machines (VM) on physical machines. The main extension is to allow the VM migrations between physical machines.

The graph processing systems can be extended in the following ways: first, although being accelerated with GPU, the processing capacity of a single machine is still limited. Some problems require the processing power or storage capacity of a cluster. The intuitive method to process a graph in a cluster is to assign an edge block to a node in the cluster. After one iteration, the synchronisation is performed among all cluster nodes. The problem for such design is that the amount of data transferred over the network is very large. Although using advanced graph partitioning methods can solve this problem, this also leads to long pre-processing time, which contradicts to the design philosophy of Wolf-Graph. Hence, we plan to investigate how to reduce the network communication without introducing too much pre-processing overhead.

It has been shown that the graphs in real-world applications exhibit significant topological differences [82]. The graph topology affects the performance of specific GPU implementations. This feature of graph data makes it difficult to design a GPU implementation that is optimal on a large variety of datasets. Therefore, we plan to explore the possibility of using an adaptive approach that takes into account the topological characteristics of the graphs to dynamically select the most suitable graph representation, graph partition, computation model and so on.

Bibliography

[1] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/. URL `https://perf.wiki.kernel.org/index.php/Main_Page`. Accessed: 18.04.2014.

[2] GLPK (GNU linear programming kit), 2006. URL `http://www.gnu.org/software/glpk`.

[3] IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/, Last 2010.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010. ISSN 0001-0782. doi: 10.1145/1721654.1721672. URL `http://doi.acm.org/10.1145/1721654.1721672`.

[5] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[7] M. Banikazemi, D. Poff, and B. Abali. Pam: A novel performance/power

aware meta-scheduler for multi-core systems. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 39:1–39:12, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9. URL http://dl.acm.org/citation.cfm?id=1413370.1413410.

[8] M. E. Belviranli, L. N. Bhuyan, and R. Gupta. A dynamic self-scheduling scheme for heterogeneous multiprocessor architectures. *ACM Trans. Archit. Code Optim.*, 9(4):57:1–57:20, Jan. 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400716. URL http://doi.acm.org/10.1145/2400682.2400716.

[9] K. Beyls and E. H. D'Hollander. Refactoring for data locality. *Computer*, 42(2):62–71, Feb. 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.57. URL http://dx.doi.org/10.1109/MC.2009.57.

[10] A. Bialecki, M. Cafarella, D. Cutting, and O. OMALLEY. Hadoop: a framework for running applications on large clusters built of commodity hardware. *Wiki at http://lucene. apache. org/hadoop*, 11, 2005.

[11] S. Blagodurov, S. Zhuravlev, and A. Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4):8:1–8:45, Dec. 2010. ISSN 0734-2071. doi: 10.1145/1880018.1880019. URL http://doi.acm.org/10.1145/1880018.1880019.

[12] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 557–558, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. doi: 10.1145/1854273.1854350. URL http://doi.acm.org/10.1145/1854273.1854350.

[13] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595–601, Manhattan, USA, 2004. ACM Press.

[14] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In S. Srinivasan, K. Ramamritham, A. Kumar, M. P. Ravindra, E. Bertino, and R. Kumar, editors, *Proceedings of the 20th international conference on World Wide Web*, pages 587–596. ACM Press, 2011.

[15] F. Busato and N. Bombieri. Bfs-4k: An efficient implementation of bfs for kepler gpu architectures. *IEEE Transactions on Parallel and Distributed Systems*, 26(7):1826–1838, July 2015. ISSN 1045-9219. doi: 10.1109/TPDS.2014.2330597.

[16] C. Cascaval, L. D. Rose, D. A. Padua, and D. A. Reed. Compile-time based performance prediction. In *Proceedings of the 12th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '99, pages 365–379, London, UK, UK, 2000. Springer-Verlag. ISBN 3-540-67858-1. URL `http://dl.acm.org/citation.cfm?id=645677.663790`.

[17] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008. ISSN 2150-8097. doi: 10.14778/1454159.1454166. URL `http://dx.doi.org/10.14778/1454159.1454166`.

