

A Thesis Submitted for the Degree of PhD at the University of Warwick

Permanent WRAP URL:

<http://wrap.warwick.ac.uk/152721>

Copyright and reuse:

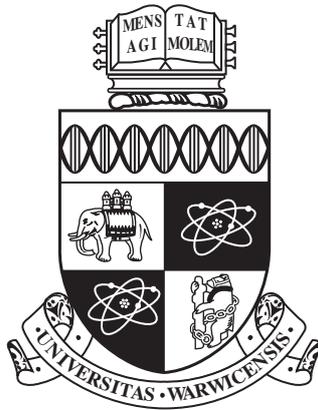
This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it.

Our policy information is available from the repository home page.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk



**Developing the Parallelization Techniques for
Finding the All-Pairs Shortest Paths in
Graphs**

by

Mohammed Alghamdi

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

July 2020

Abstract

Finding the All-Pairs Shortest Paths (APSP) is a fundamental graph problem aiming to find the shortest path between any two nodes in a graph. Solving the APSP problem in parallel is an active research area. Many algorithms have been proposed. In this thesis, a new approach are presented to solve the APSP problem for big graphs on shared and distributed systems. In this approach, a graph is partitioned judiciously and then processed in parallel. In particular, the graph is first pre-processed to prepare the partition in the computation stages. After the graph is partitioned into smaller sub-graphs, a traditional shortest path algorithm, such as the Floyd-Warshall algorithm or the Dijkstra's algorithm, can be used to find the APSP in each sub-graph. Finally, through the common nodes between the sub-graphs, the local results in each sub-graph are combined to establish the APSP for the entire graph. Our method are implemented using OpenMP for the shared memory architecture and MPI for the distributed memory architecture. In order to improve the scalability of the method, we proposed two different communication patterns among partitions (and processes) to achieve the parallelization and the combination of the local results. Further, we develop a hybrid CPU-GPU parallelization method, which can be run in a single multicore CPU and further be distributed across multiple GPUs to aggregate the results. We also developed two load-balancing schemes for this hybrid method. We have conducted extensive experiments on a high-performance

cluster with both simulated and real-world graphs. The experimental results show that comparing with the existing solution, our method is able to accelerate the solving of the APSP problem significantly.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor Dr. Ligang He, whose supervision, encouragement and support have been invaluable to me during my Ph.D journey. I benefited greatly from his insightful advices and comments in finding and solving research problems and academic work in general. I look forward to maintaining our collaboration in the future.

It is my great pleasure to work with all my lab-mates and special thanks to the workers in the Department of Computer Science at the University of Warwick, especially Sharon, and the I.T team, for providing the appropriate work environment.

I would like to thank my wife Salihah, who came with me to the UK to support me during my PhD. Her patience, encouragement and love made this journey easier and unforgettable time. To my son Azzam, whose smile was my greatest support.

Last but not the least, I would like to give my special thanks to the greatest parents who made the person I am today and always keep me in their prayers. Special thanks to my brothers and sisters for unlimited support during this period.

Declarations

This thesis is submitted to the University of Warwick in support of the author's application for the degree of Doctor of Philosophy. It has been composed by the author and has not been submitted in any previous application for any degree. The work presented was carried out by the author except where acknowledged.

Parts of this thesis have been previously published (or accepted) by the author in the following:

- 1 M. Alghamdi, L. He, Y. Zhou and J. Li. Developing the Parallelization Methods for Finding the All-Pairs Shortest Paths in Distributed Memory Architecture. *38th International Performance Computing and Communications Conference*.
- 2 M. Alghamdi, L. He, Y. Zhou and J. Li. Accelerating The solving of the All Pair Shortest Path Algorithm with GPU. *IEEE transactions on parallel and distributed systems*. under review

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- King Khalid University (KKU)
- Saudi Arabian Cultural Bureau in the U.K.

Abbreviations

APSP	All-Pairs Shortest Paths
AWS	Amazon Web Service
API	Application Programming Interface
BFS	Breadth First Search
CGT	Computational Graph Theory
CMPs	Chip Multi-core Processors
CP	Critical Path
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DFS	Depth-first search
DER	Desired Execution Requirement
EP	Embarrassingly Parallel
GA	Genetic Algorithm
GPU	Graphics Processing Unit
GPP	Graph Partitioning Problem
GC	Graph clustering
IBM	International Business Machines

IDC	International Data Corporation
IPAC	Infrared Processing and Analysis Center
MPI	Message Passing Interface
MPP	Multi Process Parallel
MTP	Multi Thread Parallel
OpenMP	Open Multi-Processing
OS	Operating System
PC	Process Communication
PM	Physical Machine
QoS	Quality-of-Service
SMP	Symmetric Multi-processors
SSSP	Single Source Shortest Path
TAA	Task Allocation Algorithm
TCP	Critical Path Execution Time
TLB	Translation Lookaside Buffer
TmpRT	Temporary Remaining Time
TS	Time-sharing
VCPU	Virtual Central Processing Unit
VM	Virtual Machine
VMM	Virtual Machine Monitor
WCET	Worst-case Execution Time

Contents

Abstract	ii
Acknowledgements	iv
Declarations	v
Sponsorship and Grants	vi
Abbreviations	vii
List of Figures	xiv
List of Tables	xv
1 Introduction	1
1.1 Parallelizing All Pair Shortest Path Algorithm	2
1.2 Key Contributions	5
1.3 Thesis Organisation	7
2 Literature Review	8
2.1 Introduction	8
2.2 Paralyzing APSP on Shared Memory Architecture	9
2.3 Parallelizing APSP on Distributed Memory Architecture	12
2.4 Using GPUs to Solve The APSP Problem	16

3	Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture	21
3.1	Introduction	21
3.2	The Parallel Method for Finding the All-Pairs Shortest Paths	23
3.2.1	Overview	25
3.2.2	The Method Steps	26
3.2.3	Reducing the number of operations	32
3.3	Graph Representation	34
3.4	Parallel Implementation and Analysis	35
3.4.1	Graph Preprocessing	36
3.4.2	Storing the Result	40
3.4.3	OpenMP	42
3.4.4	3D Matrix As Data Structure	42
3.5	Evaluation	44
3.5.1	Speedup of our parallel implementation over the serial implementation	45
3.5.2	Evaluating CNA with different parameter settings	46
3.5.3	Evaluating the parallel CNA with larger scale graphs	51
3.6	Summary	52
4	Developing the Parallelization Methods for Finding the All-Pairs Shortest Paths in Distributed Memory Architecture	54
4.1	Introduction	54
4.1.1	Motivation	55
4.2	Solving APSP in distributed memory architecture	56

4.2.1	Computing Operations	56
4.2.2	Communication Pattern among processes	59
4.2.3	Reducing Message Size	63
4.3	The Method Analysis	64
4.3.1	Communication Pattern 1	65
4.3.2	Communication Pattern 2	67
4.4	Experiments and Results	69
4.4.1	Comparing Our Method With the n-Dijkstra's Algorithm	69
4.4.2	Communication Patterns Comparison	71
4.4.3	Performance on Big Graphs Evaluation	75
4.5	Summary	76

5 Accelerating The solving of the All Pair Shortest Path Algorithm with GPU 77

5.1	Introduction	77
5.1.1	Motivation	78
5.2	HybridCNA	80
5.2.1	Processing on CPU	80
5.2.2	Processing on GPU	81
5.2.3	Data Transferring	82
5.3	H-Thread and H-Block approaches	83
5.3.1	Parallelizing the Combining Steps of HybridCNA	83
5.3.2	H-Thread	84
5.3.3	H-Block	87
5.4	Solving APSP on Multi-GPU systems	89

5.5	Experimental Evaluation	90
5.5.1	Evaluate H-Thread and H-Block	91
5.5.2	Comparing HybridCNA with SM-CNA	93
5.5.3	Evaluation on Multi-GPU systems	96
5.5.4	Evaluating HybridCNA with Real-world Graphs	99
5.6	Summary	101
6	Conclusions and Future Work	102
6.1	Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture	102
6.2	Developing the Parallelization Methods for Finding the All-Pairs Shortest Paths in the Distributed Memory Architecture	103
6.3	Accelerating the solving of the APSP problem with GPU	104
6.4	Future Works	105
	Bibliography	107

List of Figures

3.1	Graph partitioning using the BFS algorithm	24
3.2	Graph Representation, (b): adjacency list. (c): adjacency matrix [94]	34
3.3	Graph Representation using adjacency list for a wight graph [94]	34
3.4	Reordering the nodes by renaming a node with its index.	36
3.5	The performance of our two parallel implementations comparing to the serial implementation; The graph size is 16K	45
3.6	Speedup of serial and parallel CNA, Graph size is 16K	47
3.7	Comparing CAN, n-Dijkstra and ParAPSP on undirected graph	49
3.8	Comparing CAN, n-Dijkstra and ParAPSP on directed graph	50
3.9	The performance of the two versions of parallel CNA. The graph size is 16K	51
3.10	Experiment on big graphs	52
4.1	A case study for the combination of local results with the two communication patterns	57
4.2	Comparing our method with the n-Dijkstra algorithm	71
4.3	Comparing communication patterns 1 and 2; the number of graph nodes is 8k.	72
4.4	Performance of our method as the number of partitions increases.	73

4.5	Performance of our method as the graph size increases.	74
5.1	Example of a graph being partitioned into two subgraphs	80
5.2	Scheduling the operations of a graph partitioned into five sub- graph to run on three GPUs	89
5.3	Comparing the H-Thread approach with the H-Block approach. The graph size is 16k	91
5.4	Comparing HybridCNA with SM-CNA. The graph size is 16k	94
5.5	Comparing HybridCNA with SM-CNA. Graph size is 8k	94
5.6	Comparing HybridCNA with SM-CNA. Graph size is 4k	95
5.7	Comparing HybridCNA with SM-CNA. Graph size is 2k	95
5.8	Total time of HybridCNA and SM-CNA methods in different graph size	96
5.9	Comparing the kernel time of running with 1 GPU and run- ning with Multiple GPUs. The graph size is 8k	98
5.10	Comparing the kernel time of running with 1 GPU and run- ning with Multi-GPUs. The graph size is 16k	98
5.11	Comparing the total time between running with 1 GPU and running with Multiple GPUs on different graph sizes	99
5.12	Testing CPU and Hybrid methods on a real-world networks .	100

List of Tables

3.1	The summary of all symbols used in this chapter	25
3.2	Example of number of nodes actually duplicated on real-world networks	37
4.1	Communication and combination times (in milliseconds) with pattern 1 and pattern 2; the graph with 8K nodes are divided into 8 partitions	71
5.1	A case study of finding APSP in a graph with the size of 4k size; the graph is divided into two partitions.	78
5.2	Case study of the operations needed to combine the result of five subgraphs	83
5.3	Case study of finding APSP of a graph with the size of 4k and partitioned into two subgraphs.	92
5.4	A case study of comparing multi-copying with single copying strategy. The graph size is 8K and the graph is partitioned into 8 subgraphs (Time in milliseconds)	97
5.5	The dataset for the real networks we run the experiments on .	100

CHAPTER 1

Introduction

Determining the shortest path amid two or more objects in a graph is a prevalent task in solving numerous daily scientific problems [27]. The algorithms for examining the shortest path have their use in numerous domains, for instance, bioinformatics, social networks, aviation, Google maps, routings protocols and others [23] [18]. All pair shortest path (APSP) is a type of shortest path algorithms. Given a directed or undirected weighted graph $G(V, E, w)$, where V is the set of nodes in the graph, E is the set of weighted edges connecting the nodes, and w is the weight of that edge, the APSP algorithm returns the shortest path between any two nodes $V \in G$, where the shortest path is defined as the minimum sum of edge weights on the path that connect two nodes in the graph. There are diverse algorithms for examining the all pair shortest paths, for example, the Johnson's algorithm [55] and Floyd-Warshall algorithm [36]. Another well-known algorithm is Dijkstra's algorithm [29], which is initially used to solve the Single Source Shortest Path (SSSP) problem. However, when it runs from all nodes in the graph, it can solve the APSP problem for the graphs that do not containing the edges with negative weights.

For decades, researchers try to improve the performance of the APSP al-

gorithms. Many of them were inspired by the Floyd-Warshall algorithm and the Dijkstra's algorithm [69] [84]. The studies in APSP fall into two general categories: i) sequential solution, where the computation cost is reduced during the graph processing; ii) parallel solution [61], where the computation is parallelized across multiple processing elements to speed up the processing. The approach proposed in this thesis falls into the second category. We aim to improve the performance of the APSP algorithms by developing parallelization strategies for shared memory computer architecture, including multi-core computers and GPU, and distributed memory architecture such as a cluster system.

1.1 Parallelizing All Pair Shortest Path Algorithm

Recently there is the noticeable increase in using the graph to model real-world problems, which consequently demand the development of the methods for efficient graph processing [65]. The massive computation load needed to solve the APSP problem makes the sequential processing an impractical solution. Consequently, the attention to parallelizing the APSP process has been rising lately [7]. Shared memory architecture such as a multicore machine or a GPU device is a mainstream of parallel processing architecture. The main advantage of the shared memory architecture is its fast communication between the processing cores since resources are shared [33]. Therefore, a library that supports shared memory multiprocessing must be called to control

the usage of these shared resources. OpenMP is the most widely used library in this aspect [19]. In chapter 3 of this thesis, we will discuss our parallelization strategy of the APSP problem on the shared memory architecture, in which, we use the node duplication approach to reduce the computation cost significantly and achieve the excellent performance.

The sizes of graphs, such as social network graphs [81] and web-pages graphs [22], have been growing and become much larger over time, which makes it difficult to be stored and processed in the memory of a single machine [35]. Therefore, many works presented the approaches to processing the APSP problem in a distributed memory architecture such as [75] [50] [51]. When a graph is processed in a distributed system, the graph is partitioned into small subgraphs, each of which is processed by a processor in parallel. In a distributed system, every processor has a separated memory and is required to communicate with other processors through message passing. Message Passing Interface (MPI) is the most widely used parallel programming paradigm for distributed systems [41]. However, to take advantage of the power of distributed systems, managing the communication cost effectively is a necessity. Reducing communication cost is another research effort that many researchers are devoted to when solving the APSP problem in a distributed environment [38] [53]. In this theses, We addressed this problem by proposing two new communication patterns for solving the APSP problem on a distributed memory architecture.

According to [47], one of the main factors that affects the communication behaviour of processors is the way the graph is partitioned among the

processors. This effect plays a main role in determining the number of messages required to exchange between the processors as well as the size of these messages. There are many ways to partition a graph. Two well-known ways are random cut and multi-level cut [95]. The random cut is straightforward, which partitions the graph to achieve the balance (i.e., the total number of nodes in each partition) among the partitions [17]. However, such a random partition often leads to high communication cost among processors when the graph is processed in parallel in later stages. The partition method proposed in this thesis belongs to the category of multi-level cut [48], in which the graph is transformed into the graph levels by applying the algorithms such as Depth First Search (DFS) and Breadth-First Search (BFS) [34]. In addition to reducing the communication cost, another advantage of this partitioning method is to ensure the connectivity between the nodes in the same partition.

Graphics processing units (GPUs) have evolved from the specialized image processing device to general-purpose parallel processing platform. [88] [44]. Many researchers use it to improve the performance by taking advantage of much higher degree of parallelism than the multicore computers (multicore CPU) [63]. Considering that solving the APSP problem is computationally expensive, GPU offers the excellent potential to speedup the processing. However, the hardware structure of GPUs is entirely different from CPUs. Applications are required to be modified for running on GPUs and achieve the desired performance improvement [32]. In order for the application developers to program the GPU version of their applications, the libraries for GPU programming are developed, such as Nvidia CUDA [28]

and OpenCL [42]. In this thesis, we exploit the features in GPU and develop a method to accelerate the running of the APSP algorithm on GPU.

1.2 Key Contributions

In this thesis, we present a novel method to solve the APSP problem in both shared memory, including multicore computers and GPU, and distributed memory architectures such as a cluster system.

In chapter 3, we develop our method for parallelizing the solving of the APSP problem on a shared memory architecture. We present a new method in which the graph is partitioned using the BFS algorithm and the common nodes between subgraphs are duplicated. Then, the APSP of each subgraphs is computed in parallel, and the local results in subgraphs are combined through the duplicated nodes to find the APSP for the entire graph. Our method reduces the overall computation time significantly and achieves outstanding performance. Moreover, The mathematical correctness of our method is proved. Furthermore, we design an efficient data structure to store the intermediate calculation results during the graph processing and the final APSP result. We have conducted extensive experiments to evaluate the effectiveness of our method. The experiments show that our method achieved up to 6 time faster than existing methods.

To address the problem of memory limitation in a single shared-memory computer, we develop a parallelization approach in chapter 4 to solving the

APSP problem in the distributed memory architecture. By adding more machines, even larger-scale graphs can be processed and the much higher degree of parallelism can be achieved. In our method, two communication patterns are proposed to communicate the local results between the processors after the APSP has been computed for each subgraph in parallel. This method is implemented by MPI. Both patterns reduce the communication cost and consequently improve the overall performance. An example of this improvement, our algorithm can find 16 million paths in about two seconds. Furthermore, we have conducted the experiments on a high-performance cluster and evaluated our method with the graphs of different sizes.

In chapter 5, we accelerate our shared memory method using the massive parallelism offered by GPU. We propose a hybrid method in which the CPU finds the APSP of the subgraph and schedules the steps of combining local results to be run on GPU. Moreover, we develop two thread management approaches to controlling the thread processing on the GPU. By launching the appropriate number of threads, both thread management approaches minimize the data transferred between the CPU memory and the GPU memory, which leads to significant performance improvement. Furthermore, we extend the method to a multi-GPU system and improve the performance further. We have conducted the experiments with simulated and real-world graphs. The results show outstanding performance (up to 2.5x faster) comparing to our CPU version method and other algorithms.

1.3 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, we present the literature review of the related research topics. In Chapter 3, we present a new method for finding APSP on a shared memory system and also propose a new technique to reduce the storage of the graph processing results in the memory, which allows us to process larger graphs. Chapter 4 presents our method for solving APSP on the distributed system. Two communications patterns are proposed to transfer the local results calculated by each processor. In Chapter 5, we use GPU to further parallelize the algorithms proposed in Chapter 3 and present a hybrid CPU-GPU method for solving the APSP problem. Finally, Chapter 6 concludes all the work presented in this thesis and discusses the future work.

CHAPTER 2

Literature Review

2.1 Introduction

Numerous real-world problems can be modelled by the graph structure, many of which require the computations of shortest path [93]. The examples are internet route planners [85], road networks [10] and web searching [9]. Therefore, it draws the researchers' attention to study the efficient methods of solving the all-pairs shortest paths (APSP) for graphs. Recently, most of these studies focus on developing parallel solutions, given the heavy computations required to compute APSP and the noticeable advance in parallel architecture such as multi-core computers, cluster and GPU[68].

In the implementation of parallel APSP methods, the traditional algorithms that the parallel methods uses are a key factor. Habbal et al. in [45] argued that Floyd's algorithm or the triple operation algorithm was among the most common algorithms employed in APSP. The algorithm employs an insertion method to ensure the optimal conditions are satisfied between the pair of nodes on a shared memory location. Many studies proposed their works based on Floyd-Warshall algorithms such as those in [54] [70] [98] [67] and [13]. On the other hand, numerous works are based on the single-source shortest path (SSSP) algorithm such as Dijkstra's [29] and Bellman-Ford al-

gorithms [40]. According to [14] applying the SSSP algorithm repeatedly can solve the APSP problem. By using this approach, the SSSP algorithm can be configured to run in parallel where each node can be computed independently. The works that used the SSSP approach to solving the APSP include those presented in [46], [37], [83], [100], [31] and [60].

The remainder of this chapter is organized as follows. In Section 2.2 we review the current work in solving the APSP problem in parallel on the shared memory architecture. We review the existing distributed algorithms for solving the APSP in Section 2.3. In section 2.4, we discuss the related research for accelerating the processing of APSP on the GPU.