[18] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. doi: 10.1137/1.9781611972740.43. URL `http://epubs.siam.org/doi/abs/10.1137/1.9781611972740.43`.

[19] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 340–351, Washington, DC, USA, 2005. IEEE

Computer Society. ISBN 0-7695-2275-0. doi: 10.1109/HPCA.2005.27. URL `http://dx.doi.org/10.1109/HPCA.2005.27`.

[20] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, pages 242–252, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-768-1. doi: 10.1145/1274971.1275005. URL `http://doi.acm.org/10.1145/1274971.1275005`.

[21] X. E. Chen and T. Aamodt. Modeling cache contention and throughput of multiprogrammed manycore processors. *IEEE Trans. Comput.*, 61(7): 913–927, July 2012. ISSN 0018-9340. doi: 10.1109/TC.2011.141. URL `http://dx.doi.org/10.1109/TC.2011.141`.

[22] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. S. Lui, and C. He. Venus: Vertex-centric streamlined graph computation on a single pc. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1131–1142, April 2015. doi: 10.1109/ICDE.2015.7113362.

[23] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, Aug. 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824077. URL `http://dx.doi.org/10.14778/2824032.2824077`.

[24] C.-L. Chou and R. Marculescu. Contention-aware application mapping for network-on-chip communication architectures. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 164–169, Oct 2008. doi: 10.1109/ICCD.2008.4751856.

[25] T. Clinkenbeard and A. Nica. Job scheduling with minimizing data communication costs. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages

2071–2072, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2764943. URL `http://doi.acm.org/10.1145/2723372.2764943`.

[26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

[27] N. Corporation. *NVIDIA CUDA C Programming Guide*. NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, version 7.5 edition, 2015. URL `http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf`.

[28] S. Darbha and D. P. Agrawal. Optimal scheduling algorithm for distributed-memory machines. *IEEE Trans. Parallel Distrib. Syst.*, 9 (1):87–95, Jan. 1998. ISSN 1045-9219. doi: 10.1109/71.655248. URL `http://dx.doi.org/10.1109/71.655248`.

[29] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel gpu methods for single-source shortest paths. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 349–359, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-3800-1. doi: 10.1109/IPDPS.2014.45. URL `http://dx.doi.org/10.1109/IPDPS.2014.45`.

[30] T. de Gooijer and K. E. Harper. Experiences with modeling memory contention for multi-core industrial real-time systems. In *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '14, pages 43–52, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2576-9. doi: 10.1145/2602576.2602584. URL `http://doi.acm.org/10.1145/2602576.2602584`.

[31] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-

0782. doi: 10.1145/1327452.1327492. URL http://doi.acm.org/10.1145/1327452.1327492.

[32] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541941. URL http://doi.acm.org/10.1145/2541940.2541941.

[33] M. Diener, E. H. M. Cruz, and P. O. A. Navaux. Locality vs. balance: Exploring data mapping policies on numa systems. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 9–16, March 2015. doi: 10.1109/PDP.2015.11.

[34] D. Eklov, D. Black-Schaffer, and E. Hagersten. Fast modeling of shared caches in multicore systems. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 147–157, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0241-8. doi: 10.1145/1944862.1944885. URL http://doi.acm.org/10.1145/1944862.1944885.

[35] A. Fedorova, M. Seltzer, and M. Smith. Cache-fair thread scheduling for multicore processors. *Division of Engineering and Applied Sciences, Harvard University, Tech. Rep. TR-17-06*, 2006.

[36] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 25–38, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2944-5. doi: 10.1109/PACT.2007.40. URL http://dx.doi.org/10.1109/PACT.2007.40.

[37] A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for

shared resources on multicore processors. *Commun. ACM*, 53(2):49–57, Feb. 2010. ISSN 0001-0782. doi: 10.1145/1646353.1646371. URL `http://doi.acm.org/10.1145/1646353.1646371`.

[38] J. Feliu, S. Petit, J. Sahuquillo, and J. Duato. Cache-hierarchy contention-aware scheduling in cmps. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):581–590, March 2014. ISSN 1045-9219. doi: 10.1109/TPDS.2013.61.