2.2 Paralyzing APSP on Shared Memory Architecture

Parallellizing APSP poses many challenges. According to reference [1], the best algorithm is the one that is capable of optimizing time and resource usage. Therefore, a number of approaches have been presented from different angles to address the shortest-Path Problem. This has led to the development of new approaches to decomposing the APSP problem for the ease of management. It is worthy noting that reference [6] also argued that due to the availability of powerful computing capabilities, current studies are focusing more on having efficient algorithms to improve the performance of finding the shortest path in huge networks. This means that computing system that

has large shared resources is capable of splitting a big network into sub-networks and accommodate the sub-networks in various memory resources. Such a system makes it possible to utilize the processing power to achieve the maximum performance, particularly for complex problems. The above approach has become more practical in that it applies the divide-and-conquer mechanism, which allows the sub-domains to be processed in parallel. The approach therefore allows the solution of the sub-domains to be integrated and form the optimized solution to the original problem. With regards to this problem, researchers have advocated for the approaches that adopt the network decomposition techniques to optimize the usage of computing and storage resources [66].

The study presented in [82] aimed at investigating the performance of an algorithm for solving the parallel all-pairs shortest path problem with a network decomposition approach. However, the study employed a single level parallelization approach, where all processors were assigned to find the shortest paths for a single sub-network at a time. By recording the time taken for each sub-network until all sub-networks were processed it was possible to compute all time required to process the network. Additionally, parallel single-source shortest path algorithm execution time was used as the benchmark for his study. Consequently, the study found that the time needed to execute the decomposition-based algorithm and the parallel SSSP algorithm decreased as the increase in processors. This result is confirmed by the experiments conducted with the decomposition-based algorithm. Therefore, the approach employed a multi-phase approach to decompose the network into

independent directed acyclic sub-networks for efficient parallel processing. The research further found that the performance of an algorithm is limited to the network decomposition approach used. Consequently, the study developed an algorithm that can save about 50% of the execution time compared to the benchmark algorithm used in the study. The results were supported by the study presented in [21], which argued that the increasing resources in a computing environment improve the performance proportionally given all other factors remain constant.

Yanagisawa et al. in [99] presented a multi-source label correcting (MSLC) algorithm to solve the APSP problem, which is an extension of the labeling method (Cherkassky, Goldberg, and Radzik) [20] for the single source shortest path (SSSP) problem. The MSLC algorithm used the Single Instruction Multiple Data (SIMD) model to exploit the data-level parallelism in the scan operation. This algorithm uses the working space of $O(m + Bn)$ (where m is number of edges and n is number of nodes) and works only with the sparse graphs. They tested the algorithm with a road network and synthetic graphs.

Another work about the parallelization of the APSP problem is presented in [57]. The work first proposes a fast sequential algorithm which ranks the nodes in the descending order of their degrees and then runs the Dijkstra's algorithm. The study proposes an efficient shared-memory and parallel allpairs shortest path (APSP) algorithm that was based on the modern sequential APSP algorithm. The algorithm was therefore able to practically meet time complexity of $O(n^{2.4})$ (n is number of nodes) to further optimize the perfor-

mance. Moreover, the study proposed a parallel algorithm without the need for extra partitioning steps. The algorithm named ParAPSP is designed based on the principle of Dijkstra’s algorithm advocated in [80]. Therefore, the algorithm offers an efficient idea that does not need to load the previous values of shortest paths. Furthermore, the study advocated for optimizing its ordering scheme and proposed a solution to achieving the speedup.

The work presented in [89] is one of the latest work that solved the APSP problem on shared memory. They advocated that the Floyd-Warshall algorithm is a more superior algorithm to process the all pairs shortest path problem. This was particularly observed in high-performing applications. However, the algorithm was found to misbehave in sparse graphs and perform unnecessary asymptotic work. Therefore, the study argued for using the known algebraic relationship found between Floyd-Warshall and Gaussian limitation among other methods to solve the problem. The purpose of this approach is to improve locality, reduce the computation and enhance the parallelism.

2.3 Parallelizing APSP on Distributed Memory Architecture

According to Solomonik et al.[92], the work presented in [52] was the first to propose an algorithm for the APSP problem on 2D distributed memory which considers the original schedule of Floyd-Warshall. Blocking opera-

tions are not applied in the algorithm and thus the global synchronization has to be performed. Later this work was improved in [59] through the use of pipelining for avoiding the global synchronizations. Therefore, the sync cost was reduced, and there was low reuse of data in these algorithms with each processor performing a number of updates by the local unblocked operations. Obtaining a high performance in practice requires an increase in data locality, which can be achieved through the blocked algorithm of divide and conquer. The divide and conquer algorithm presented in [87] is widely used by researchers to solve the APSP problem in a distributed environment. Most works discussed in this section are based on this approach.

The work in reference [92] implemented a divide and conquer algorithm to solve the APSP problem. In order to scale the APSP problem to higher concurrencies, two requirements have to be fulfilled: i) maximize the temporal data locality, and ii) minimize the inter-processor communication. Thus, an APSP algorithm called block-cyclic 2D algorithm has been proposed in this paper, which helps minimize the latency and maximize the usable bandwidth. This technique minimizes the communication and maximizes the temporal locality reuse. The authors conducted the experiments on a supercomputer with 24,576 cores, and the results show that the implementation of the divide-and-conquer APSP algorithm is well suited for a distributed memory environment with its lower bandwidth costs and improved parallel scalability.

The work presented in [3] has discussed the problem of computing APSP

and the shortest paths for k sources in a weighted graph in the distributed CONGEST model. For the graphs with non-negative integer edge weights (including zero weights), the authors make use of a pipelined algorithm to calculate the shortest path distances. The authors used the construction technique called h -hop CSSSP collection, which can be constructed neither using a pipelined algorithm nor the Bellman-Ford algorithm. It places additional stringent conditions on the structure of h -hop SSSP trees in the collection. Also, the authors made use of the simplified versions of the short-range algorithms that run in two stages, namely i) increment of every zero-degree weight to a positive value, and ii) then computing the h -hop SSSP through a BFS variant. The results reveal that the deterministic distributed algorithms turn out to be beneficial to computing weighted shortest paths in the graphs with moderate non-negative integer weights. This work was further improved in [4], where the authors implemented the pipelined algorithms on both directed and undirected graphs.

Ghaffari et al. in [39] states that computation of shortest paths is one of the central problems in the theory of distributed computing. Thus as a solution, it presented a single source shortest path algorithm with the complexity of $O(n^{3/4}D^{1/4})$ (where n is number of nodes and D is the hop-diameter of the graph). Also, an improved variant of the algorithms was introduced to deal with the graphs with larger diameters. Moreover, the authors implemented the algorithms on both undirected and directed graphs. In terms of techniques used, the authors proposed the optimization techniques such as scaling the framework of a graph with integer weights, implementing a

hybrid of Bellman-Ford and BFS algorithms, virtually sampling the nodes, distributed scheduling of the algorithms. The results obtained indicate that the single-source shortest path algorithm is beneficial and leads to the improvement in finding APSP of integer-weighted graph. Also, the usage of various algorithmic techniques helped simplify the complex problem of computing shortest paths.

Anther research in [5] presents a deterministic distributed algorithm to compute APSP in a weighted directed or undirected graph. Several algorithms were implemented by the authors at different levels. These algorithms included a compute-blocker algorithm to find a blocker set and the algorithms that calculate the initial and ancestors of a node. Besides, a pipelined algorithm to update the scores at various nodes was implemented. Moreover, the classic Bellmen-Ford algorithm was used for weighted APSP. The implementation of these algorithms resulted in a new distributed algorithm for computing weighted APSP in both directed and undirected graphs. The results indicated that the algorithm could solve the APSP problem in about $O(n^{3/2})$ rounds (where n is the number of nodes).

The work in reference [11] investigates the problem of approximation time algorithm and the nearly-tight lower bound in APSP. Consequently, it provides the solution to this problem by developing a randomized time algorithm that matches the lower bound to the polylogarithmic factors. The authors made use of directed graphs with zero and negative edge weights. Moreover, a random filtered broadcasting technique was used, which applies to

the settings where one needs to broadcast a large amount of information to every vertex. It has also been shown in the paper that how one can use the randomization to filter out most of the messages and reduce the congestion on each edge. The algorithm presented was mathematically proved. It reduced the running time of solving APSP. However, they did not provide an experiment with any graph to support their theory.

2.4 Using GPUs to Solve The APSP Problem

In graph theory, determining shortest paths is termed as the basic operation. The primary challenge in solving the APSP problem is that it needs a considerable amount of computation to determine APSP. Therefore, with GPU devices being improved over time, the mechanisms for parallelizing the APSP on GPU have been developed [97].

Ortega-Arranz et al.[79] conducted the experiments with large graphs. They described a shared memory implementation using Dijkstra's algorithm to solve the APSP problem on directed sparse graphs. They presented the parallel solutions for a heterogeneous system consisting of two GPUs and a multicore CPU, and implemented two load-balancing methods. They achieved the best performance through the equitable scheduling that maps all costly tasks to GPUs and leaves light ones to the CPU cores.

The work in [78] shows GPU can be modified to accelerate the solving of APSP. Through their proposed scheme, they presented a fast algorithm with the capability of solving the APSP problem on the CUDA-enabled GPU.

The scheme, which was based on the SSSP-based algorithm, was able to solve multiple SSSP problems in parallel as the result of efficient usage of the on-chip shared memory. The algorithm allowed stream processors (SPs) to concurrently access similar data since every SP participates in solving one of the tasks. Notably, this kind of access, which is common, reduces the access of data to the off-chip memory. Moreover, their scheme contributed to achieving higher speedup with less synchronization and more parallelized tasks on the GPU.

Another work that used multi-SSSP to solve the APSP on GPU is proposed in [72]. It presents a multi-search abstraction as a method for expressing the algorithms that execute the BFS algorithm simultaneously. This research involves an efficient implementation of the abstraction, which is demonstrated to outperform the existing GPU methods implicitly for large graphs of various diameters by more than a factor of two. The authors further demonstrated that a single GPU can be used to solve the APSP problem on sparse graphs with millions of nodes. Their BFS algorithm works only on unweighted sparse graphs, and solves the APSP problem with the complexity of $O(mn)$ (where m is number of edges and n is the number of nodes).

On the investigation concerning a parallel implementation of Johnson's algorithm, reference [93] developed the approach which has the capability of solving the APSP problem based on the current or recent GPU architecture. The objective of the new approach was formulated to increase the speed of APSP computation for large graphs in relation to the CPU. Since

GPU can provide high computational cost at a minimal cost, it has been utilized as a substantial alternative. Additionally, to enhance the execution of operations on GPU, the operations are programmed using the framework such as CUDA. The study experimented on three parallel implementations of Johnson’s APSP algorithm on the GPU. It further compared the three implementations based on their execution times to determine their advantages and drawbacks.

A work that used the Floyd-Warshall algorithm is described in [71], which presented a blocked algorithm to solve the APSP problem on a hybrid CPU-GPU system. They proposed a united algorithm that can solve the APSP problem for a graph whose size is greater than the capacity of the GPU memory. The total number of operations for this algorithm is $2n^3$ (n is number of nodes). Their algorithm achieved the peak performance when the number of vertices in the graph is larger than a few thousand.

The work in reference [30] is implemented on a distributed GPU. It aimed at exploiting the input graph structure and the partitioning properties to parallelize the shortest path computations. The approach was based on two algorithms. The first one is the Floyd–Warshall algorithm, which helps the approach to work with the graphs containing negative edges, while the second algorithm is the Dijkstra algorithm, which is used for various computations. Therefore, the approach can only be utilized with positive edge weight graphs. The two algorithms used have the optimal time complexity; on the other hand, their regular matrix-based structure is adequate to allow efficient

implementation on the GPUs. Additionally, the approach has the nearly optimal number of operations, steady matrix-structured computations as well as an approach with a high degree of parallelism. The divide-and-conquer approach was applied to enable the use of multi-GPU clusters.

One of the latest work that solved the APSP on GPU is presented in [74]. They implemented a GPU version of the adjacency matrix (ADJ-APSP) and the breadth-first search (BFS-APSP), and then evaluated its performance. Moreover, they developed two versions of their method. The first version is implemented by OpenMP, while the second one is a hybrid implementation by OpenMP and MPI. They conducted the experiments on an unweighted graph only. The results showed that parallelizing ADJ-APSP on a single GPU improved the performance by up to 16.53 times over the single-CPU implementation. On the other hand, parallelization over multiple GPUs achieved even more performance improvement, with the recorded speedup of up to 101.10 times over the single-GPU implementation.

To conclude, the work discussed in this chapter addressed the problem of improving the performance of the APSP in sheared and distributed memory architecture. Our work in this thesis succeeded to improve it up to 6x faster than some compared methods. We partition the graph in a novel way which led to reducing the number of computation needed to find the shortest paths massively. In the distributed memory environment, the communication between the processor is a main challenge. Our partitioning method helped us to design two communication patterns which reduced communication cost

and improve the overall performance. Moreover, unlike many GPU works discussed in this chapter, we do not need to copy the graph itself to the GPU, but only copy a distance matrix once, which reduces the communication cost and also allows us to solve the APSP for bigger graphs that cannot be fitted in the GPU memory.

CHAPTER 3

Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture

3.1 Introduction

Finding the All-Pairs Shortest-Path is an essential problem, which has been studied for decades. There are many applications of this problem such as road networks, transportation, and robotics. The problem is to find the shortest path between all nodes in a graph, where the shortest path between two nodes is defined as the path with minimum total weights of nodes and edges among all paths between the two nodes. Many algorithms are designed to solve this problem. Two of the well-known algorithms are the Floyd-Warshall algorithm and the Dijkstra's algorithm [91]. Dijkstra's algorithm is initially used to solve the Single Source Shortest Path (SSSP) problem. When it runs for all node pairs in the graph, it can solve the APSP problem for the graphs that do not contain negative edges. The Floyd-Warshall algorithm solves the APSP on graphs with positive or negative edges in n iterations (where n is the number of nodes in the graph). Most research for solving this problem used these two algorithms.

As the graph size continues to explode nowadays, much research investigated the parallelization of graph processing [73], including parallelization of the Floyd-Warshall algorithm and the Dijkstra's algorithm. Our method supports Dijkstra's algorithm. In theory, however, any algorithm that can solve the APSP problem can be supported by the work proposed in this chapter.

Parallelization of graph processing often involves graph partitioning. Two factors need to be considered when partitioning a graph: balance of sub-graph sizes and communication cost between the subgraphs. A naive way to achieve the balanced partitioning is to partition the graph nodes randomly (i.e., random cut of graph edges) as long as the resulting sub-sets are balanced in terms of the number of graph nodes [17]. However, such a random partition often leads to high communication cost among sub-graphs when the graph is processed in parallel in later stages. Another way of partitioning a graph is to use the multi-level cut [48], which can be achieved by many algorithms, e.g., the Depth First Search (DFS) and Breadth First Search (BFS) algorithms [25]. Although this partitioning method does not guarantee the sub-graph balance, it has the advantage of reducing the communication cost and ensuring the connectivity between the nodes in the same partition. We used the multi-level cut as a way to partition the graph in our method.

Shared memory architecture such as a multicore machine or a GPU device is the main stream of a parallel processing architecture. Most parallelization strategies in the literature are developed for the shared memory architecture [79],[56],[72], [99], [57]. The main advantage of the shared memory architecture is the fast communication between the processing cores. In this chapter,

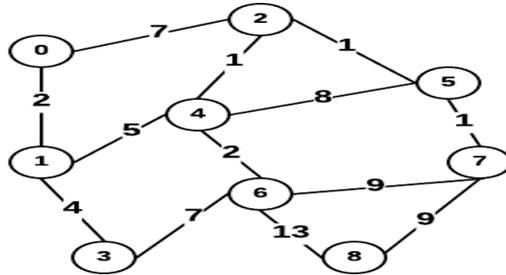
we introduce a new method to solve the APSP problem on a shared memory environment. In this method, the graph is partitioned into subgraphs. All-pairs shortest paths in each subgraph are computed in parallel. Then the local results in sub-graphs are combined to find all pairs shortest paths in the entire graph. The proof of the method correctness is given. Further, we reduce the computation by using an indexing strategy that reorders the nodes in subgraphs, and use a 3D matrix to reduce the memory consumption so as to enable the processing of bigger graphs. Moreover, we conduct the experiments with different types of graph to verify the effectiveness of the proposed method.

The remainder of this chapter is organized as follows. In Section 3.2 we present our method in detail. The representation of the graphs is described in Section 3.3. Section 3.4 presents the analysis of the parallelization implementation including the preparation of the graphs. Experimental results are presented in Section 3.5. Finally, this chapter is concluded in Section 3.6.

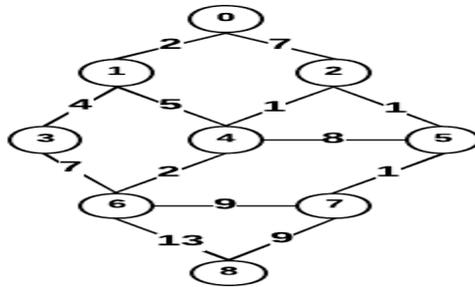
3.2 The Parallel Method for Finding the All-Pairs Shortest Paths

In this section we present our method in detail for solving the All-Pairs Shortest Paths (APSP) problem. First, in subsection 3.2.1 we briefly summarize the problem and determine the graph type we target in this chapter. Subsection 3.2.2 then discusses the steps of our method that solve the

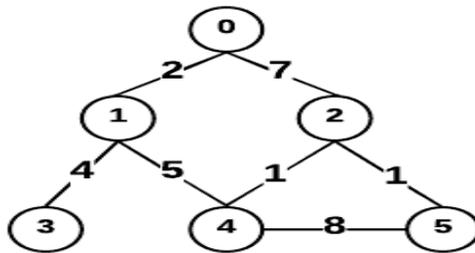
3. Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture



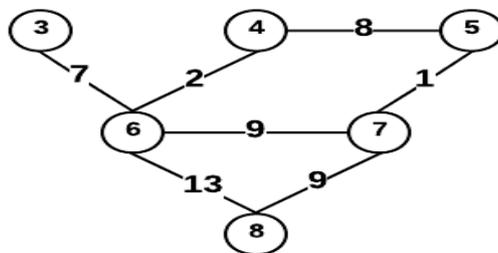
(a) Original graph



(b) BFS graph from node (0)



(c) First partition ($P1$)



(d) Second partition ($P2$)

Figure 3.1: Graph partitioning using the BFS algorithm

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

V	The total number of nodes in graph
V_i	The nodes v with index i in the graph
E	The number of of edge in a graph.
W	The weight of an edge connecting two nodes.
P	The total number of partitions.
P_i	The index of partition.
$S(v_i, v_j)$	The shortest path between node i and node j .
$T(v_i, v_j)$	The temporary shortest path between node i and node j .
$W(v_i, v_j)$	The weight of the shortest path between node i and node j .
$W_T(v_i, v_j)$	The weight of the temporary shortest path between node i and node j .
C	The total number of common nodes in the graph.
$C_{i,j}$	The common nodes between two partitions in the graph.
thr	The total number of core or threads.

described problem in detail. Subsection 3.2.3 presents how we reduce the number of operations and increase the performance of the method.

3.2.1 Overview

Given a directed or undirected weighted graph $G(V, E, W)$, where V is the set of nodes in the graph, E is the set of weighted edges connecting the nodes, and W is the set of weights of the edges. In the case of an unweighted graph, we assume that all edges have equal weights. The shortest path between two nodes, v_i and v_j , is denoted by $S(v_i, v_j)$. $T(v_i, v_j)$ denotes the temporary shortest path between v_i and v_j . $W(v_i, v_j)$ and $W_T(v_i, v_j)$ are the weights of the shortest path and the temporary shortest path, respectively, between v_i and v_j . P is the number of partitions (sub-graphs) after partitioning graph G . In this work, we solve the All-Pairs Shortest-Path (APSP) problem for G by partitioning G and processing each partition in parallel.

3.2.2 The Method Steps

There are five stages in our method, which are detailed in this subsection.