[39] J. Forrest and R. Lougee-Heimer. *CBC User Guide*, chapter Chapter 10, pages 257–277. doi: 10.1287/educ.1053.0020. URL `http://pubsonline.informs.org/doi/abs/10.1287/educ.1053.0020`.

[40] Z. Fu, M. Personick, and B. Thompson. Mapgraph: A high level api for fast development of high performance graph analytics on gpus. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, pages 2:1–2:6, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2982-8. doi: 10.1145/2621934.2621936. URL `http://doi.acm.org/10.1145/2621934.2621936`.

[41] G. Gamrath, T. Fischer, T. Gally, A. M. Gleixner, G. Hendel, T. Koch, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, S. Vigerske, D. Weninger, M. Winkler, J. T. Witt, and J. Witzig. The scip optimization suite 3.2. Technical Report 15-60, ZIB, Takustr.7, 14195 Berlin, 2016.

[42] A. Gharaibeh, L. Beltrão Costa, E. Santos-Neto, and M. Ripeanu. A yoke of oxen and a thousand chickens for heavy lifting graph processing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 345–354, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370866. URL `http://doi.acm.org/10.1145/2370816.2370866`.

[43] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph:

Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL `http://dl.acm.org/citation.cfm?id=2387880.2387883`.

[44] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 599–613, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL `http://dl.acm.org/citation.cfm?id=2685048.2685096`.

[45] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. How well do graph-processing platforms perform? an empirical performance evaluation and analysis. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 395–404, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-3800-1. doi: 10.1109/IPDPS.2014.49. URL `http://dx.doi.org/10.1109/IPDPS.2014.49`.

[46] Y. Guo, A. L. Varbanescu, A. Iosup, and D. Epema. An empirical performance evaluation of gpu-enabled graph-processing systems. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 423–432, May 2015. doi: 10.1109/CCGrid.2015.20.

[47] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, pages 342–362, New York, NY, USA, 2006. Springer-Verlag New York, Inc. URL `http://dl.acm.org/citation.cfm?id=1515984.1516011`.

[48] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1:1–1:14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592799.

[49] W.-S. Han, S. Lee, K. Park, J.-H. Lee, M.-S. Kim, J. Kim, and H. Yu. Turbograph: A fast parallel graph engine handling billion-scale graphs in a single pc. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 77–85, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2174-7. doi: 10.1145/2487575.2487581. URL `http://doi.acm.org/10.1145/2487575.2487581`.

[50] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-77219-7, 978-3-540-77219-4. URL `http://dl.acm.org/citation.cfm?id=1782174.1782200`.

[51] A.-H. Haritatos, G. Goumas, N. Anastopoulos, K. Nikas, K. Kourtis, and N. Koziris. Lca: A memory link and cache-aware co-scheduling approach for cmps. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 469–470, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2809-8. doi: 10.1145/2628071.2628123. URL `http://doi.acm.org/10.1145/2628071.2628123`.

[52] L. He, H. Zhu, and S. A. Jarvis. Developing graph-based co-scheduling algorithms on multicore computers. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1617–1632, June 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2468223.

[53] B. Hendrickson and J. W. Berry. Graph analysis with high-performance

computing. *Computing in Science Engineering*, 10(2):14–19, March 2008. ISSN 1521-9615. doi: 10.1109/MCSE.2008.56.

[54] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006. ISSN 0163-5964. doi: 10.1145/1186736.1186737. URL `http://doi.acm.org/10.1145/1186736.1186737`.

[55] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1972457.1972488`.

[56] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. doi: 10.1145/1941553.1941590. URL `http://doi.acm.org/10.1145/1941553.1941590`.

[57] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 343–354, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2126-6. doi: 10.1109/MICRO.2004.4. URL `http://dx.doi.org/10.1109/MICRO.2004.4`.

[58] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 208–219, New York,

NY, USA, 2008. ACM. ISBN 978-1-60558-282-5. doi: 10.1145/1454115.1454145. URL http://doi.acm.org/10.1145/1454115.1454145.