Stage 1: graph partition

We partition the graph in this stage. We first run the BFS (Breath-First Search) algorithm on the graph and transform the graph into a multi-level graph. The root node of the graph has the level 0. When BFS visits the successor nodes of root, it labels them with level 1. After BFS is completed, every graph node is labelled with a level and the graph is effectively transformed into a multi-level graph. Figures 3.1 (a) and (b) show an original graph and the corresponding multi-level graph whose nodes are arranged in levels. The graph is partitioned across the nodes in the same level, which is called the boundary level of two neighbouring partitions. Apart from the partitioning, the nodes on the boundary level are duplicated in both neighbouring partitions. Assume the multi-level graph is partitioned into two parts (i.e., P_1 and P_2) across the nodes 3, 4 and 5 in figure 3.1 (b). Nodes 3, 4 and 5 are duplicated and the resulting two sub-graphs are shown in Figure 3.1 (c) and (d), in which nodes 3, 4 and 5 appear in both sub-graphs.

In general, after graph partition, each part P_i contains a number of levels. Since we duplicate the boundary level, the last level in P_i is the first level in P_{i+1} . We call the nodes in the duplicated level the common nodes. The set of common nodes duplicated in the neighbouring partitions P_i and P_{i+1} is denoted by C_i . In Figure 3.1, P_1 contains three levels: levels 0, 1 and 2,

while P_2 contains levels 2, 3 and 4. The nodes on level 2 (i.e., nodes 3, 4 and 5) are called the common nodes between P_1 and P_2 .

In order to achieve the node balance among partitions, we set a counter when running the BFS algorithm. When BFS visits a node and the counter is equal to N/P , where N is the number of nodes in graph G and P is the number of sub-graphs we decide to partition graph G into, the level that contains that node is set as the boundary level.

Stage 2: Find the temporary shortest paths in each subgraph in parallel

In this stage, we apply either the Floyd-Warshall algorithm (or any all-pairs shortest path algorithm) on each partition and find the All pairs Shortest Path (APSP) in each part in parallel. After the Floyd-Warshall algorithm is completed, $T(v_i, v_j)$ (temporary shortest path between v_i and v_j) is obtained for all node pairs in each partition. However, since the common nodes are duplicated in the neighbouring partitions, different values of $T(v_i, v_j)$ will be obtained by the algorithm in each partition when v_i and v_j are the common nodes. For example, we can see from figure 3.1 that there will be only one path found between nodes 0 and 3 in P_1 , which is $\langle 0, 1, 3 \rangle$ and whose distance is 6. In P_2 , only one path is found between nodes 7 and 8, which is $\langle 7, 8 \rangle$ and whose distance is 9. However, both partitions will calculate the shortest paths between the common nodes 4 and 5, whose distances are $\langle 4, 2, 5 \rangle = 2$ in P_1 and $\langle 4, 5 \rangle = 8$ in P_2 .

Just because the distance of a path between the common nodes may be different in two neighbouring partitions, a shortest path between two nodes

calculated by a partition may not be the actual shortest path in the graph (hence called temporary shortest path) if the path go through two common nodes.

Stage 3: Calculate the actual shortest paths in each subgraph

As mentioned in stage 2, different partitions have different $T(v_s, v_t)$ for common nodes. In this stage, we compare the path distances between the common nodes obtained by the neighbouring partitions, and the one with the shorter distance is kept as the shortest distance for this pair of common nodes. Assume for partitions P_k and P_{k+1} , and a pair of common nodes (v_s, v_t) , $v_s, v_t \in C_k$, we keep $W_T(v_s, v_t)$ (the distance of the temporary shortest path) calculated in P_k . Then, we update all paths (and their shortest distances) that contain nodes v_s and v_t in P_{k+1} using the following equation, where the temporary shortest path between v_i and v_j go through the common nodes v_s and v_t .

$$W(v_i, v_j) = W(v_i, v_j) - W_T(v_s, v_t) + W(v_s, v_t) \quad (3.1)$$

For example, in Figure 3.1, after comparing the distances of the temporary shortest paths between nodes 4 and 5, we keep the path $\langle 4, 2, 5 \rangle$ in P_1 because its distance is 2, less than the other path $\langle 4, 5 \rangle$ in P_2 , whose distance is 8. We apply Equation (1) to the path $\langle 7, 5, 4 \rangle = 9$ in P_2 , which contains the common nodes 4 and 5. The shortest distance of $\langle 7, 5, 4 \rangle$ is then updated to be $W(7, 4) = 9 - 8 + 2 = 3$.

After stage 3, we can obtain the actual shortest paths between two nodes

that are in the same partition, which is proved in the following theorem.

Theorem 1. *The shortest path between any node pair (v_i and v_j) in the same partition are obtained after Stage 3 is completed.*

proof *There are three cases for the shortest path calculated in a partition (assume to be P_k) between v_i and v_j .*

Case 1: The shortest path between v_i and v_j does not contain more than one common node. In this case, all possible paths between v_i and v_j are in P_k . So the shortest path between v_i and v_j calculated in P_k in stage 2 is the actual shortest path.

Case 2: The shortest path between v_i and v_j contains two common nodes, e.g., v_s and v_t . Such a path can be divided into three path segments, $\langle v_i, \dots, v_s \rangle$, $\langle v_s, \dots, v_t \rangle$ and $\langle v_t, \dots, v_j \rangle$. Then the shortest path in the first (or the third) path segment calculated in the partition are the actual shortest path between v_i and v_s (and between v_t and v_j) as stated in case 1. For the path segment $\langle v_s, \dots, v_t \rangle$, all possible paths between v_s and v_t are in either partition P_k or in partition P_{k+1} which shares the common nodes v_s and v_j with P_k . Therefore, the shorter temporary shortest path between v_s and v_t calculated in P_k and P_{k+1} must be the actual shortest path between the two common nodes.

Case 3: The shortest path between v_i and v_j contains more than two common nodes. In this case, the shortest path can be divided into more than 3 path segments. Each path segment is either a path falling into case 1 or case 2. Therefore, the shortest path between v_i and v_j calculated by stage 3 is the actual shortest path between the two nodes. \square

Stage 4: Calculate the shortest distance of a path that crosses two neighbouring subgraphs

In this stage, we combine the local results obtained in two individual neighbouring partitions and obtain the shortest distance of a path that crosses these two partitions (i.e., the source node is in a partition and the destination node is in the neighbouring partition).

The shortest path between $v_i \in P_i$ and $v_j \in P_j$ MUST pass at least one common node between the two partitions. We make use of this insight to combine the local results in two neighbouring partitions. The calculation is presented and proved in the following theorem.

Theorem 2. *Assume $v_{k_1}, v_{k_2}, \dots, v_{k_r}$ is the set of common nodes that are connected to both v_i and v_j . Then the distance of the shortest path between v_i and v_j , $W(v_i, v_j)$, can be calculated by Eq. 3.2.*

$$W(v_i, v_j) = \text{Min}\{W(v_i, v_{k_m}) + W(v_{k_m}, v_j) | 1 \leq m \leq r\} \quad (3.2)$$

Proof. *The shortest distances between v_i and v_{k_m} and between v_{k_m} and v_j can be calculated in Stage 3. The sum of them is then the shortest distance among all paths between v_i and v_j that pass the common node v_{k_m} . Therefore, Eq. 3.2 can obtain the shortest distance among all paths between v_i and v_j that pass any common node from v_{k_1} to v_{k_r} . \square*

Take Figure 3.1 as an example again. We want to calculate the shortest distance between node 0 from P_1 and node 8 in P_2 . There are three common nodes connected to both nodes 0 and 8, which are nodes 3, 4 and

5. $W(v_0, v_3) + W(v_3, v_8) = 26$ is the shortest distance among all paths between v_0 and v_8 that passes the common node v_3 . Similarly, the shortest distance among all paths between v_0 and v_8 that passes the common node v_4 is $W(v_0, v_4) + W(v_4, v_8) = 19$ while the shortest distance among all paths between v_0 and v_8 that passes the common node v_5 is $W(v_0, v_5) + W(v_5, v_8) = 18$. The minimum of these three figures give the shortest distance between v_0 and v_8 in the entire graph, which is 18.

Stage 5: Find all pair shortest paths for the entire graph

This stage is operating when the number of partitions more than two parts. Stage 4 calculates the path that crosses two neighbouring subgraphs. In stage 5, we find the shortest distances for the paths that cross more than two subgraphs (i.e., the source node is in a partition and the destination node is in not neighbouring partition). The equation that calculate this shortest path is presented and proved in the following theorem.

Theorem 3. *Assume $v_{i-1} \in P_{i-1}, v_i \in P_i, v_{i+1} \in P_{i+1}$ and $v_{k_1}, v_{k_2}, \dots, v_{k_r}$ is the set of common nodes that are connected to both v_i and v_{i+1} . Then the distance of the shortest path between v_{i-1} and v_{i+1} , $W(v_{i-1}, v_{i+1})$, can be calculated by Eq. 3.3.*

$$W(v_{i-1}, v_{i+1}) = \text{Min}\{W(v_{i-1}, v_{k_m}) + W(v_{k_m}, v_{i+1}) | 1 \leq m \leq r\} \quad (3.3)$$

Proof. v_{k_m} is the set of common node connecting the nodes in parts P_i and P_{i+1} . Then, $v_{k_m} \in P_i$ and P_i is neighbour with part P_{i-1} . The shortest distances between v_{i-1} and v_{k_m} can be calculated in Stage 4, and between v_{k_m}

and v_{i+1} can be calculated in Stage 3. The sum of them is then the shortest distance among all paths between v_{i-1} and v_{i+1} that pass the common node v_{k_m} . Therefore, Eq. 3.3 can obtain the shortest distance among all paths between v_{i-1} and v_{i+2} that pass any common node from v_{k_1} to v_{k_r} . \square

This stage repeated until the result between all graph partitions are combined. The number of iterations of this step is equal to $\sum_{i=2}^{P_i-1} P_i - i$ (where P_i is the total number of parts). After the last iteration terminated, we have the shortest path between any two nodes in the graph and the APSP problem is solved.

The parallelization method can be briefly outlined in the algorithm 1.

3.2.3 Reducing the number of operations

Reducing the number of operations required to process each partition can increase the performance of our method. In the steps of our approach, we noticed that in stages 3 3.2.2, a path has to be updated because we do not know the actual shortest path between the common nodes when we run the APSP algorithm on the subgraphs in stage 2 3.2.2. This means that in the worst-case scenario, we need to update all paths stored by partition P_i , the number of which equals to $n * (n - 1)$ where n is the total number of nodes in P_i . To avoid storing all paths, in stage 2 3.2.2 we compute the temporary shortest path only between the common nodes instead of between any nodes in the subgraph. Then, in stage 3 3.2.2 after we calculate the actual shortest path of the common nodes, we run the APSP algorithm on the subgraph. By doing this, since we have the actual shortest path between

Algorithm 1 The Parallel Algorithm for Finding the All-Pairs Shortest Paths

Input: A weighted graph $G(V, E, w)$, where V is the set nodes, E edges between nodes, and w is the weight of the edge.

Output: Result matrix of APSP in G

```

1 Run BFS algorithm on  $G$ ;
  Put the duplicated level between  $P_i$  and  $P_{i+1}$  in  $C_{i,i+1}$ 
  Let Value be a Matrix
  for ( $n$  in  $C_{i,i+1}$ ) do
2   for ( $m$  in  $C_{i,i+1}$ ) do
3     Find shortest path between  $n$  and  $m$ ;
     Add labeled edge;
4 for (each Part P) do
5   Run APSP algorithm ;
   Update Value;
6 for ( $x$  in  $C_{i,i+1}$ ) do
7   for ( $n$  in  $P_i$ ) do
8     for ( $m$  in  $P_{i+1}$ ) do
9       if  $Valu_{n,m} > Valu_{n,x} + Valu_{x,m}$  then
10        Update  $Valu_{n,m}$ 

```

3. Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture

the common nodes, we will have the actual shortest path between any nodes in the subgraph as explained in Theorem 1.

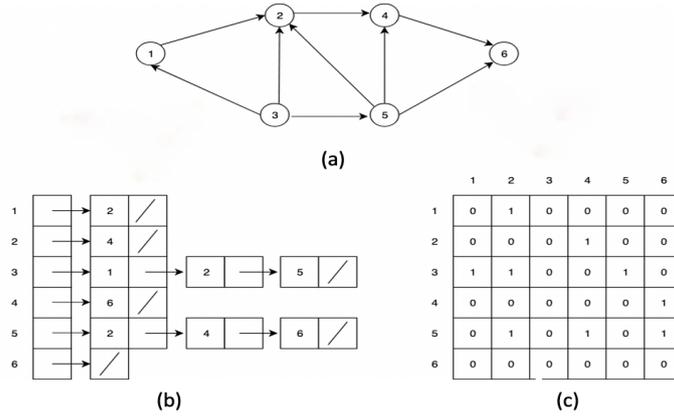


Figure 3.2: Graph Representation, (b): adjacency list. (c): adjacency matrix [94]

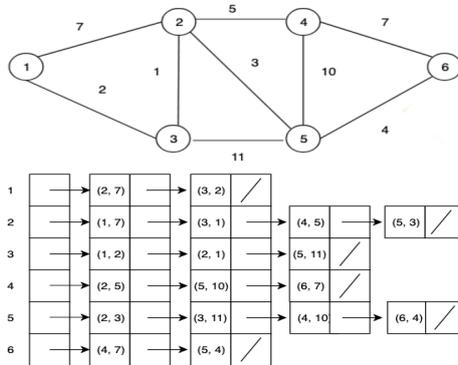


Figure 3.3: Graph Representation using adjacency list for a wight graph [94]

3.3 Graph Representation

How to store the graph being processed in memory plays an important role in achieving good overall performance. The most common way of representing a graph is the adjacency list and the adjacency matrix [24][90]. All

the work discussed in the literature review chapter and the algorithms we compare our method with in the experiments section use one of these two ways. In the work of this thesis, we store the graph using the adjacency list. The reason is because the adjacency list requires $O(V + E)$ space (where V is the number of vertices, and E is the number of edges), which is less than the storage space required by adjacency matrix, $O(V^2)$.

In the adjacency list representation, an array is used to hold a number of lists, which equals to the number of vertices in the graph. Each element represents a vertex in the graph and points to a list of vertices that are neighbours of the element. For example, in figure 3.2 element 3 of the array points to its neighbours which are 1, 2 and 5. In case of weighted graph, an element in the list is in the form of a pair. The first element is the neighbouring vertex while the second is the weight of the edge connecting the two vertices, which is shown in figure 3.3. In order to determine if a source vertex is connected to a destination vertex, we need to traverse through the list indexed by the source vertex and find the connectivity.

3.4 Parallel Implementation and Analysis

Four of the five steps in our method can be run in parallel, which are steps 2-5 in 3.2.2. In this section, we present how to run these steps in parallel. Starting with section 3.4.1 which contains the details of how to pre-process the graph and change the nodes ordering to help achieve an efficient parallel execution. In section 3.4.2, we present the data structure used to store the intermediate computing results during the graph processing. The OpenMP

paradigm is explained in section 3.4.3. Finally, in section 3.4.4 we present our method for reducing the memory consumption for storing the computing results and therefore enable the processing of bigger graphs.

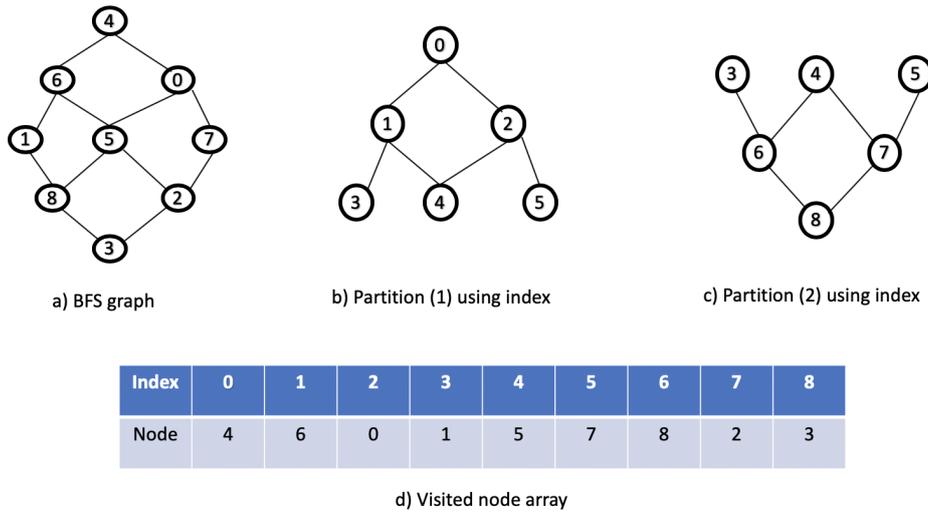


Figure 3.4: Reordering the nodes by renaming a node with its index.

3.4.1 Graph Preprocessing

Graph preprocessing is a crucial step to achieve an efficient parallel processing. We preprocess the graph by i) partitioning it into subgraphs and ii) rearranging the vertices in each subgraph.

For partitioning, the graph is divided into a number of parts (sub-graphs). Each part is assigned to be run on one core (by one thread). As discussed in stage 1 of our method 3.2.2, the Breadth-First-Search algorithm is used to achieve this task. One benefit of using BFS is that it can achieve the connectivity, i.e., all nodes in a subgraph are connected. Besides, BFS visits

3. Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture

Graph's name	The number of nodes in a level	The number of nodes that need to be duplicated
Western US Power Grid	258	183
out.ego-facebook	471	5
road-minnesota	56	40
SW-Trial	77	42
rt_israel.	732	200

Table 3.2: Example of number of nodes actually duplicated on real-world networks

all nodes by travelling all edges. Accordingly, we can track the paths. Then We can transform the graph into a graph with levels. Consequently, a node in level 0 has no direct connection to a node in level 2 and needs to pass through at least one node in level 1. Similarly, a node in level 0 needs to pass through at least one node in level 1 and one in level 2 to reach a node in level 3. This indicates that the shortest path between two nodes in different levels is the path with the minimum cost crossing the levels. This can be modelled in the following equation 3.4

$$W(v_{L_i}, v_{L_{i+2}}) = Min\{W(v_{L_i}, v_{L_{i+1}}) + W(v_{L_{i+1}}, v_{L_{i+2}})\} \quad (3.4)$$

We use this observation to choose the nodes in a level to be common nodes between two graph partitions. The common nodes can be regarded as the interface between two neighbouring partitions. Take the graph in figure 3.1 as an example, level 2 is duplicated, if node 1 in partition $P1$ wants to reach node 8 in partition $P2$, then the path contains at least one node from level 2, which is 4 in this situation.

Duplicating a node does not mean that we have to process it twice because

the edges of the node are not doubled. The only edges that are duplicated are those connecting the common nodes. Therefore, when processing this node, the number of operations is the same, but the nodes and edges in each subgraph are processed in parallel. Suppose that s is a node in the duplicated level i . It has five edges with two connecting to the nodes in level $i - 1$ and three to level $i + 1$. When we examine s , two edges will be processed in first subgraph and three in the second subgraph. The total number is five, same as when we process the entire graph as a single part. When a node in the level along which the graph is partitioned is not connected to the next level, this node will not be duplicated. When a graph is sparse, there are usually a limited number of nodes that are connected to the next level. Table 3.2 list the examples in real_world networks.

Graph preprocessing in this work also involves reordering the nodes in the partitions by renaming a node with its index. The idea is to assign the nodes to the partitions in the ascending order of index. Taking the graph in figure 3.4 as an example, we have 9 nodes named 0, 1, 2, ..., 8. After partitioning the graph without reordering the nodes, nodes (4,6,0,1,5,7) are assigned to P1 while nodes (1,5,7,8,2,3) assigned to P2. After reordering, the nodes are re-named by the order in which BFS visits the nodes, which we call the indices of the nodes. Then, P1 contains nodes (0,1,2,3,4 and 5), while P2 contains nodes (3,4,5,6,7 and 8). There is no extra cost for reordering since it is conducted while BFS runs. Besides, the information about the partitions is stored in an array. The size of this array is equal to the total number of partitions plus one, i.e., the size is $P + 1$. Each index points to a partition that the node belongs to and holds two values related to this

partition. The first value is the index of the first node in the partition, while the second is the total number of common nodes between this partition and the neighbouring partition. For the last index of the array, the first value is the size of the entire graph while the second value is zero.

The reordering method can improve the performance by saving enormous memory space. To understand the benefits of reordering, let us take a look at how to find the shortest paths between the nodes in different partitions (step 4 and 5). To find those paths, we need three pieces of data, the node in the one partition, the node in the other partition and the common nodes between the two partitions. The naive way is to store the nodes of each partition in a separate array, and store the common nodes between each pair of neighbouring partitions in another different array. This means that $P+P-1$ arrays are needed to store the information, which makes it inefficient to loop through the info. For example, in order to pick a source node from the first partition, we need to loop through the common nodes to make sure it is not one of the common nodes. It is similar for picking the destination node from the other partition. But now the nodes are in the ascending order after reordering. We can replace all these arrays with only one array, and use this array to find the information we need by using simple mathematical calculations. For example, if you need to find a node in partition P_i , then it is in the range of $(Ar[P_i][0], Ar[P_{i+1}][0])$. The common nodes between neighbouring partitions P_i and P_{i+1} are in range $(Ar[P_{i+1}][0], Ar[P_{i+1}][0] + Ar[P_i][1])$. Algorithm 3 describes how a thread uses the reordering array to find the shortest path between the nodes in different partitions.

The time complexity of graph preprocessing (3.2.2) is $O(v + E)$. It can be parallelized by running the breadth first search algorithm in parallel, regarding which there are many works such as the work presented in [58][16][8][43].

Algorithm 2 The Algorithm for Graph Preprocessing

Input: $BFS(G, s)$, where G is the weighted graph, s is the source node to start the BFS from.

Output: Indexing Array

```

1 Let  $q$  be a Queue;
  Index[ $V_i$ ] array with size equal to the total number of vertexes in  $G$ 
  count = 0;
  Add label to  $s = \text{count}$ ;
  Index.insert( $s, \text{count}$ );
  q.enqueue( $s$ )
  while  $q$  is not empty do
2   n = q.dequeue;
   for all neighbours  $w$  of  $n$  in Graph  $G$  do
3     count = n.lable;
     if  $w$  is not labled then
4       lable  $w$  with count+1;
       Index.insert( $w, \text{count}$ );
       q.enqueue( $w$ )

```

3.4.2 Storing the Result

The data structure we used to store the distance of the shortest path between the nodes is a 2D matrix. One advantage of using a matrix is that searching and updating the matrix are efficient with the time complexity

3. Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture

of $O(1)$. The matrix size is $N * N$, where N is the total number of nodes in the graph. A row or a column index corresponds to a node, while an element $[i, j]$ in the matrix holds the weight of the edge connecting node i and j . At the beginning, the elements in the diagonal of the matrix are filled with 0 (representing the distance from a node to itself), while other elements are filled with an infinity (∞) (representing the current distance between a node and another node). Since in this chapter we are considering the shared memory architecture, each CPU core can access the matrix and update the values. At the end of stage five of our method, the matrix will be filled with the distances of the shortest paths between any two nodes in the graph.

Algorithm 3 The algorithm of combining the local results

Input: Ar (the partitioning array with the size of $(P+1)$), M (the matrix holding the distances of the shortest paths between nodes in graph G)

Output: Result matrix [M] of APSP in G

```

1 parts = total number of partition
  Rank = partition rank(0 to parts-1);
  for  $x = Rank + 1$  to parts do
2   for  $i = Ar_{rank,0}$  to  $Ar_{rank+1,0}$  do
3     for  $j = Ar_{(x,0)+(x-1,1)}$  to  $Ar_{(x+1,0)+(x,1)}$  do
4       for  $c = Ar_{(x,0)}$  to  $Ar_{(x,0)+(x-1,1)}$  do
5         if  $M_{i,j} > M_{i,c} + M_{c,j}$  then
6            $M_{i,j} = M_{i,c} + M_{c,j}$ ;

```

3.4.3 OpenMP

The OpenMP parallel programming paradigm is used to implement running a sub-graph on a CPU core (by a thread) in parallel and synchronised the running of multiple threads. Every thread searches for the shortest path between the common nodes and all other nodes in a subgraph. The Dijkstra's algorithm is used to find the shortest path. The time complexity of this step is $O(E_i \log V_i)$ (where E_i is the total number of edges and V_i is the total number of vertices in a subgraph). During the searching for the shortest paths, the matrix is updated simultaneously by the threads. The number of times each thread updates the matrix in stage 2 3.2.2 is $C_i \times (C_i - 1)$, while the time complexity of stage 3 3.2.2 is $(V_i - C_i) \times ((V_i - C_i) - 1) + C_i \times (v_i - C_i)$. The number of operations in stage 4 3.2.2 and stage 5 3.2.2 can be modelled by the following equation, where C is the total number of common nodes between two parts and P_i is the total number of parts we divide the graph into.

$$\sum_{i=1}^{P_i} V_i \times C_{i,i+r} V_{i,i+r} |1 \leq r \leq P_i \quad (3.5)$$

As all threads process their subgraphs in parallel, the total time of our method is equal to the maximum time taken by a thread to finish all the five stages of the method.

3.4.4 3D Matrix As Data Structure

One disadvantage of the shared memory architecture is the limitation of its memory, which can restrict the size of the graph we can process. In the case of the APSP, the bigger the graph, the more memory space is needed

to store the results (distances of the shortest paths). In our method, a 2D matrix is used to store the results, which needs the memory space of $n^2/8$ bytes, where n is the total number of graph nodes. We propose a new data structure, which is a 3D matrix, to store the results. In this data structure, only the weights needed to calculate the distances of other paths are stored. With this new data structure, less data are stored. Consequently, the memory consumption is reduced, which allows us to process larger scale graphs.

In stages 4 and 5, the information required to find the shortest path between source node s and destination node t is the distance of the shortest path between s and a common node and that from the common node to t . Other data are not needed. For example, in figure 3.1 the shortest path between 0 and 1 is not needed when we want to find the shortest path from 1 to 8.

Therefore, we replace the 2D matrix ($N*N$) with the 3D matrix. The size of this 3D matrix is $c * n * 2$, where c (rows) is the total number of the common nodes between all parts, n (columns) is the total number of nodes in the graph, and the third dimension holds two values: i) the distance of the shortest path from the common node to the node $W(c, n)$, and ii) the distance from the node to the common node $W(n, c)$. In the case of the undirected graphs, we do not need the third dimension since the two values are the same. The reordering method presented in 3.4.1 is used to rearrange the common nodes. Instead of using one array for columns and rows, two arrays are used, one for columns which contain all nodes of the graph (from 0 to $n - 1$) while the other for rows which holds the common nodes only (from 0 to $c - 1$).

3.5 Evaluation

In this section, we evaluate the method presented in this chapter to solve the APSP problem. First, we compare the performance between the serial implementation and parallel implementation. We test the method with undirected and directed graphs of different sizes up to 128K. These graphs are partitioned into a number of subgraphs up to 16, which are processed on multiple cores (2, 4, 8 and 16). The graph generator presented in [49] is used to create the graphs for testing. The generated graphs follow the power-law distribution, which has been shown to be the property of most of the real-world networks. Next, we compare our method with two existing APSP algorithms, n-Dijkstra [26] and ParAPSP [57]. The n-Dijkstra algorithm parallelizes the traditional Dijkstra algorithm by running the algorithm from multiple nodes in parallel. The nodes of the graph are divided among the cores and are processed simultaneously until the shortest paths between all nodes are found. The ParAPSP algorithm is one of the latest works for solving the APSP problem in the shared memory architecture. It is a modified Dijkstra’s algorithm, which utilizes the information obtained in previous iterations during the processing and uses an ordering approach based on node degree to reduce the parallel overhead. Finally, we evaluate our method with even larger-scale graphs using the 3D matrix.

We conducted the experiments on a high-performance cluster called Orac hosted in the Centre for Scientific Computing at the University of Warwick. The cluster mainly consists of 84 Lenovo NeXtScale nx360 M5 servers, 128

3. Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture

GB 2400 MHz DDR4 memory and 28 cores per node [77]. The network fabric is Intel Omni-Path X16 with the bandwidth of 100Gbit/s. Throughout this evaluation, we denote our method we presented by CNA (Common Nodes Algorithm).

3.5.1 Speedup of our parallel implementation over the serial implementation

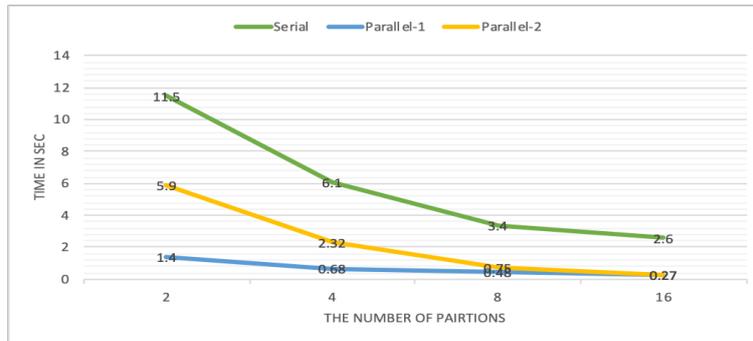


Figure 3.5: The performance of our two parallel implementations comparing to the serial implementation; The graph size is 16K

We first run the serial implementation of CNA, in which after the graph is partitioned, the shortest paths between the nodes in all the subgraphs are found in sequence and the aggregation of local results in each subgraph is conducted in sequence. We run the parallel CNA in two different ways: i) the graph is partitioned into a fixed number of subgraphs, and then the subgraphs are assigned to a different number of cores for processing; ii) the number of subgraphs that the graph is partitioned into is the same as the number of cores in the multicore computer, and then each subgraph is assigned to a core.

Figure 3.5 shows the comparison between the serial implementation and the two parallel implementations of our CNA.

The graph size used in this experiment is 16K. The graph is partitioned into 2, 4, 8 and 16 subgraphs. When the graph is partitioned, the cost of finding the APSP is reduced significantly. The cost of finding shortest paths between the nodes in different partitions are replaced with the cost of aggregating the local results in subgraphs, which is much lower. As the result, we can observe that the more partitions, the faster the algorithm is for solving the APSP problem. Figure 3.6 (a) shows the performance of serial CNA and parallel CNA as the number of partitions increases.

Parallel-2 in figure 3.5 is the parallel CNA when the subgraphs are processed simultaneously on multiple cores. It can be seen from figure 3.6(b) that our parallel CNA delivers the outstanding performance, comparing to the serial CNA.

In parallel-1 in this experiment, the number of partitions is fixed to 16, which are then processed on a different number of cores (2, 4, 8 and 16). The results are shown in figure 3.5. We can see from this figure that it is faster than parallel-2 when the number of cores is small. It catches up as the number of partitions increases.

3.5.2 Evaluating CNA with different parameter settings

In this subsection, we evaluate parallel CNA with different settings for graph size and the number of partitions and compare it with the n-Dijkstra

3. Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture

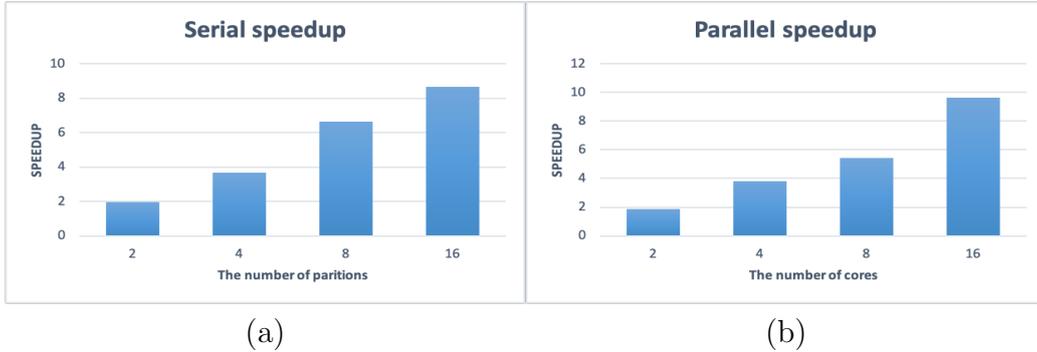


Figure 3.6: Speedup of serial and parallel CNA, Graph size is 16K

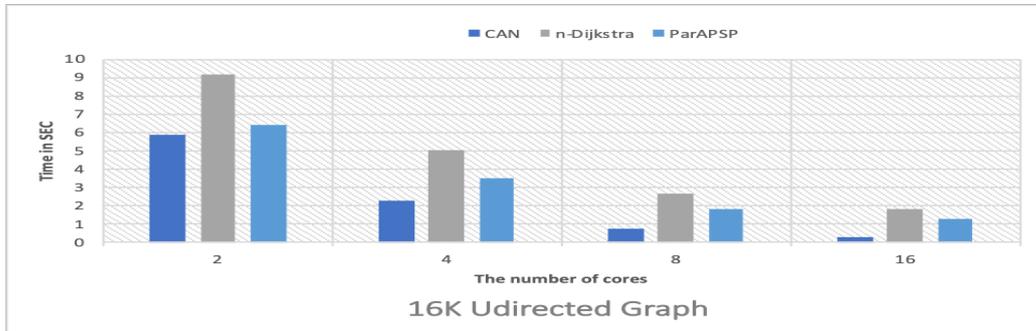
algorithm and the ParAPSP algorithm. The graph size is 2K, 4K, 8K and 16K. The graph is divided into up to 16 subgraphs.

Figure 3.7 shows the experimental results with undirected graphs. We can observe from this figure that parallel CNA achieves the outstanding performance with different sizes of graphs, comparing to n-Dijkstra and ParAPSP. It solves the APSP problem for the graph of 2K size in about 0.06 seconds, which is 2.5x faster than ParAPSP and 3.5x faster than n-Dijkstra. Besides, for big graphs, it takes about 0.3 seconds to process the 16k graph. We observed the similar results when conducting the experiments with directed graphs. The results are shown in figure 3.8.

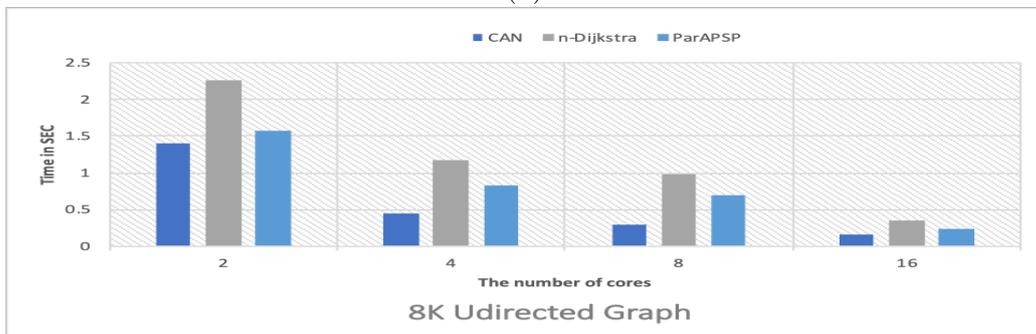
An important factor that affects the performance of parallel CNA is the number of subgraphs that we partition the graph into. Unlike n-Dijkstra and ParAPSP, after the graph partitioned the computation cost for the steps of finding the shortest paths among nodes in the same partition is reduced in the CNA. Consequently, the more partitions the graph is divided into, the

better performance parallel CNA can deliver. We can observe from figure (a) 3.7 that when the number of partitions is equal to two, parallel CNA is about 1.5x faster than n-Dijkstra and 1.08x faster than ParAPSP. When we partition the graph further into 16 subgraphs, it is 6.4x faster than n-Dijkstra and 4.5x faster than ParAPSP. Accordingly, for CNA_parts in figure 3.9, we partitioned the graph into a fixed number of subgraphs (16) and ran it on different number of cores. We can see that the excellent performance is achieved by CNA_parts in a small number of cores (2 and 4).

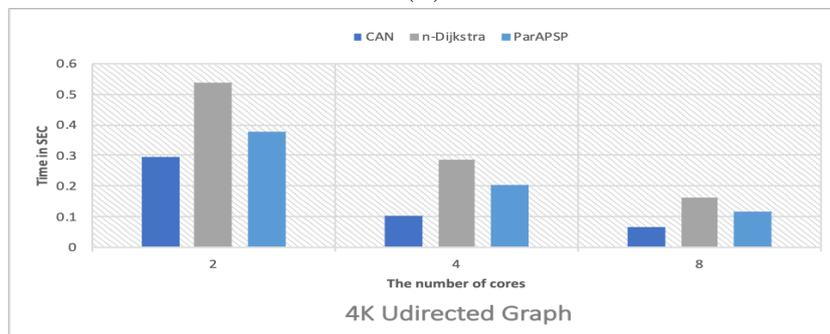
3. Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture



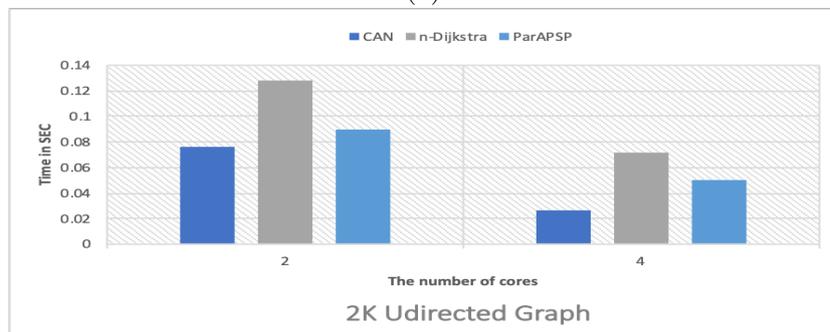
(a)



(b)



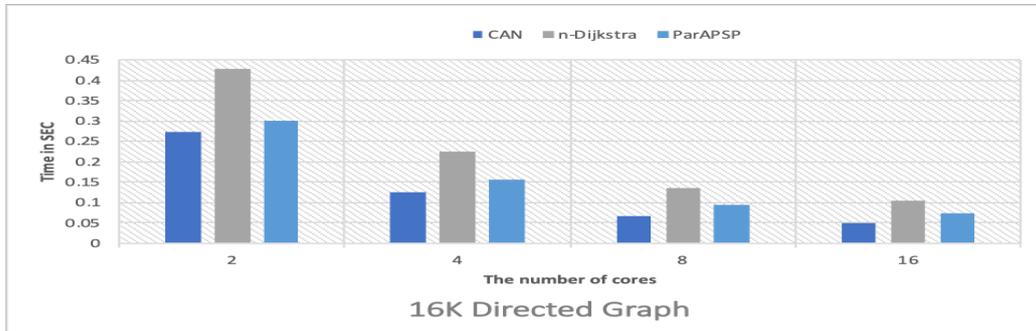
(c)



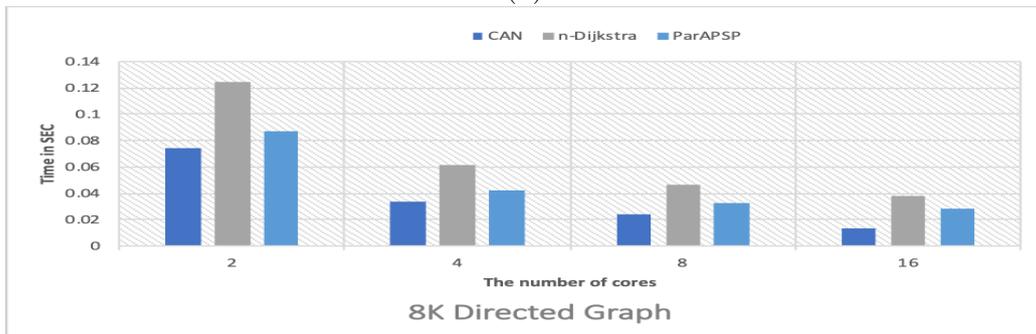
(d)