[59] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 220–229, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-282-5. doi: 10.1145/1454115.1454146. URL http://doi.acm.org/10.1145/1454115.1454146.

[60] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC'10, pages 201–215, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11514-4, 978-3-642-11514-1. doi: 10.1007/978-3-642-11515-8_16. URL http://dx.doi.org/10.1007/978-3-642-11515-8_16.

[61] Y. Jiang, K. Tian, X. Shen, J. Zhang, J. Chen, and R. Tripathi. The complexity of optimal job co-scheduling on chip multiprocessors and heuristics-based solutions. *IEEE Trans. Parallel Distrib. Syst.*, 22(7): 1192–1205, July 2011. ISSN 1045-9219. doi: 10.1109/TPDS.2010.193. URL http://dx.doi.org/10.1109/TPDS.2010.193.

[62] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3895-2. doi: 10.1109/ICDM.2009.14. URL http://dx.doi.org/10.1109/ICDM.2009.14.

[63] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *ACM Trans. Knowl. Discov. Data*, 5

(2):8:1–8:24, Feb. 2011. ISSN 1556-4681. doi: 10.1145/1921632.1921634. URL `http://doi.acm.org/10.1145/1921632.1921634`.

[64] A. Khan and S. Elnikety. Systems for big-graphs. *Proc. VLDB Endow.*, 7(13):1709–1710, Aug. 2014. ISSN 2150-8097. doi: 10.14778/2733004.2733067. URL `http://dx.doi.org/10.14778/2733004.2733067`.

[65] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 239–252, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2749-7. doi: 10.1145/2600212.2600227. URL `http://doi.acm.org/10.1145/2600212.2600227`.

[66] F. Khorasani, R. Gupta, and L. N. Bhuyan. Scalable simd-efficient graph processing on gpus. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 39–50, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-9524-3. doi: 10.1109/PACT.2015.15.

[67] H.-S. Kim, I. El Hajj, J. Stratton, S. Lumetta, and W.-M. Hwu. Locality-centric thread scheduling for bulk-synchronous programming models on cpu architectures. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 257–268, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8161-8. URL `http://dl.acm.org/citation.cfm?id=2738600.2738632`.

[68] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 111–122, Washington, DC, USA, 2004. IEEE Computer Society.

ISBN 0-7695-2229-7. doi: 10.1109/PACT.2004.15. URL http://dx.doi.org/10.1109/PACT.2004.15.

[69] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28 (3):54–66, May 2008. ISSN 0272-1732. doi: 10.1109/MM.2008.48. URL http://dx.doi.org/10.1109/MM.2008.48.

[70] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28 (3):54–66, May 2008. ISSN 0272-1732. doi: 10.1109/MM.2008.48. URL http://dx.doi.org/10.1109/MM.2008.48.

[71] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. An analysis of performance interference effects in virtual environments. In *2007 IEEE International Symposium on Performance Analysis of Systems Software*, pages 200–209, April 2007. doi: 10.1109/ISPASS.2007.363750.

[72] E. Kontoghiorghes. *Handbook of Parallel Computing and Statistics.* Chapman & Hall/CRC, 2005. ISBN 082474067X.

[73] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 591–600, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772751. URL http://doi.acm.org/10.1145/1772690.1772751.

[74] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.*, 59(3):381–422, Dec. 1999. ISSN 0743-7315. doi: 10.1006/jpdc.1999.1578. URL http://dx.doi.org/10.1006/jpdc.1999.1578.

[75] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–

471, Dec. 1999. ISSN 0360-0300. doi: 10.1145/344588.344618. URL http://doi.acm.org/10.1145/344588.344618.

[76] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387884.

[77] L. L. N. Laboratory. *MPI Pi Reduce*, 2016 (Accessed 22 Dec. 2016). URL https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_pi_reduce.c.

[78] S. Lattanzi, B. Moseley, S. Suri, and S. Vassilvitskii. Filtering: A method for solving graph problems in mapreduce. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 85–94, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0743-7. doi: 10.1145/1989493.1989505. URL http://doi.acm.org/10.1145/1989493.1989505.