Figure 3.7: Comparing CAN, n-Dijkstra and ParAPSP on undirected graph

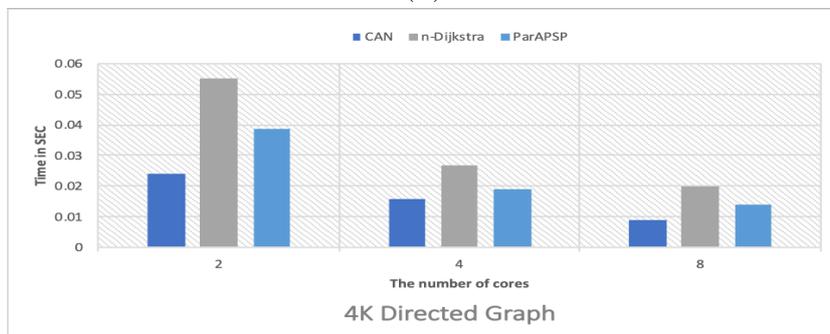
3. Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture



(a)



(b)



(c)

Figure 3.8: Comparing CAN, n-Dijkstra and ParAPSP on directed graph

3. Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture

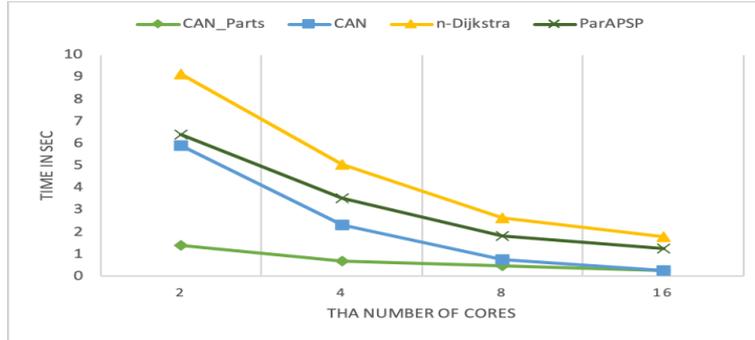
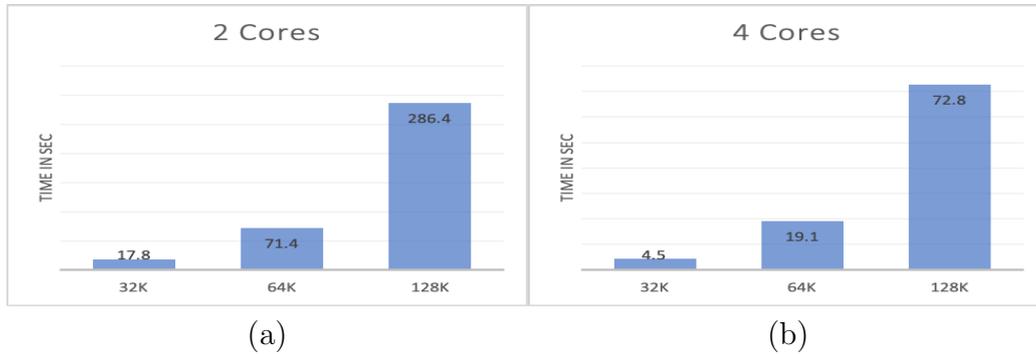


Figure 3.9: The performance of the two versions of parallel CNA. The graph size is 16K

3.5.3 Evaluating the parallel CNA with larger scale graphs

In this subsection, we test the ability of our method in processing big graphs by using the 3D matrix as the data structure to store the results. We increase the graph size from 32K, 64K to 128K and show the results in figure 3.10. We can see that our method solves the APSP problem using 16 cores in about 0.5, 1.3 and 5.4 seconds for the graph size of 32K, 64K and 128K, respectively.



3. Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture

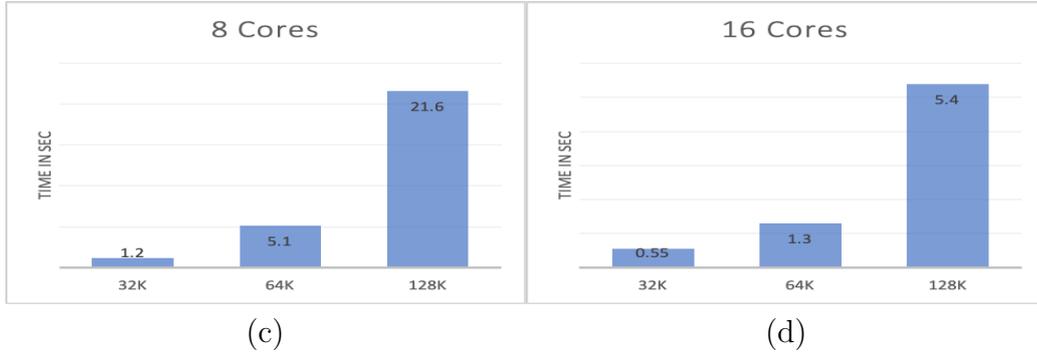


Figure 3.10: Experiment on big graphs

3.6 Summary

In this chapter, we presented a new method to find the APSP of sparse graphs on a shared memory architecture. In the method, we partition the graph into smaller subgraphs and compute the APSP for each subgraph in parallel, and then aggregate the local results. The BFS algorithm is used to determine the common nodes between neighbouring subgraphs. The common nodes are duplicated, which can reduce the computation of APSP significantly and consequently improve the overall performance. The proof of the method correctness is given. Further, openMP is used to schedule the work among the CPU cores. Although we design this method for the shared memory architecture in this chapter, we still manage to process big graphs by designing a data structure (3D matrix) to store the results, which reduces the memory consumption significantly. We conducted the experiments on a high-performance cluster and evaluated our method with different experimental settings. The experiment results show that our method is up to 6x

3. Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture

faster than the existing methods in literature.

CHAPTER 4

Developing the Parallelization Methods for Finding the All-Pairs Shortest Paths in Distributed Memory Architecture

4.1 Introduction

Distributed memory architecture such as a cluster is another mainstream parallel processing architecture. Communication cost is an important factor that affects the performance of parallel applications running in distributed systems. When dependent tasks are processed on different processors in a distributed memory computing system, the processors need to communicate with each other to obtain the information needed to progress further. If the communication cost is high, it will cancel the advantage of running the application in parallel. In the APSP problem, the edges between nodes represent the dependent relation between nodes. How the graph is partitioned and the way in which the processors communicate with each other will have significant impact on the communication performance. In this chapter, we address this problem and propose an efficient method to solve the APSP problem in distributed memory architecture.

In this method, the graph is partitioned into subgraphs. Each subgraph is assigned to a processor and all-pairs shortest paths in each subgraph computed in parallel. Next, the local results in sub-graphs are combined to find all pairs shortest paths in the entire graph. We propose two communication patterns for the sub-graphs to propagate the local results. The communication patterns are implemented using Message Passing Interface (MPI). Further, we modelled the computation time and communication time with these two communication patterns. Moreover, we conducted experiments to verify the effectiveness of the communication patterns and the proposed method.

4.1.1 Motivation

In chapter 3, we presented a method to solve the APSP on a shared memory environment. Shared memory architecture such as a multicore machine is a mainstream parallel processing architecture. The main advantage of the shared memory architecture is the fast communication between the processing cores. However, the scale of the graphs that can be processed by a shared memory machine is limited by the memory size and the number of cores in the machine. In order to alleviate the resource limitation in a single shared memory computer, we develop a parallelization scheme for the distributed memory architecture such as a cluster to solve the APSP problem. By adding more machines, our method can process even larger scale graphs and achieve a much higher degree of parallelism.

The remainder of this chapter is organized as follows. In Section 4.2 we present our distributed method in detail including the two communication patterns. Then we analyze the method mathematically in Section 4.3. Section 4.4 presents the experiments and discusses their results. Finally, this chapter is concluded in Section 4.5.

4.2 Solving APSP in distributed memory architecture

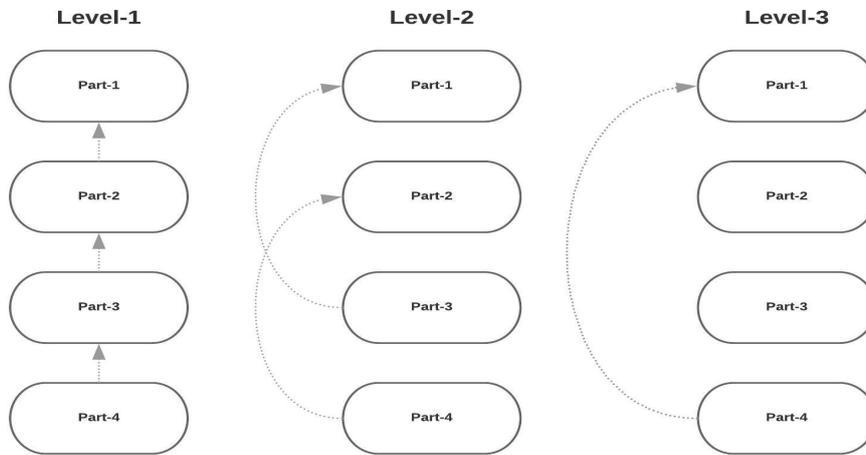
In this section, we detail the method for solving the APSP problem in distributed memory architecture, which is extended from the method developed for shared memory architecture (presented in Chapter 3).

4.2.1 Computing Operations

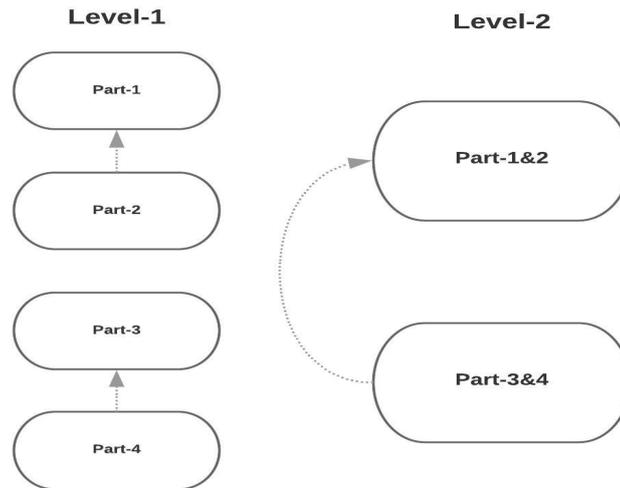
In this method, the graph is preprocessed in the same way as that for shared memory architecture. Namely, the graph is transformed to levelled graph and then partitioned into subgraphs along the nodes at certain levels. The common nodes between neighbouring subgraphs are duplicated. Each subgraph is then allocated to a processor in the cluster, and handled by a process in parallel.

When finding the APSP for the allocated subgraph on a processor, the process creates a *distance matrix* with size $n * m$ to hold the distance of the found shortest paths, where n is the total number of nodes in the subgraph assigned to the processor, and m is the total number of nodes in the whole

4. Developing the Parallelization Methods for Finding the All-Pairs Shortest Paths in Distributed Memory Architecture



(a) First pattern



(b) Second pattern

Figure 4.1: A case study for the combination of local results with the two communication patterns

graph. A process finds the shortest paths between any two nodes in the local subgraph, and then exchanges the distances of the shortest paths between common nodes with its neighbouring processes. The exchanged information will be used to combine the local distances in each subgraph and obtain the

distances of the shortest paths for the entire graph.

After process A receives the distance of shortest paths between common nodes from a neighbouring process (e.g., process B), A compares them with the distance of shortest paths between common nodes calculated locally by process B, and then updates the distances between common nodes by the smaller values in the distance matrix. When a process updates any value in the distance matrix, it adds an edge in the graph connecting the corresponding nodes, and the updated value is assigned as the edge weight. After the above steps are completed, the distance matrix has the distances of the shortest paths between any two nodes in the same subgraph.

Next, we need to find the shortest paths between the nodes in different subgraphs. In order to achieve this, the processes propagate their local results to other processes. Assume v_i is the source node in subgraph A, v_j is the destination node in subgraph B, and v_{C_m} is the common nodes between subgraphs A and B. When the process handling subgraph A receives the local results from the process handling subgraph B, it applies the following equation to update the distance matrix. Where $W(v_i, v_{C_m})$ is the distance of the shortest path that is calculated locally in subgraph A and $W(v_{C_m}, v_j)$ is the value received from subgraph B.

$$W(v_i, v_j) = \text{Min}\{W(v_i, v_{C_m}) + W(v_{C_m}, v_j) | 1 \leq m \leq r\} \quad (4.1)$$

After propagating the local results, the matrix will eventually have the solution of the APSP problem for the entire graph.

4.2.2 Communication Pattern among processes

When running an application in a distributed memory architecture, we need to consider the communication cost and reduce it as much as possible. There are different ways for a process to propagate the local information to other processes, which are called *communication patterns*.

An intuitive communication pattern is to make one process act as a master, and all other processes send their local results to the master. The master process combines the local results received to find the all-pairs shortest paths for the entire graph (using the step of combining the results in chapter 3). This intuitive communication pattern is good in terms of reducing the communication cost since a process sends only one message. However, with this communication pattern, the operations for finding the shortest path between the nodes in different subgraphs are run in sequence by the master process and lose the parallelization opportunity provided by the cluster. Another straightforward pattern is to make every processor send its local results to all other processors. However, there will be an excessive number of messages transmitted on the network, which will be costly.

In this chapter, we propose two other communication patterns for the processes to propagate the local results. Figure 4.1 illustrates these two communication patterns. The communication patterns operate in rounds. In round 1 of the first pattern, a process with the rank k sends its local results to the process directly above it, i.e., the process with the rank of $k - 1$. In the next round, process k sends its local results to process $k - 2$. The propagation continues until the local results of last process have been

sent to the first process (with rank 0). For example, in figure 4.1 (a), there are four processes ranked as 1, 2, 3 and 4. In the first round, process 4 sends its local results to 3. At the same time, process 3 sends the local results to 2, so does process 2 to 1. In the next round, process 4 sends its results to process 2 while process 3 sends its results to process 1. In the last round, process 4 sends its results to process 1.

When a process receives the local results sent by another process, it finds the shortest path between the nodes in these two different subgraphs. Algorithm 4 outlines the steps of sending and receiving local results with pattern 1

In the second communication pattern, the processes are divided in groups. Each group consists of two processors with one acting as the sender while the other as a receiver. The receiver finds the shortest paths between the nodes in these two different subgraphs in the same group. In next round, a new group is established, which consists of the two receivers from two groups in the previous round. One receiver sends the local results obtained from previous round to the other receiver. The receiver will find the shortest paths between the nodes in the two groups in this round. For example, in figure 4.1 (b), there are four processes ranked from 1 to 4. In the first round, processes 4 and 3 are in one group, while processes 2 and 1 are in another group. Process 4 sends its local results to 3. Process 3 combines the local results of these two processes to find the shortest paths between the nodes in these two different subgraphs. At the same time, process 2 sends its results to 1. Process 1 combines the local results and finds the shortest paths between the nodes in

Algorithm 4 Pseudo-code for sending and receiving local results with communication pattern 1

Input: $[Arr_s]$ the array holding the shortest path values

Output: APSP matrix

```
1 Let  $Arr_s$  be the array processors exchange;
   Let  $size$  be the total number of processors;
   Let  $rank$  be the the processors rank;
   MPI_Comm_rank(MPI_COMM_WORLD,rank);
   MPI_Comm_size(MPI_COMM_WORLD,size);
   % sending loop
   for ( $p = 0; p < size - 1; p ++$ ) do
2   |   if  $rank > p$  then
3   |   |   MPI_Bsend( $Arr_s, (rank - (p + 1)) \% size, MPI\_COMM\_WORLD$ );
4   |   |
4   % Receiving loop
   if  $rank < size - 1$  then
5   |   for ( $p = rank + 1; p < size; p ++$ ) do
6   |   |   MPI_Recv( $Arr_s, p, MPI\_COMM\_WORLD$ );
   |   |
```

4. Developing the Parallelization Methods for Finding the All-Pairs Shortest Paths in Distributed Memory Architecture

the subgraphs handled by processes 0 and 1. In the next round, process 3 sends the results (including the results received from process 4) to process 1, which finds the shortest paths between 1 and 3, 1 and 4, 2 and 3 and 2 and 4. The rounds continue until the local results of all processes have been combined to find the shortest paths between any two nodes in the entire graph. Algorithm 5 outlines the steps of sending and receiving results with pattern 2

The two communication patterns are analyzed in more details in section 4.3.

Algorithm 5 Pseudo-code of communication pattern 2 sending and receiving

Input: $[Arr_s]$ the array holding the shortest path values

Output: APSP matrix

```

1 Let  $Arr_s$  be the array processors exchange;
  Let  $size$  be the total number of processors;
  Let  $rank$  be the the processors rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  int  $group = 2$ ;
  while ( $group \leq size$ ) do
2   | % Sending
3   |   if  $rank \% group == (group/2)$  then
4   |   | MPI_Bsend( $Arr_s, (rank - (group/2)), MPI\_COMM\_WORLD$ );
5   |   |    $group = group * 2$ ;
   |   |
   |   | % Receiving
   |   | else if  $rank \% group == 0$  then
   |   |   MPI_Recv( $Arr_s, (rank + (group/2)), MPI\_COMM\_WORLD$ );
   |   |    $group = group * 2$ ;
   |   |
   |   |

```

4.2.3 Reducing Message Size

In this section, we aim to reduce the size of the exchanged messages between processes. As discussed in previous subsection, we should reduce the communication cost to achieve excellent overall performance in distributed memory environments. One of the main factors that affect the communication cost is the size of the messages being transferred between processes.

The message sent by a process contains the distances of the shortest paths between the nodes in the subgraph that the process is handling. The receiver uses this information to find the shortest path between nodes in two different subgraphs. A message sent by a process should contain $n * n$ elements, where n is the number of nodes in the process' subgraph, and $n * n$ is the number of node pairs (hence the number of shortest paths) in the subgraph. Each element carries three pieces of information, which is source node ID, destination node ID and the distance of the shortest path between source and destination nodes. Therefore, the size of the message is $3 * (n * n)$. However, if we look at equation 4.1, we can see that the information required to find the shortest path between source node s in one subgraph and destination node t in a different subgraph is the distance of the shortest path between s and c and that between c and t , where c is the common nodes between the two subgraphs. Therefore, the process does not need to send the information of all shortest paths in a subgraph, but only send the information of the shortest path between common nodes and other nodes in the subgraph. Therefore, the total number of elements in the message can be reduced to $c * n$, where $|c|$

is the number of common nodes connecting the two subgraphs, and the total message size is reduced to $3 * |c| * |n|$. This is a massive reduction since the number of common nodes is typically much less than the number of nodes in a subgraph. Algorithm 6 outlines how a process decodes the message received from another process, namely uses the information contained in the received message to update its local distance matrix.

Algorithm 6 Pseudo-code of decoding the received message

Input: $[M_r]$ the distance matrix in a processor; and $[Arr_s]$ the array processors exchange;

Output: APSP matrix $[M_r]$

```

1 int size;
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  if rank < size then
2   MPI_Recv(Arr_s, p, MPI_COMM_WORLD);
   for (x in Arr_s) do
3   |   M_r[Arr_s[x][0], Arr_s[x][1]] == Arr_s[x][3];
   |
   |

```

4.3 The Method Analysis

In this section, we analyze our method for distributed memory architecture presented in the last section. Moreover, we present the details how the processes combine the local results communicated with the two proposed communication patterns and find the all-pairs shortest paths for the entire graph.

In the Stage of finding the all-pairs shortest paths in each subgraph in parallel, any all-pairs shortest algorithm can be used. For example, if the Dijkstra's Algorithm is used, the time complexity of finding the all-pairs shortest paths in a subgraph is $O(n_i^2)$, where n_i is the number of nodes in partition p_i .

After all-pairs shortest paths in a partition are calculated by a process, the process needs to exchange the information of the shortest paths between common nodes with the neighbouring partition. Therefore, in this stage, in order to calculate the shortest path between the nodes in two neighbouring partitions (Eq. 4.1), one partition needs to send its information of the shortest paths (i.e., $W(v_i, v_{k_m})$ in Eq. 4.1 if v_i is in the partition) to another partition (assume the partition where v_j in Eq. 4.1 is located). We presented two communication patterns to propagate location information. We now analyze both of the communication patterns in detail.

4.3.1 Communication Pattern 1

As discussed in previous section, the propagation of local results is conducted in rounds. In the first round, every partition P_i sends its local results to P_{i-1} in parallel, the process handling P_{i-1} combines the local results in partitions P_i and P_{i-1} . In this round, the number of messages transmitted between partitions (processes) is $P - 1$ (P is the number of partitions). In the second round, every partition P_i sends its local results to P_{i-2} in parallel in order to combine the local results in partitions P_i and P_{i-2} . The number of messages transmitted is $P - 2$. The pattern continues until in the last

round, the last partition P_P sends its local results to the first partition P_1 , and then local results in all partitions are combined. In pattern 1, the number of rounds is $P - 1$. The number of nodes in each partition is n_i , and the number of common nodes between P_i and P_{i-1} is denoted by c_i . The size of the message sent by partition i is $c_i \times (n_i - c_i)$, i.e., the number of paths in a partition which needs to be sent to the neighbouring partition. Since all messages in the same round can be transmitted in parallel, the total communication time can be modelled by expression 4.2, where t_m is the time for sending a message with the unit size.

$$\max\{t_m \times c_i \times (n_i - c_i) \times (P - 1) | 1 \leq i \leq P\} \quad (4.2)$$

The maximum number of (v_s, v_t) pairs, where v_s is in partition P_i while v_t is in partition P_{i+1} , is $n_i \times n_{i+1}$. For each (v_s, v_t) pair, we need to try all possible common nodes to determine the shortest path between v_s and v_t . Therefore, in this stage, the maximum number of operations needed to combine the local results of two neighbouring partitions P_i and P_{i+1} can be calculated by $n_i \times |C_{i,i+1}| \times n_{i+1}$. Every two neighbouring partitions P_i and P_{i+1} can combine the local results in parallel. Therefore, we can determine the number of operations (i.e., the operation of calculating $W(v_i, v_{k_m}) + W(v_{k_m}, v_j)$ in Eq. 4.1) performed in $level_i$ (denoted by $R(level_i)$) of pattern

1 as follows.

$$\begin{aligned}
 R(level_1) &= \max\{n_i \times n_{i+1} \times C_{i,i+1} | 1 \leq i \leq P-1\} \\
 R(level_2) &= \max\{n_i \times n_{i+2} \times C_{i,i+2} | 1 \leq i \leq P-2\} \\
 &\quad \dots \quad \dots \\
 &\quad \dots \quad \dots \\
 R(level_{P-1}) &= n_i \times n_{i+P-1} \times C_{i,i+P-1}
 \end{aligned} \tag{4.3}$$

Therefore, the computation time of combining local results with pattern 1 can be modelled by expression 4.4, where t_c is the time for performing one operation.

$$\max\{t_c \times \sum_{i=1}^{P-1} R(level_i) | 1 \leq i \leq P\} \tag{4.4}$$

Combining Eq. 4.2 and Eq. 4.4, we can obtain the execution time for stages 4 and 5 with pattern 1 (denoted by T_{ptn_1}), which is

$$T_{ptn_1} = t_m \times c_i \times (n_i - c_i) \times (P-1) + t_c \times \sum_{i=1}^{P-1} R(level_i) \tag{4.5}$$

4.3.2 Communication Pattern 2

Figure (b) 4.1 shows communication pattern 2 used to propagate the local results. In this pattern, every two subgraphs combine their local results and form a bigger subgraph in each level. The size of a partition double as the computation moves to the next level. As shown in the case study in figure (b) 4.1, in the first level, part-1 combines with part-2 while part-3 combines with part-4. In the second level, the newly combined part-3&4 combines

with the new part-1&2. In level i , the size of the partition is $2^{i-1} \times n_i$. In pattern 2, the number of levels is $\log_2 P$. Similar as Eq.4.3, the number of operations in level i of pattern 2 can be modelled by Eq. 4.6.

$$R(\text{level}_i) = \max\{(2^{i-1} \times n_i)^2 \times C_{2^{j+1}, 2^{j+1}} \mid 0 \leq j \leq \frac{P}{2^i} - 1\} \quad (4.6)$$

The total computation time of stages 4 and 5 with pattern 2 can be modelled by expression

$$\max\{t_c \times \sum_{i=1}^{\log_2 P} R(\text{level}_i) \mid 1 \leq i \leq P\} \quad (4.7)$$

Although the number of levels in pattern 2 is much less than that in pattern 1, the message size in each level increases dramatically in pattern 2 (while in pattern 1, the message size remains the same in each level). The message size sent by each partition (process) is $c_i \times (n_i - c_i)$ in the first level while the second level the message size become $c_i \times (2n_i - c_i)$. In general, the message size sent by each partition in $c_i \times (2^{i-1} \times n_i - c_i)$. Then the total communication time can be modelled by Eq. 4.8.

$$\max\{t_m \times \sum_{i=1}^{\log_2 P} c_i \times (2^{i-1} \times n_i - c_i) \mid 1 \leq i \leq P\} \quad (4.8)$$

So the execution time for combining the local results with pattern 2 (denoted by T_{ptn_2}) can be modelled by Eq. 4.9.

$$T_{ptn_2} = t_c \times \sum_{i=1}^{\log_2 P} R(\text{level}_i) + t_m \times \sum_{i=1}^{\log_2 P} c_i \times (2^{i-1} \times n_i - c_i) \quad (4.9)$$

4.4 Experiments and Results

In this section, we will present the experimental results to verify the effectiveness of our method. We first present the experimental settings including the graphs and the supporting platform, and then conduct the experiments to compare the two communications patterns and evaluate our methods with large scale graphs.

Our parallelization method supports any sequential all-pairs shortest path algorithm (to find APSP in a subgraph). The algorithm used in this evaluation is Dijkstra’s algorithm. In our experiments, since many real-world graphs follow the power-law distribution [2][96], we used the power-law-based graph generator [49] to generate the synthetic graphs. We generated the graphs with different sizes, ranging from 1K nodes to 128K nodes. We conducted the experiments on our high performance cluster called Orac. The cluster mainly consists of 84 Lenovo NeXtScale nx360 M5 servers, 128 GB 2400 MHz DDR4 memory per node [77]. The network fabric is Intel Omni-Path X16 with the bandwidth of 100Gbit/s. We partitioned a graph and ran different partitions on different computing nodes of Orac in parallel. The two proposed communication patterns are tested. MPI (Message Passing Interface) is used to implement our parallelization method on the cluster.

4.4.1 Comparing Our Method With the n-Dijkstra’s Algorithm

In this subsection, we compared our method with n-Dijkstra’s algorithm [99]. n-Dijkstra’s algorithm is an existing algorithm that can be implemented

in distributed memory architecture. This algorithm uses multiple processors to run the single source shortest path algorithm in parallel. Each processor processes a subset of graph nodes, finding the shortest path between each node in this subset and any node in the graph.

However, one drawback of n-Dijkstra's algorithm is that since a process needs to find the shortest path between a node and any node in the graph, every process needs to hold the entire graph. Therefore, although this algorithm can be easily implemented in the distributed memory architecture, it has the same limitation as the parallel APSP method for shared memory architecture, i.e., the scale of the graph that the n-Dijkstra's algorithm can process is limited by the memory size of each processor in the distributed memory architecture.

On another note, since in n-Dijkstra's algorithm each processor finds the shortest path between a node in its local node subset and any node in the graph, there is no such a stage of combining the local results as in our method. Even so, it can be seen from the experimental results presented in this section that our method outperforms n-Dijkstra's algorithm.

Figure 4.2 shows the comparison between our method and the n-Dijkstra algorithm. As seen from Figure 4.2, our method performs much better than n-Dijkstra when the number of partitions is small. When the number of partitions increases, the performance of n-Dijkstra's algorithm catches up with the performance of our method. This is because n-Dijkstra's algorithm does not have the communication cost as explained above.

4. Developing the Parallelization Methods for Finding the All-Pairs Shortest Paths in Distributed Memory Architecture

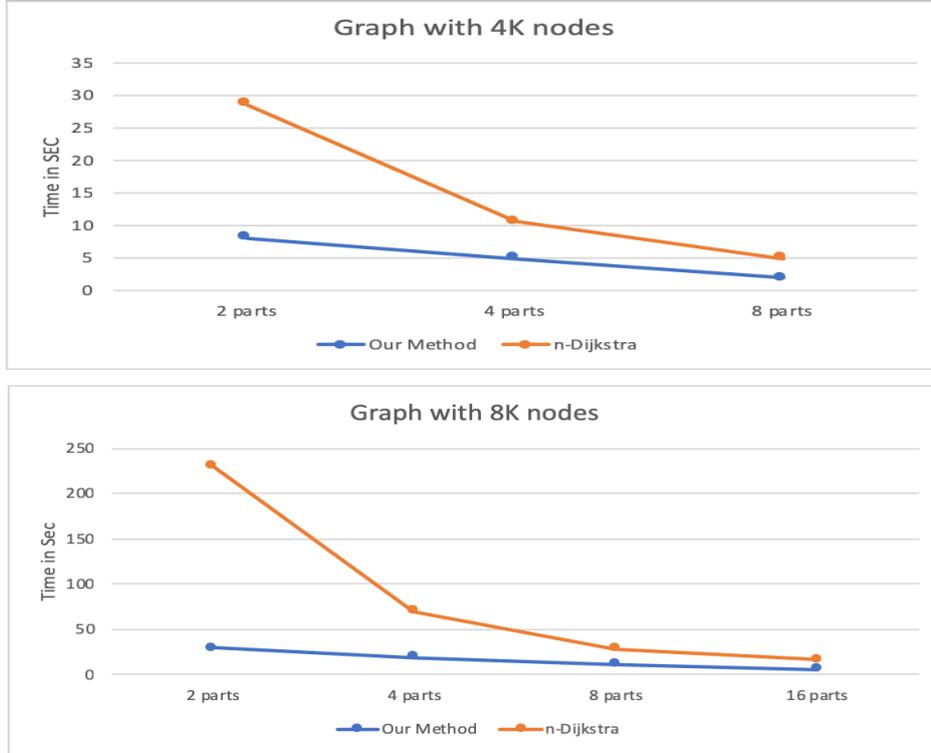


Figure 4.2: Comparing our method with the n-Dijkstra algorithm

4.4.2 Communication Patterns Comparison

Figure 4.3 shows the execution time of our method with two communication patterns as we increase the number of partitions. The results show that in all cases, our method with communication pattern 1 demonstrates much better performance than pattern 2. As the number of partitions increases, the performance difference becomes more prominent.

Table 4.1: Communication and combination times (in milliseconds) with pattern 1 and pattern 2; the graph with 8K nodes are divided into 8 partitions

Progress levels	Pattern 1							Pattern 2		
	Level-1	Level-2	Level-3	Level-4	Level-5	Level-6	Level-7	Level-1	Level-2	Level-3
Communication Time	0.7	0.7	0.7	0.7	0.7	0.7	0.7	0.7	1	4
Combination Time	1200	1200	1200	1200	1200	1200	1200	1200	5200	16900

4. Developing the Parallelization Methods for Finding the All-Pairs Shortest Paths in Distributed Memory Architecture

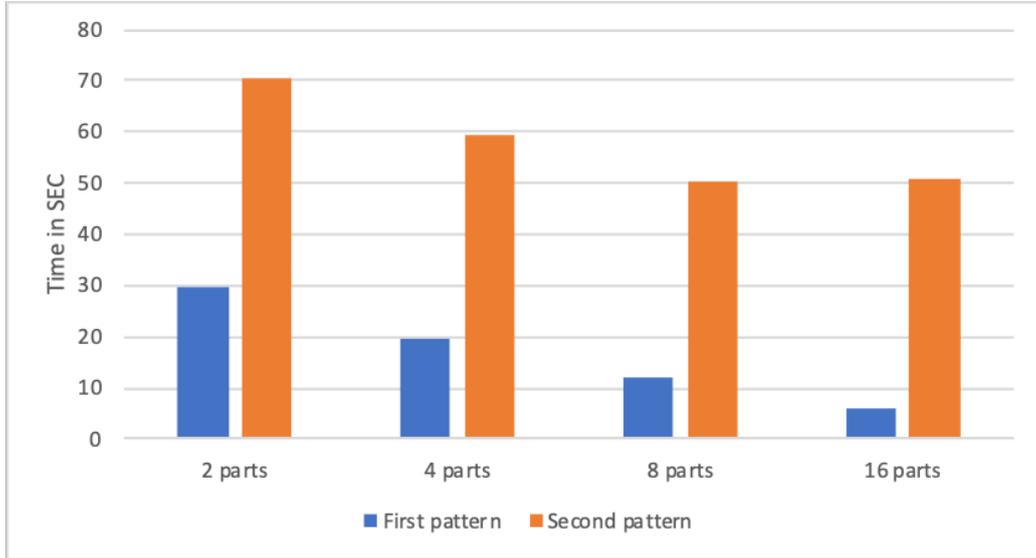


Figure 4.3: Comparing communication patterns 1 and 2; the number of graph nodes is 8k.

It is not expected that pattern 2 shows poorer performance than pattern 1 since pattern 2 goes through a less number of rounds (levels) than pattern 1 to combine local results in processes as analyzed in Section 4.3. We conducted further analysis regarding this. Table 4.1 shows the comparison between pattern 1 and 2 in terms of communication time and combination time. the combination time is the time spent in combining local results in neighbouring partitions as shown in equation 4.4 for pattern 1 and equation 4.7 for pattern 2. It can be seen from this table that with pattern 2, the communication time and combination times increase dramatically as the algorithm progresses along with the levels, while they remain almost constant with pattern 1. There are two main reasons why pattern 2 performs poorly comparing with pattern 1. First, in pattern 2, the message size increases dramatically as the combination of local results goes on. The sharp increase

in the communication cost slows down the progress. Second, as the algorithm progresses to the next round, the number of partitions decreases by half, which means that the degree of parallelism decreases dramatically as the algorithm progresses. It can be seen from these results and analyses that pattern 2 is not suitable for large-scale graphs.

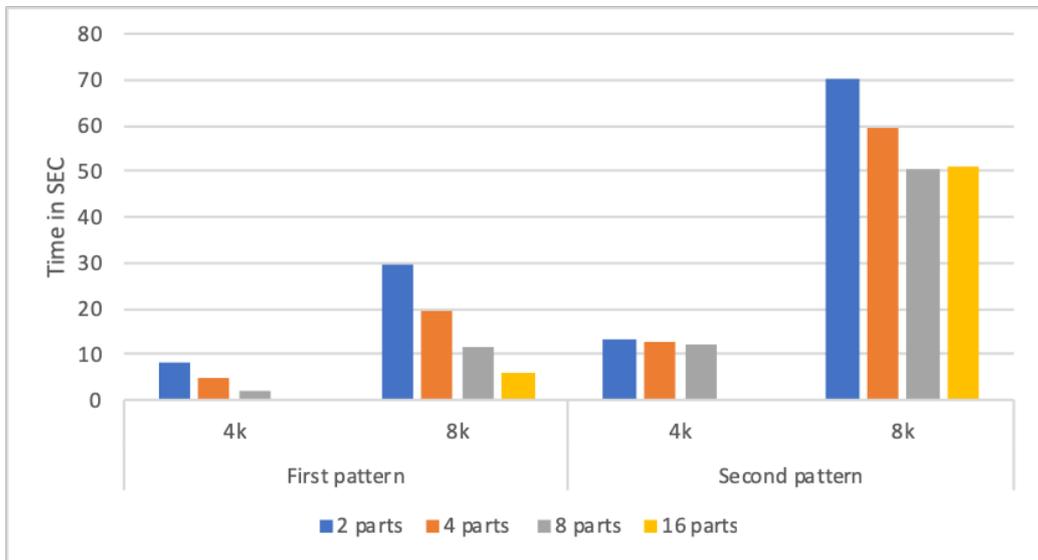


Figure 4.4: Performance of our method as the number of partitions increases.

Another main factor that affects the performance of our method is the number of partitions that the graph is divided into. We conducted experiments to evaluate the impact of this factor. We generated two graphs (with 4k and 8k nodes and process them with an increasing number of partitions under two communication patterns. Figure 4.4 shows the experimental results. With pattern 1, the execution time of our method decreases as the number of partitions increases for the graphs with both 4k and 8k nodes. Moreover, it can be seen that the execution time with pattern 2 is much higher than that with pattern 1 for both graphs. These results verify the

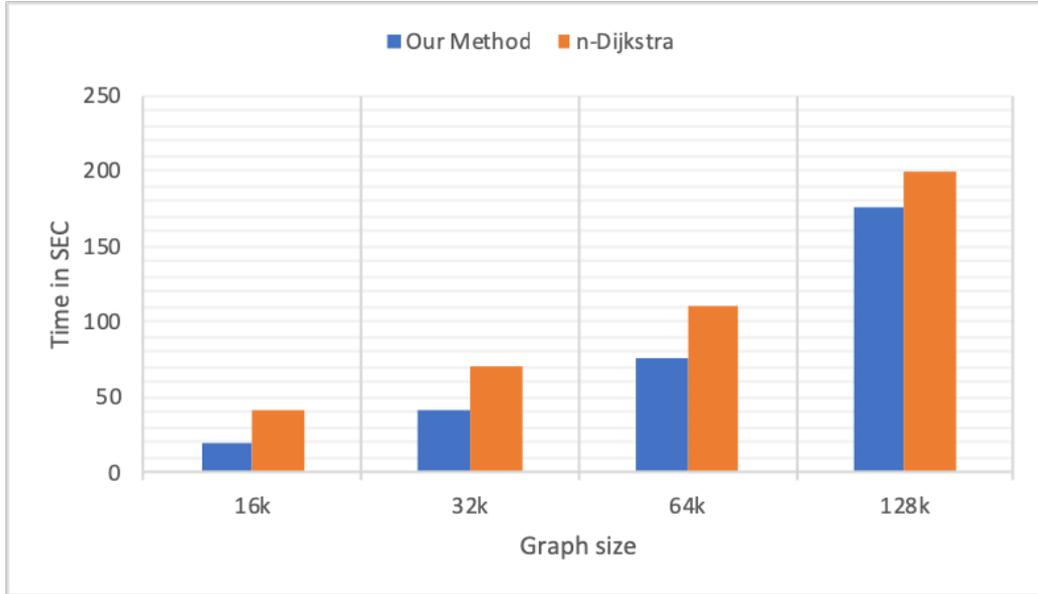


Figure 4.5: Performance of our method as the graph size increases.

results obtained in Figure 4.3. We also conducted the experiments with the graph sizes increasing to 128k. The similar trends can be observed with those larger-scale graphs.

Another observation is that with pattern 2, the execution time remains almost unchanged as the number of partitions increases for the graph with 4k nodes. This is because the execution time is dominated by the communication cost when the graph is of a relatively small scale (4k). When the number of graph nodes is 8k, the execution time decreases when the number of partitions increases from 2 to 8. However, when the number of partitions further increases to 16, the execution time increases slightly. This is also due to the high communication cost in pattern 1. Increasing the number of partitions further will greatly increase the communication time, which cancels

the decrease in computation time in each partition.

4.4.3 Performance on Big Graphs Evaluation

Graph size is obviously one of the most important factors that determines the execution time of a graph processing algorithm. Figure 4.5 shows the performance of our method with the graphs of different sizes. In these experiments, the graph size increases up to 128k and the graphs are processed by up to 64 processors. Only pattern 1 is presented since it achieves better performance than pattern 2 as shown in Figures 4.3 and 4.4. With a particular graph size, we ran the experiments on the number of processors increasing to 64 and chose the number of processors that delivered the best performance, which is the performance plotted in the Figure. We also used the n-Dijkstra algorithm to process those graphs and show their performance as the baseline in Figure 4.5.

As we can see from Figure 4.5, the execution time achieved by our method increases as the graph size increases, but is still less than that delivered by the n-Dijkstra algorithm in all graph sizes. Remember that we discussed at the beginning of this section that there is no communication cost in n-Dijkstra. Even so, our method still outperforms n-Dijkstra in all cases presented in the figure. These results indicate once again that our algorithm designed for distributed memory architecture is efficient.

4.5 Summary

We presented a new method to solve the APSP problem for the distributed memory architecture, aiming to address the limitation of the parallel APSP method developed for the shared memory architecture. In our method, the entire graph is partitioned into sub-graphs, each being allocated to a processor. The processors process their local sub-graphs in parallel and find the shortest paths between any node in the subgraphs. Then the local results are combined through two proposed communication patterns to obtain all-pairs shortest paths for the entire graph. We used MPI to implement our method. The experiments show that our algorithm can find 16 million paths (in the graph with 4k nodes) and 64 million paths in about two seconds and five seconds, respectively.