[79] M. Lee and K. Schwan. Region scheduling: Efficiently using the cache architectures via page-level affinity. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 451–462, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2151023. URL http://doi.acm.org/10.1145/2150976.2151023.

[80] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[81] J. Leskovec and R. Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Trans. Intell. Syst. Technol.*, 8(1):1:1–1:20, July 2016. ISSN 2157-6904. doi: 10.1145/2898361. URL http://doi.acm.org/10.1145/2898361.

[82] D. Li and M. Becchi. Deploying graph algorithms on gpus: An adaptive solution. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, pages 1013–1024, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4971-2. doi: 10.1109/IPDPS.2013.101. URL `http://dx.doi.org/10.1109/IPDPS.2013.101`.

[83] Y. Liang and T. Mitra. Cache modeling in probabilistic execution time analysis. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 319–324, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-115-6. doi: 10.1145/1391469.1391551. URL `http://doi.acm.org/10.1145/1391469.1391551`.

[84] G. H. Loh. 3d-stacked memory architectures for multi-core processors. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 453–464, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3174-8. doi: 10.1109/ISCA.2008.15. URL `http://dx.doi.org/10.1109/ISCA.2008.15`.

[85] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010. URL `http://arxiv.org/abs/1006.4990`.

[86] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012. ISSN 2150-8097. doi: 10.14778/2212351.2212354. URL `http://dx.doi.org/10.14778/2212351.2212354`.

[87] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proc. VLDB Endow.*, 8 (3):281–292, Nov. 2014. ISSN 2150-8097. doi: 10.14778/2735508.2735517. URL `http://dx.doi.org/10.14778/2735508.2735517`.

[88] A. LUMSDAINE, D. GREGOR, B. HENDRICKSON, and J. BERRY. Challenges in parallel graph processing. *Parallel Processing Letters*, 17 (01):5–20, 2007. doi: 10.1142/S0129626407002843. URL `http://www.worldscientific.com/doi/abs/10.1142/S0129626407002843`.

[89] P. R. Luszczek, D. H. Bailey, J. J. Dongarra, J. Kepner, R. F. Lucas, R. Rabenseifner, and D. Takahashi. The hpc challenge (hpcc) benchmark suite. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188677. URL `http://doi.acm.org/10.1145/1188455.1188677`.

[90] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. In *2015 IEEE 31st International Conference on Data Engineering*, pages 363–374, April 2015. doi: 10.1109/ICDE.2015.7113298.

[91] N. Maclaren. Mandelbrot set, 2016 (Accessed 22 Dec. 2016). URL `http://people.ds.cam.ac.uk/nmm1/MPI/Programs/mandelbrot.c`.

[92] Z. Majo and T. R. Gross. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. *SIGPLAN Not.*, 46(11):11–20, June 2011. ISSN 0362-1340. doi: 10.1145/2076022.1993481. URL `http://doi.acm.org/10.1145/2076022.1993481`.

[93] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0032-2. doi: 10.1145/1807167.1807184. URL `http://doi.acm.org/10.1145/1807167.1807184`.

[94] E. P. Markatos and T. J. LeBlanc. Load balancing vs. locality management in shared-memory multiprocessors. Technical report, Rochester, NY, USA, 1991.

[95] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 248–259, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1053-6. doi: 10.1145/2155620.2155650. URL `http://doi.acm.org/10.1145/2155620.2155650`.

[96] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970. ISSN 0018-8670. doi: 10.1147/sj.92.0078. URL `http://dx.doi.org/10.1147/sj.92.0078`.

[97] A. Merkel and F. Bellosa. Task activity vectors: A new metric for temperature-aware scheduling. *SIGOPS Oper. Syst. Rev.*, 42(4):1–12, Apr. 2008. ISSN 0163-5980. doi: 10.1145/1357010.1352594. URL `http://doi.acm.org/10.1145/1357010.1352594`.