CHAPTER 5

Accelerating The solving of the All Pair Shortest Path

Algorithm with GPU

5.1 Introduction

Graphics processing units (GPUs) have been developed for computation-intensive applications and improved the performance significantly. Many researchers used GPU to improve the performance by taking advantage of its much more processing cores than multi-core computers [15]. Many applications have been accelerated through GPU programming, such as deep learning and neural networks, image processing, scientific computing, medical imaging, cryptography, and so on [12]. One of the areas in which GPU is used is solving the problem of finding shortest paths [88].

In this chapter, we present a hybrid method that uses both CPU and GPU to solve the APSP problem for a weighted graph. This method parallelizes the first three steps of our CNA (common node algorithm, presented in Chapter 3) on the CPU and the last two steps, which combine the local results, on the GPU. The steps run in the CPU used OpenMP for parallelization in the same way as we did in chapter 3. For running the steps on GPU, Cuda is used for programming the steps. Two approaches are designed to combine the local results. We call these approaches Hybrid Thread and Hybrid Block

(H-thread and H-block for short). Each approach employs its own way of scheduling the operations on the GPU. Moreover, both H-thread and H-block ensure achieving excellent performance by considering two factors: i) minimizing the communication between the host (CPU) and the device (GPU), and ii) launching the appropriate kernel to take advantage of the GPU power. In addition, we also extend our GPU version of parallel CNA to work in a multi-GPU system in order to further accelerate the processing.

We conducted the experiments with different graph sizes and compared our hybrid CNA (HybridCNA) with our shared memory-based CNA (SM-CNA), n-Dijkstra algorithm [26] and ParAPSP algorithm [57]. Besides, we test HybridCNA with different real-world graphs such as social networks and road networks. The experimental results show that HybridCNA can achieve far better performance than other methods.

5.1.1 Motivation

No. of Common node	Combining time (sec)	Total time(sec)
10	0.296	7.008
100	2.196	8.88
500	11.109	17.809
1000	24.284	30.967

Table 5.1: A case study of finding APSP in a graph with the size of 4k size; the graph is divided into two partitions.

Solving the APSP problem is computation expensive. With the methods proposed in previous chapters, these computations can be performed in parallel. We have shown that our parallelization methods can achieve ex-

cellent performance. However, one of the factors that affect the number of operations in our method is the number of common nodes between the parts (subgraphs). When there is one more common node between two partitions, thousands of more computations may be needed depending on the number of nodes in the subgraph. As shown in table 5.1, the time increases dramatically when the number of common nodes increases. Therefore, if we can further parallelize these computations in the stages of combining local results, we can further improve the performance. The steps of combining local results, i.e., step 4 and 5 in our CNA 3.2.2, find the shortest paths between the nodes in different parts. The shortest path is the one with the minimum cost among all paths that connect the source and destination nodes and cross one of the common nodes between the two subgraphs. So the computations are independent and can be parallelized. Taking figure 5.1 as an example, finding the shortest path between node 0 in the first partition and node 7 in the second partition can be computed in parallel with finding the shortest path between node 1 and node 7. Unlike the CPU, which is limited by the number of CPU cores, GPUs have thousands of processing cores and therefore has the ability to run thousands of operations in parallel. Since we may have to perform millions of operations (depending on the graph size) for combining the local results to find the shortest between any nodes in different subgraphs, GPU provides an excellent opportunity to further accelerate the combining process.

The remainder of this chapter is organized as following: Section 5.2 presents the hybridCNA for solving the APSP problem, which includes the steps running on CPU, the steps on GPU and the communication behaviour

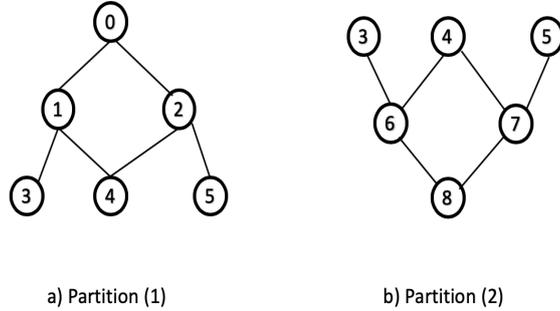


Figure 5.1: Example of a graph being partitioned into two subgraphs

between them. In section 5.3, we explain the two approaches of finding the shortest paths between the nodes in different parts (combining steps) on the GPU. Section 5.4 extends the HybridCNA to the multi-GPU environment. Experimental results are presented and analyzed in Section 5.5. Finally, the chapter is summarized in section 5.6.

5.2 HybridCNA

HybridCNA further accelerates SM-CNA (shared memory-based CNA) presented in chapter 3. The acceleration works by processing the steps of combining local results on GPU. In this section, we present the details of HybridCNA. In particular, we describe the steps executed on the CPU and those run on the GPU as well as the mechanism through which the CPU and the GPU transfer the data between each other.

5.2.1 Processing on CPU

The steps run on the CPU (host) start with reading the graph and partition it into a certain number of subgraphs. Then the 2D distance matrix

are created, which is updated with the distances of the shortest paths during the processing. OpenMP is used to generate threads and each thread runs on a CPU core to process a subgraph. All CPU threads are running simultaneously to find the shortest path between the nodes in the same subgraph. Each thread updates the distance matrix with the distances of discovered shortest paths. CPU allocates the device memory on GPU. The size of the allocated memory equals to the size of the 2D distance matrix in the CPU memory. After this, the processing on CPU is completed and the execution is shifted to GPU by launching the GPU kernel, which is used to aggregate the local results between the subgraphs and find the shortest paths between the nodes in different subgraphs. We refer to this process the combining processes. When the combining process is completed on GPU, GPU transfers the updated distance matrix, which now holds the distances of the shortest paths between any two nodes in the entire graph, to the host.

5.2.2 Processing on GPU

The threads on the GPU run simultaneously and apply the following equation to combine the local results in each subgraph and find the shortest paths between the nodes in different sub-graphs.

$$W(v_i, v_j) = \text{Min}\{W(v_i, v_{C_m}) + W(v_{C_m}, v_j) | 1 \leq m \leq r\} \quad (5.1)$$

The threads find the values $W(v_i, v_{C_m})$ and $W(v_{C_m}, v_j)$ in the distance matrix, which is received from the CPU. Then the threads update the matrix by adding the found values ($W(v_i, v_j)$). Finally, after finding all shortest path

between nodes in all subgraphs, the GPU sends the distance matrix back to the CPU. We design two approaches, called H-Thread and H-Block, to manage the execution of the GPU threads for combining local results. The details of these two approaches work are presented in section 5.3.

5.2.3 Data Transferring

The communication between the CPU and the GPU needs to be minimized in order to achieve good overall performance. The threads on GPU perform the steps of combining the local results. As presented before, in these steps, the threads find the shortest path between two nodes in different subgraphs among all paths that connect the source and destination nodes and pass through one of the common nodes. Assume s is source node in the first subgraph, t is the destination node in the second subgraph and c is the common nodes connecting two subgraphs. Then the information required to find the shortest path between s and t is the shortest path between s and c , and between c and t . This information is stored in the distance matrix hold in the CPU memory. This distance matrix needs to be transferred from CPU memory to GPU memory. Unlike many GPU works for solving the shortest path problem, we do not need to copy the graph itself to the GPU, but only copy this distance matrix once, which reduce the communication cost and also allows us to solve the APSP for even bigger graphs that cannot be fit in the GPU memory. When the graph is so big that the corresponding 2D matrix is bigger than the GPU memory, we can use the 3D matrix presented in 3.4.4 as the data structure or partition the matrix along the rows and send

Table 5.2: Case study of the operations needed to combine the result of five subgraphs

$$\begin{array}{c}
 0 \text{ to } 1 \\
 0 \text{ to } 2 \\
 0 \text{ to } 3 \\
 0 \text{ to } 4
 \end{array}
 \left| \begin{array}{c}
 1 \text{ to } 2 \\
 1 \text{ to } 3 \\
 1 \text{ to } 4
 \end{array} \right|
 \left| \begin{array}{c}
 2 \text{ to } 3 \\
 2 \text{ to } 4
 \end{array} \right|
 \left| \begin{array}{c}
 3 \text{ to } 4
 \end{array} \right.$$

a subset of rows in the matrix to the GPU at a time.

5.3 H-Thread and H-Block approaches

The key factor of achieving excellent performance on the GPU is by managing how the threads act to fulfil the work. In this section we propose two approaches to manage the GPU threads.

5.3.1 Parallelizing the Combining Steps of HybridCNA

We refer to the combining step which aggregates the local results from the subgraphs as an operation. Namely, every two subgraphs need to consume one operation to find the shortest path between their nodes. Toward executing this combining step, we need to do a certain number of operations. The total number of these operations can be calculated by the following equation:

$$\sum_{i=1}^{P-1} P - i \quad , \quad P \text{ is the total number of partitions} \quad (5.2)$$

Now we get into the details of these operations and the parallelism possibility.

Suppose that we have a graph partitioned into five parts ranked as 0,

1, 2, 3 and 4. The operations needed to combine the results are illustrated in table 5.2, where “0 to 1” represents finding the shortest paths between the nodes in partitions 0 and 1. There are two types of parallelism we can exploit:

1. Operations parallelism :

Operations parallelism runs more than one operation at the same time. Since some operations are independent, such as (0 to 1) and (1 to 2) (which are the operations in different columns in table 5.2), they can be run in parallel. However, the operations in the same column in the table have dependency and have to be run in sequence. For example, the operation (0 to 2) can only start after (0 to 1) has been completed.

2. Nodes parallelism :

Nodes parallelism finds the shortest paths between the nodes in two different subgraphs (one operation) in parallel. For example, if s and t are the nodes in partition 0, finding the shortest paths from s to all nodes in the partition 1 can be run in parallel with finding the shortest paths from t to all nodes in the partition 1.

Based on these two types of parallelism, we designed two approaches (H-Thread and H-Block) to achieve the parallelism on the GPU, aiming to exploit the full potential of GPU.

5.3.2 H-Thread

In this approach, all the GPU threads operate to find the shortest path between two partitions at a time. The node parallelism is implemented,

which finds the shortest path of the nodes in one partition to the nodes on another partition in parallel. The operations are run in sequence.

The H-Thread works by assigning a certain number of nodes to a thread, ranging from 1 to n_i , where n_i is the total number of nodes in the subgraph. When all threads complete the processing, they repeat the processing for next operation until all operations have been completed. Consider the example in table 5.2. The nodes in subgraph 0 are assigned to a number of threads. Each thread finds the shortest path between the nodes assigned to it and all nodes in subgraph 1. After the first operation is completed, the threads move to run the second operation (0 to 2). The procedure goes on until the last operation, (3 to 4), has been completed. Algorithm 7 shows the process of the H-thread approach.

Algorithm 7 Pseudo-code of H-Thread

Input: $[M_r]$ the distance matrix; $[P]$ the total number of partitions; and $[C_{i,j}]$ the list of common nodes between subgraphs i and j ;

Output: APSP matrix $[M_r]$

```

1 index = blockIdx.x * blockDim.x + threadIdx.x;
  x= index;
  for i in range(0 to P - 1) do
2   x= x+ rank of first node in i;
   for j in range(i + 1 to P - 1) do
3     for y in j nodes do
4       for c in  $C_{i,j}$  do
5         if  $M_r[x][y] > M_r[x][c] + M_r[c][y]$  then
6            $M_r[x][y] = M_r[x][c] + M_r[c][y]$ ;

```

The number of nodes assigned to a thread equal to $(\frac{n_i}{T_i} | T_i \leq n_i)$, where

5. Accelerating The solving of the All Pair Shortest Path Algorithm with GPU

n_i is the total number of nodes in a subgraph and T_i is the total number of threads launched to run the kernel. The maximum number of threads that can be launched equals to the total number of nodes in the biggest subgraph. The time that a thread takes to complete one operation is modelled as follows, where n_i is the total number of nodes in subgraph i and $c_{i,i+1}$ is the total number of common nodes that connect the two neighbouring subgraphs i and $i + 1$.

$$\frac{n_i}{T_i} \times c_{i,i+1} \times (n_{i+1} - c_{i,i+1}) \quad (5.3)$$

Thus the total time a thread takes to cover all the operations (denoted by t_{time}) can be modelled by the following equation: where P_i is the total number of graph partitions (subgraphs)

$$t_{time} = \sum_{i=0}^{P_i-1} \frac{n_i}{T_i} \times \sum_{j=i+1}^{P_i-1} c_{i,j} \times (n_j - c_{i,j}) \quad (5.4)$$

Using equation 5.4, we can apply the following equation to calculate the total time that the GPU kernel takes to finish the combing step with the H-Thread approach, Where CO is the communication time between CPU and GPU

$$GPU_{time} = \max\{t_{time}\} + CO \quad (5.5)$$

5.3.3 H-Block

In the H-Block approach, the GPU threads are divided into a number of blocks. Each block has a number of threads. We schedule the operations between the blocks. The block of threads process the operations simultaneously to find the shortest paths between the subgraphs. Both operation parallelism and node parallelism are implemented in this approach. For instance, if we schedule the operations in table 5.2 to run by four blocks, the operations 0 to 1, 1 to 2, 2 to 3 and 3 to 4 will be run by four blocks in parallel, which is the operation parallelism. Moreover, the threads in each block finds the shortest paths between the nodes in one operation in parallel, which is the node parallelism. Algorithm 8 outlines the H-Block approach.

The maximum number of blocks that we generate is $P_i - 1$, where P_i is the total number of subgraphs. In each block, the number of threads equals to $\lfloor \frac{T_i}{B_i} \rfloor$, $B_i \leq T_i$, where T_i is the total number of threads launched and B_i is the total number of blocks generated. The time that a thread needs to complete one operation is modelled as follows, where T_b is the total number of threads assigned to a block.

$$\frac{n_i}{T_b} \times c_{i,i+1} \times (n_{i+1} - c_{i,i+1}) \quad (5.6)$$

The total time that a thread in one block needs to complete all the operations, denoted by (tb_{time}), can be calculated by the following expression, where S_i is the total number of subgraphs assigned to the block.

Algorithm 8 Pseudo-code of H-Block

Input: $[M_r]$ the distance matrix; $[P]$ the total number of partitions; and
 $[C_{i,j}]$ the list of common nodes between subgraphs i and j ;

Output: APSP matrix $[M_r]$

```

1 index = threadIdx.x;
  i = blockIdx.x;
  x= index+ rank of first node in  $P_i$ ;
  for  $j$  in range( $i + 1$  to  $P - 1$ ) do
2   for  $y$  in  $j$  nodes do
3     for  $c$  in  $C_{i,j}$  do
4       if  $M_r[x][y] > M_r[x][c] + M_r[c][y]$  then
5          $M_r[x][y] = M_r[x][c] + M_r[c][y]$ ;

```

$$tb_{time} = \sum_{i=0}^{S_i-1} \frac{n_i}{T_i} \times \sum_{j=i+1}^{P_i-1} c_{i,j} \times (n_j - c_{i,j}) \quad (5.7)$$

B_{time} is the time that a block needs to find the shortest paths between all the subgraphs assigned to it. We can calculate it by taking the maximum of equation 5.7 (i.e., $\max\{tb_{time}\}$). Lastly, we can apply the following expression to determine the total time that a GPU requires to accomplish the combining task with the H-Block approach, where CO is the communication time between CPU and GPU

$$GPU_{time} = \max\{B_{time}\} + CO \quad (5.8)$$

Which approach should be used depends on the type of graph we are processing. We compare the H-Thread and H-Block approaches in detail in the evaluation section 5.5.

5.4 Solving APSP on Multi-GPU systems

GPU 1	GPU 2	GPU 3
0 to 1	1 to 2	2 to 3
0 to 2	1 to 3	2 to 4
0 to 3	1 to 4	3 to 4
0 to 4		

Figure 5.2: Scheduling the operations of a graph partitioned into five sub-graph to run on three GPUs

In last section, we presented our HybridCNA method based on a single CPU (plus a single GPU). In this section, we extend the method to be run on the computer which is equipped with multiple GPUs. In this setting, the CPU schedules the operations to run across multiple GPUs. The CPU sends the required data to each GPU and launch the kernels, and all GPUs process the operations in parallel. The result matrix is copied to all GPUs. Moreover, this matrix will hold the distances of the shortest paths between any nodes in the entire graph after all GPUs complete their work. When launching the kernels, we choose the H-Thread or H-Block approach to manage the threads in each GPU. Which approach should be used depends on the operations assigned. Take the operations in figure 5.2 as an example. In GPU1 and GPU2, we can use only the H-Thread approach since the operations depend on each other. On GPU3, however, we can use the H-Block approach (operations in a different colour are independent with each other).

The time that each GPU needed to complete the work can be calculated by equation 5.5 or equation 5.8 depending on which approach is used. The total time of combining the local results on a multi-GPU system equals to

$\max\{GPU_{time}\} + CO$, where CO is the total time of communication.

5.5 Experimental Evaluation

In this section, we evaluate the efficiency of the HybridCNA method presented in this chapter. First, we evaluate the performance of these two approaches, H-Thread and H-Block, with various graph sizes; then we compare the HybridCNA method with the SM-CNA method presented in chapter 3. Finally, we evaluate the effectiveness of the HybridCNA method with a real-world graph.

The experiments were run on the Joint Academic Data Science Endeavour (JADE) server at the Oxford University [76]. It contains eight NVIDIA Tesla V100 GPUs interconnected by NVIDIA's NV link interconnect technology. The specification of CPU is Dual 20-Core Intel Xeon E5-2698 v4 2.2 GHz. The main Memory is 512 GB 2,133 MHz DDR4 RDIMM. Cuda 10.1 with OpenMP 3.0.0 are installed to implement the HybridCNA.

We conduct the experiments with different graph sizes from 2k to 16k. The graph generator [49] is used to generate the graphs following power-law distribution, which has been shown to be the property of real-world graphs. To evaluate the impact of the common nodes, we first ran the experiment with a different number of common nodes from 2 to 1000. After that, we set the total number of common nodes to be 200 for the rest of the experiments, which is the highest number of common nodes that the real-world graph used in our experiments has.

5.5.1 Evaluate H-Thread and H-Block

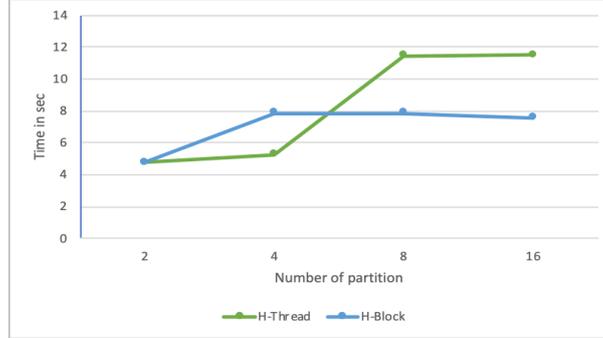


Figure 5.3: Comparing the H-Thread approach with the H-Block approach. The graph size is 16k

In this section, we compare the two approaches, H-Thread and H-Block, proposed to manage the GPU threads. We ran the experiment on a graph with the size of 16k and partitioned the graph into 2, 4, 8 and 16 subgraphs. The experimental results are shown in figure 5.3, where the y-axis is the time taken to aggregate the results between the subgraphs (kernel time) with each approach. We neglected the communication time between the CPU and the GPU (copying the data to the GPU memory) because it is the same in both approaches.