[98] A. Merkel, J. Stoess, and F. Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 153–166, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. doi: 10.1145/1755913.1755930. URL `http://doi.acm.org/10.1145/1755913.1755930`.

[99] D. Merrill, M. Garland, and A. Grimshaw. Scalable gpu graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 117–128, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1160-1. doi: 10.1145/2145816.2145832. URL `http://doi.acm.org/10.1145/2145816.2145832`.

[100] M. Michael, J. E. Moreira, D. Shiloach, and R. W. Wisniewski. Scale-up

x scale-out: A case study using nutch/lucene. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007. doi: 10.1109/IPDPS.2007.370631.

[101] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 63–74, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3174-8. doi: 10.1109/ISCA.2008.7. URL `http://dx.doi.org/10.1109/ISCA.2008.7`.

[102] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 57–68, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250671. URL `http://doi.acm.org/10.1145/1250662.1250671`.

[103] A. Nisar, W.-k. Liao, and A. Choudhary. Scaling parallel i/o performance through i/o delegate and caching system. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 9:1–9:12, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9. URL `http://dl.acm.org/citation.cfm?id=1413370.1413380`.

[104] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2008. ISBN 0123744938, 9780123744937.

[105] Y. Perez, R. Sosič, A. Banerjee, R. Puttagunta, M. Raison, P. Shah, and J. Leskovec. Ringo: Interactive graph analytics on big-memory machines. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1105–1110, New York,

NY, USA, 2015. ACM. ISBN 978-1-4503-2758-9. doi: 10.1145/2723372.2735369.

[106] S. J. Plimpton and K. D. Devine. Mapreduce in {MPI} for large-scale graph algorithms. *Parallel Computing*, 37(9):610 – 632, 2011. ISSN 0167-8191. Emerging Programming Paradigms for Large-Scale Scientific Computing.

[107] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. *SIGARCH Comput. Archit. News*, 34(2):167–178, 2006. ISSN 0163-5964. doi: 10.1145/1150019.1136501.

[108] J. S. Rosenthal. Parallel computing and monte carlo algorithms. *Far east journal of theoretical statistics*, 4(2):207–236, 2000.

[109] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522740. URL `http://doi.acm.org/10.1145/2517349.2522740`.

[110] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3834-9. doi: 10.1145/2815400.2815408.

[111] A. Roytman, A. Kansal, S. Govindan, J. Liu, and S. Nath. Pacman: Performance aware virtual machine consolidation. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 83–94, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-02-7. URL `https://www.usenix.org/conference/icac13/technical-sessions/presentation/roytman`.

[112] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345220. URL `http://doi.acm.org/10.1145/1345206.1345220`.

[113] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1921-8. doi: 10.1145/2484838.2484843. URL `http://doi.acm.org/10.1145/2484838.2484843`.

[114] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 351–364, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465386. URL `http://doi.acm.org/10.1145/2465351.2465386`.

[115] D. Sengupta, S. L. Song, K. Agarwal, and K. Schwan. Graphreduce: Processing large-scale graphs on accelerator-based systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 28:1–28:12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3723-6. doi: 10.1145/2807591.2807655. URL `http://doi.acm.org/10.1145/2807591.2807655`.

[116] B. Shao, H. Wang, and Y. Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2037-

5. doi: 10.1145/2463676.2467799. URL http://doi.acm.org/10.1145/2463676.2467799.

[117] K. Shirahata, H. Sato, T. Suzumura, and S. Matsuoka. A scalable implementation of a mapreduce-based graph processing algorithm for large-scale heterogeneous supercomputers. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 277–284, May 2013. doi: 10.1109/CCGrid.2013.85.

[118] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. doi: 10.1145/2442516.2442530. URL http://doi.acm.org/10.1145/2442516.2442530.

[119] J. Shun, L. Dhulipala, and G. E. Blelloch. Smaller and faster: Parallel processing of compressed graphs with ligra+. In *Proceedings of the 2015 Data Compression Conference*, DCC '15, pages 403–412, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8430-5. doi: 10.1109/DCC.2015.8. URL http://dx.doi.org/10.1109/DCC.2015.8.

[120] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive set pinning: Managing shared caches in chip multiprocessors. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 135–144, New York, NY, USA, 2008. ACM. doi: 10.1145/1346281.1346299.

[121] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive set pinning: Managing shared caches in chip multiprocessors. *SIGARCH Comput. Archit. News*, 36(1):135–144, mar 2008. ISSN 0163-5964. doi: 10.1145/1353534.1346299. URL http://doi.acm.org/10.1145/1353534.1346299.

[122] S. Srikantaiah, R. Das, A. K. Mishra, C. R. Das, and M. Kandemir. A

case for integrated processor-cache partitioning in chip multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 6:1–6:12, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-744-8. doi: 10.1145/1654059.1654066. URL `http://doi.acm.org/10.1145/1654059.1654066`.

[123] G. Staples. Torque resource manager. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. ISBN 0-7695-2700-0. doi: 10.1145/1188455.1188464. URL `http://doi.acm.org/10.1145/1188455.1188464`.

[124] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3): 66–73, May 2010. ISSN 0740-7475. doi: 10.1109/MCSE.2010.69. URL `http://dx.doi.org/10.1109/MCSE.2010.69`.

[125] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 117–, Washington, DC, USA, 2002. IEEE Computer Society. URL `http://dl.acm.org/citation.cfm?id=874076.876484`.

[126] S. Sujan and R. Kanniga Devi. *An Efficient Task Scheduling Scheme in Cloud Computing Using Graph Theory*, pages 655–662. Springer India, New Delhi, 2016. ISBN 978-81-322-2674-1. doi: 10.1007/978-81-322-2674-1_62. URL `http://dx.doi.org/10.1007/978-81-322-2674-1_62`.

[127] Q. Tang, T. Basten, M. Geilen, S. Stuijk, and J.-B. Wei. Task-fifo co-scheduling of streaming applications on mpsocs with predictable memory hierarchy. In *Proceedings of the 2015 15th International Conference on Application of Concurrency to System Design*, ACSD '15, pages 90–99, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4673-

7882-6. doi: 10.1109/ACSD.2015.9. URL http://dx.doi.org/10.1109/ACSD.2015.9.

[128] K. Tian, Y. Jiang, and X. Shen. A study on optimally co-scheduling jobs of different lengths on chip multiprocessors. In *Proceedings of the 6th ACM Conference on Computing Frontiers*, CF '09, pages 41–50, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-413-3. doi: 10.1145/1531743.1531752. URL http://doi.acm.org/10.1145/1531743.1531752.

[129] M. Tolubaeva, Y. Yan, and B. Chapman. Predicting cache contention for multithread applications at compile time. In *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, pages 624–631, May 2014. doi: 10.1109/IPDPSW.2014.73.

[130] H. Topcuouglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, Mar. 2002. ISSN 1045-9219. doi: 10.1109/71.993206. URL http://dx.doi.org/10.1109/71.993206.

[131] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990. ISSN 0001-0782. doi: 10.1145/79173.79181. URL http://doi.acm.org/10.1145/79173.79181.

[132] J. Wang, N. Abu-Ghazaleh, and D. Ponomarev. Controlled contention: Balancing contention and reservation in multicore application scheduling. In *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '15, pages 946–955, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8649-1. doi: 10.1109/IPDPS.2015.62. URL http://dx.doi.org/10.1109/IPDPS.2015.62.

[133] K. Wang, G. Xu, Z. Su, and Y. D. Liu. Graphq: Graph query processing with abstraction refinement—scalable and programmable analytics over very large graphs on a single pc. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 387–401,

Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL `https://www.usenix.org/conference/atc15/technical-session/presentation/wang-kai`.

[134] Y. Wang, Y. Cui, P. Tao, H. Fan, Y. Chen, and Y. Shi. Reducing shared cache contention by scheduling order adjustment on commodity multicores. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 984–992, May 2011. doi: 10.1109/IPDPS.2011.248.

[135] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 265–266, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3205-7. doi: 10.1145/2688500.2688538. URL `http://doi.acm.org/10.1145/2688500.2688538`.

[136] J. Weinberg and A. E. Snavely. Accurate memory signatures and synthetic address traces for hpc applications. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 36–45, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. doi: 10.1145/1375527.1375536. URL `http://doi.acm.org/10.1145/1375527.1375536`.

[137] C. Wu, J. Li, D. Xu, P. C. Yew, J. Li, and Z. Wang. Fps: A fair-progress process scheduling policy on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 26(2):444–454, Feb 2015. ISSN 1045-9219. doi: 10.1109/TPDS.2014.2306411.

[138] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 408–

421, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3651-2. doi: 10.1145/2806777.2806849.

[139] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 607–618, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. doi: 10.1145/2485922.2485974. URL `http://doi.acm.org/10.1145/2485922.2485974`.

[140] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee. Fast iterative graph computation: A path centric approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 401–412, Piscataway, NJ, USA, 2014. IEEE Press. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.38. URL `http://dx.doi.org/10.1109/SC.2014.38`.

[141] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1863103.1863113`.

[142] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. *Trans. Storage*, 2(3):283–308, Aug. 2006. ISSN 1553-3077. doi: 10.1145/1168910.1168913. URL `http://doi.acm.org/10.1145/1168910.1168913`.

[143] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 89–102, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9. doi: 10.

1145/1519065.1519076. URL http://doi.acm.org/10.1145/1519065.1519076.

[144] Q. Zhao, D. Koh, S. Raza, D. Bruening, W.-F. Wong, and S. Amarasinghe. Dynamic cache contention detection in multi-threaded applications. *SIGPLAN Not.*, 46(7):27–38, Mar. 2011. ISSN 0362-1340. doi: 10.1145/2007477.1952688. URL http://doi.acm.org/10.1145/2007477.1952688.

[145] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 45–58, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-201. URL http://dl.acm.org/citation.cfm?id=2750482.2750486.

[146] J. Zhong and B. He. Medusa: Simplified graph processing on gpus. *IEEE Trans. Parallel Distrib. Syst.*, 25(6):1543–1552, June 2014. ISSN 1045-9219. doi: 10.1109/TPDS.2013.111. URL http://dx.doi.org/10.1109/TPDS.2013.111.

[147] C. Zhou, J. Gao, B. Sun, and J. X. Yu. Mocgraph: Scalable distributed graph processing using message online computing. *Proc. VLDB Endow.*, 8(4):377–388, Dec. 2014. ISSN 2150-8097. doi: 10.14778/2735496.2735501. URL http://dx.doi.org/10.14778/2735496.2735501.

[148] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 177–188, New York, NY, USA, 2004. ACM. ISBN 1-58113-804-0. doi: 10.1145/1024393.1024415. URL http://doi.acm.org/10.1145/1024393.1024415.

[149] H. Zhu, L. He, and S. A. Jarvis. Optimizing job scheduling on multicore computers. In *Proceedings of the 2014 IEEE 22Nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, MASCOTS '14, pages 61–70, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-5610-4. doi: 10.1109/MASCOTS.2014.16. URL `http://dx.doi.org/10.1109/MASCOTS.2014.16`.

[150] H. Zhu, L. He, B. Gao, K. Li, J. Sun, H. Chen, and K. Li. Modelling and developing co-scheduling strategies on multicore processors. In *2015 44th International Conference on Parallel Processing*, pages 220–229, Sept 2015. doi: 10.1109/ICPP.2015.31.

[151] X. Zhu, W. Han, and W. Chen. Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, July 2015. USENIX Association. ISBN 978-1-931971-225. URL `https://www.usenix.org/conference/atc15/technical-session/presentation/zhu`.

[152] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-839-1. doi: 10.1145/1736020.1736036. URL `http://doi.acm.org/10.1145/1736020.1736036`.

[153] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, Dec. 2012. ISSN 0360-0300. doi: 10.1145/2379776.2379780. URL `http://doi.acm.org/10.1145/2379776.2379780`.