As shown in figure 5.3, when the number of the partitions is small (two and four), the H-Thread approach achieves better performance than H-Block. However, as the number of partitions increases H-Block overtakes H-Thread. This is because when a graph is partitioned into a small number of subgraphs, the number of nodes in each subgraph is big. Since in H-Thread the nodes in an operation are parallelized and the operations are handled one at a time, there will be more threads engaged in the computation simultaneously than when the graph is partitioned into a smaller number of subgraphs. Addition-

5. Accelerating The solving of the All Pair Shortest Path Algorithm with GPU

Number of Common node	<i>CPU</i>		GPU	
	Combining Time	Total time	GPU Time	Total time
10	0.296	7.008	0.477	7.136
100	2.196	8.88	0.629	7.346
500	11.109	17.809	1.259	7.92
1000	24.284	30.967	1.814	8.475

Table 5.3: Case study of finding APSP of a graph with the size of 4k and partitioned into two subgraphs.

ally, in H-Block, at least one block completes its task and terminates after each iteration, which means a less number of threads work simultaneously after each iteration (i.e, the decrease in the degree of parallelism). The effect is noticeable if the number of threads in each block is large (such as the four partitions in the figure 5.3). However, when the number of partitions increase to 8 and 16, a fewer number of GPU threads are terminated and more threads work simultaneously for longer time (i.e., the increase in the degree of parallelism). This is when H-Block performs better than H-Thread.

Also, we can notice that unlike H-Thread, the number of partitions does not affect much the performance of the H-Block approach. Since the operations are run in parallel in H-Block, all nodes in the subgraphs are assigned to the GPU threads and run at the same time.

To summarize, we can conclude that both H-Thread and H-Block can achieve excellent performance in different conditions. When we have a large subgraph, it is better to use the H-Thread approach. When we have many small subgraphs, using H-Block is more beneficial.

5.5.2 Comparing HybridCNA with SM-CNA

In this subsection, we compare our HybridCNA with the shared memory-oriented CNA (SM-CNA) presented in chapter 3. Three parameters are investigated in the experiments: the number of common nodes, the graph size, and the number of partitions.

As we discussed before, the number of common nodes plays an important role in the performance of our method since it affects the number of operations required to combine the local results. To investigate the effect of this parameter, we ran the experiment on a graph with the size of 4k and partitioned into two subgraphs. We increased the number of common nodes from 10 to 1000 and recorded the time spent in the combining step as well as the total time of the algorithm. The results are shown in table 5.3. It is apparent that for both two methods, the more common nodes, the more time spent in combining the local results. However, the increasing rate of the time is much higher in SM-CNA than in HybridCNA. SM-CNA took 0.2 sec to combine the results when there are 10 common nodes and took 24.2 sec when the number of common nodes increased to 1000, which is about 82 times slower. HybridCNA took 0.4 sec when the number of common nodes is 10 and 1.8 sec when it increased to 1000, which is only about 3.8 times slower. The reason for this is because of the much higher power of the GPU, which supports the simultaneous running of much more threads. On the contrary, the number of threads that can be generated in SM-CNA is limited by the number of CPU cores.

Another observation from the table is that when the number of common

5. Accelerating The solving of the All Pair Shortest Path Algorithm with GPU

nodes is small (such as 10), SM-CNA is faster. This is because the communication overhead between the host and the device in HybridCNA is higher, which will be discussed in next subsection.

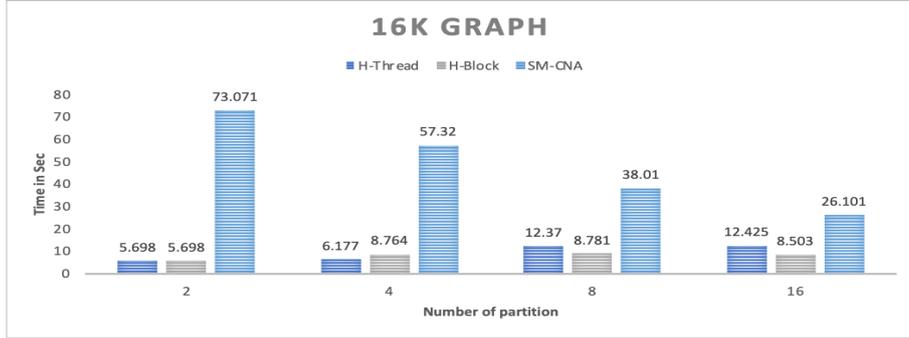


Figure 5.4: Comparing HybridCNA with SM-CNA. The graph size is 16k

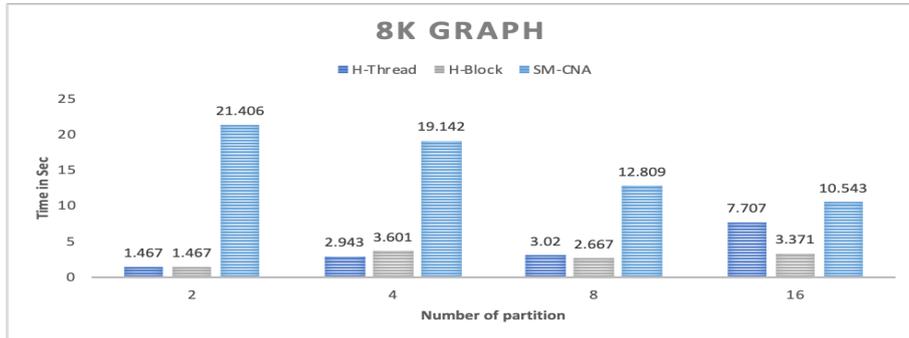


Figure 5.5: Comparing HybridCNA with SM-CNA. Graph size is 8k

In next experiment, we ran HybridCNA and SM-CNA with different graph sizes and each graph being partitioned into up to 16 subgraphs. We recorded the time of the combining step spent by each method. The combining time of Hybrid-CNA consists of the kernel time plus the time of copying the data between the CPU memory and the GPU. We set the number of common nodes to be 200 in this evaluation.

As figures 5.4, 5.5, 5.6 and 5.7 show, with all graph sizes the combining

5. Accelerating The solving of the All Pair Shortest Path Algorithm with GPU

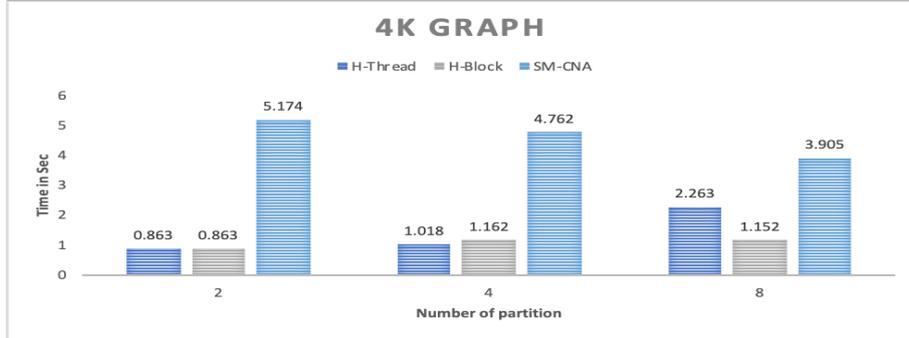


Figure 5.6: Comparing HybridCNA with SM-CNA. Graph size is 4k

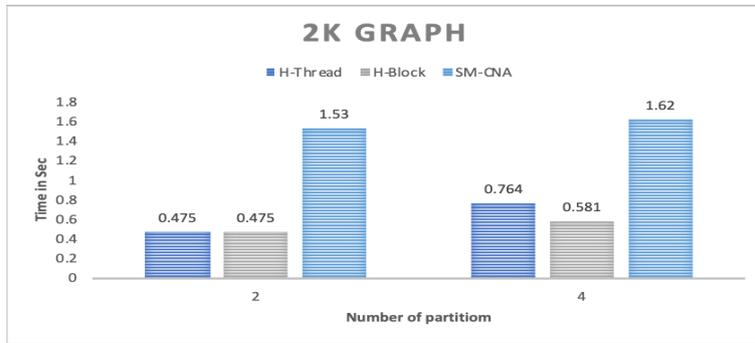


Figure 5.7: Comparing HybridCNA with SM-CNA. Graph size is 2k

time in HybridCNA is less than that in SM-CNA. Furthermore, as the graph size increases, the performance gap between the two methods increases. For example, as shown in 5.7, for the graph with the size of 2k and partitioned into two subgraphs, HybridCNA is about 3.2x faster than CPU SM-CNA. For the graph with the size of 16k, HybridCNA is 12.8x faster, as shown in Figure 5.4. This is because when the graph is bigger, more operations are performed. The GPU has the ability to run these operations in parallel.

When investigating the number of partitions, we notice that when the number of partitions increases, the time of SM-CNA decreases, and the time of HybridCNA increases slightly (or remains the same). That is because the more subgraphs we partition the graph into, the more CPU cores are needed

by SM-CNA to find the shortest paths between the subgraphs, where in HybridCNA the number of partitions does not affect the number of threads used. However, even with this increased time of SM-CNA, HybridCNA still achieves better performance by at least 3x.

In figure 5.8, we show the total time that HybridCNA and SM-CNA achieve when running the graphs with different sizes. In all cases, HybridCNA outperforms SM-CNA. This result is expected since the combining time influences the total time of the methods.

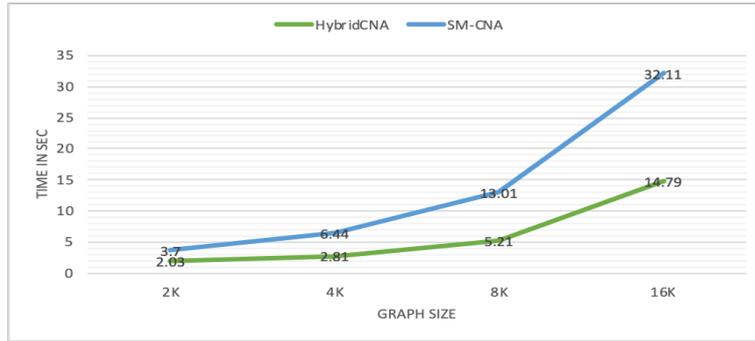


Figure 5.8: Total time of HybridCNA and SM-CNA methods in different graph size

5.5.3 Evaluation on Multi-GPU systems

In this section, we evaluate HybridCNA in a computer equipped with multiple GPUs. The experiments were run on up to four GPUs, and examined HybridCNA with three different graph sizes: 4k, 8k and 16k.

We need to copy the distance matrix from CPU to the GPU. Since the JAD architecture [76] has a memory layer where the multiple GPUs share, it gives us the option whether to copy the matrix once to the GPU memory shared by multiple GPUs, or copy it multiple times to the local memory

inside each GPU. Table 5.4 shows the difference between these two copying strategies. As you can see, copying the data once reduces the communication cost between the CPU and the GPU and needs less time than copying the data multiple times. Additionally, when using the multi-copying strategy, the more GPUs, the more time needed. However, we noticed that the kernel time increased significantly when copying the matrix only once, and that the more GPUs are engaged in the computation, the more time needed to finish the work. The reason behind this is because of the racing condition [44]. There are thousands of GPU threads trying to read and write to the same memory address in GPU, which cause the dramatic increase in kernel time. Therefore, copying the data multiple times achieved more better performance than copying once in terms of the total kernel time. Therefore, we will use the multi-copying strategy in the following experiments.

Copying way	1GPU			2GPUs			3GPUs		
	copy time	kernel time	Total GPU time	copy time	kernel time	Total GPU time	copy time	kernel time	Total GPU time
one copy	536	2131	2667	536	4887	5423	536	10644	11180
Multi copying	536	2131	2667	631	1639	2270	740	1386	2126

Table 5.4: A case study of comparing multi-copying with single copying strategy. The graph size is 8K and the graph is partitioned into 8 subgraphs (Time in milliseconds)

Figure 5.9 and 5.10 show the total GPU time (data copying time plus the kernel time) of running HybridCNA on the graphs with the sizes of 8K and 16K. We can observe from these two figures that HybridCNA achieves better performance as the number of GPU increases. Running with two GPUs achieved the excellent speedup (up to 1.6). Similarly, running with 4 GPUs achieved up to 2.5 speed up. This performance is affected by the graph size and the number of partitions. For instance, as shown in 5.9, when

5. Accelerating The solving of the All Pair Shortest Path Algorithm with GPU

the graph is partitioned into 16 subgraphs, running with 4 GPUs took longer than with 3 GPUs when the graph size is 8K. When processing the graph with the size of 16K, running with 4 GPUs achieved better performance than with 3 GPUs, as shown in 5.10.

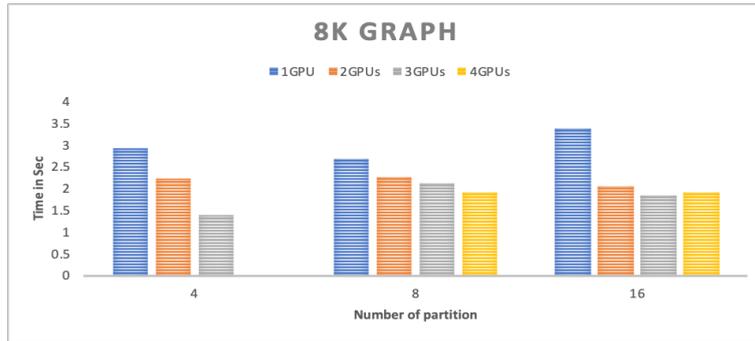


Figure 5.9: Comparing the kernel time of running with 1 GPU and running with Multiple GPUs. The graph size is 8k

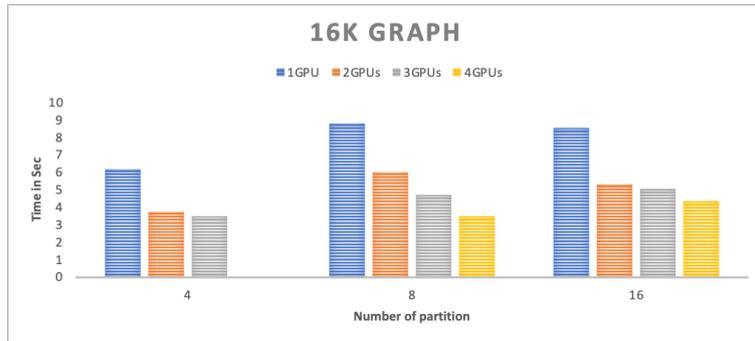


Figure 5.10: Comparing the kernel time of running with 1 GPU and running with Multi-GPUs. The graph size is 16k

Figure 5.11 shows the total time when HybridCNA processes the graphs with various sizes on multiple GPUs. We can observe that for the graph of 4K, processing it with 2 GPUs is slightly better. Further increasing the number of GPUs did not improve the performance. On the contrary, for the 16K graph, processing it with 2 GPUs achieved the excellent speed up. As

we increased the number of GPUs further to 3 and 4, we obtained even better performance. Therefore, we can conclude that it is beneficial to use multiple GPUs when the graph size is big.

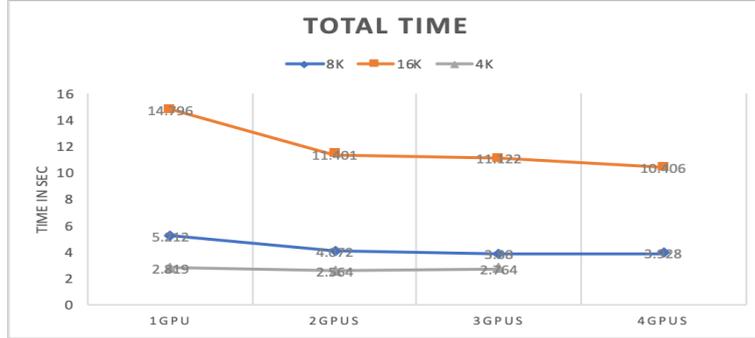


Figure 5.11: Comparing the total time between running with 1 GPU and running with Multiple GPUs on different graph sizes

5.5.4 Evaluating HybridCNA with Real-world Graphs

To prove the effectiveness of our SM-CNA and HybridCNA, we evaluate them with real-world graphs and compare them with the n-Dijkstra algorithm [26] and the ParAPSP algorithm [57]. We use diverse types of graphs, including social media network, road network and power networks. Table 5.5 lists the dataset of these networks, showing the size of the graph, the number of subgraphs we partition it into, the total number of the common nodes, and finally the time needed to pre-process the graphs. All the real graphs used in this thesis are obtained from SNAP [64], KONECT [62] and Network Repository [86].

Figure 5.12 shows the comparison between SM-CNA, HybridCNA, n-Dijkstra and ParAPSP. Observing the performance of SM-CNA and HybridCNA, we can see that when the number of common nodes is small (such as

5. Accelerating The solving of the All Pair Shortest Path Algorithm with GPU

Network's Name	Size	Number of partition	Total number of common nodes	preparing (time in milliseconds)
road-minnesota	2.6K	2	40	103
Western-US.Power	4.9K	3	183	360
Tweeter_israel.	3.6K	2	200	180
out.ego-facebook	2.8K	2	5	141
SW-Trial	10k	7	43	660

Table 5.5: The dataset for the real networks we run the experiments on

the out.ego-facebook network), SM-CNA achieves better performance. However, if the number of common nodes is big (such as Western-US.Power and Tweete-risrael), HybridCNA delivers much better performance. We can draw the same conclusion when comparing SM-CNA with ParAPSP. Furthermore, from the performance of n-Dijkstra and SM-CNA in figure 5.12, we can see that SM-CNA is faster than the n-Dijkstra algorithm in all cases. However, the gap between them decreases when the number of common nodes increases. This outcome matches the results of the experiments conducted on the simulated graph. It motivated us to develop HybridCNA, which achieves higher performance than all other compared algorithms.

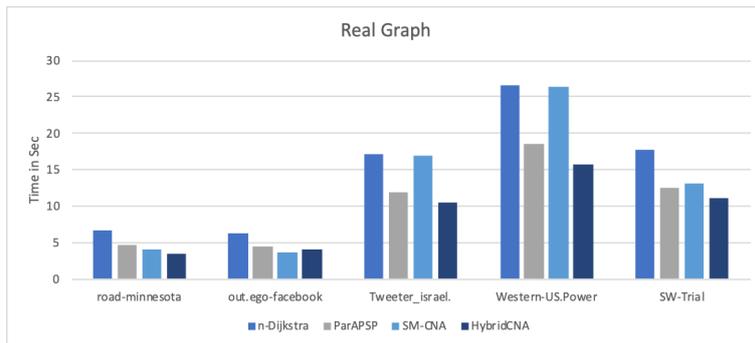


Figure 5.12: Testing CPU and Hybrid methods on a real-world networks

5.6 Summary

In this chapter, we proposed a new method, HybridCNA, for solving the APSP problem on the GPU. It is a hybrid approach using both CPU and GPU. In HybridCNA, we developed two approaches, called H-Thread and H-Block, to manage the GPU threads. Both approaches achieved excellent performance on particular types of graphs. Furthermore, we extend HybridCNA to the multi-GPU systems, which achieved up to 2.5x speedup compared to running with a single GPU. To evaluate the proposed method, the experiments were conducted with simulated and real-world graphs. The results show that the performance of HybridCNA is better than all other compared methods.

CHAPTER 6

Conclusions and Future Work

The work presented in this thesis aims to parallelize the process of finding the All-pairs Shortest Path (APSP) in graphs. We developed efficient methods to find the APSP on shared memory architecture, including multi-core computer and GPU, and distributed memory architecture. Furthermore, we extend our method to a hybrid architecture with both CPU and GPU. The hybrid method, called HybridCNA further improves the performance significantly. Key contributions of this thesis are summarized in the first three sections of this chapter. Further work is discussed in Section 6.4.

6.1 Developing the Parallelization Method for Finding the All-Pairs Shortest Paths in the Shared Memory Architecture

Finding the All-Pairs Shortest-Path in graphs is a fundamental problem, which has attracted the attention of many researchers for decades. There are many applications of this problem, such as road networks, transportation, and robotics. The massive computations required to solve the APSP problem and the ever-increasing network size demand the efficient parallel solutions

to the problem. In chapter 3, we addressed this problem and presented a new parallel method called SM-CNA, to find the APSP on shared memory architecture. In this method, the graph is partitioned into subgraphs. We find the APSP in each subgraph in parallel, and then combine the local results in subgraphs to find the APSP in the entire graph. The Breadth-First Search (BFS) algorithm is used to transform the graph into a levelled topology, and the graph is partitioned along the nodes in particular levels. The common nodes between the subgraphs are duplicated. By using the information of the duplicated nodes, we reduced the overall computations of the APSP problem significantly, and achieved outstanding performance. Moreover, the correctness of our method is mathematically proved in the chapter. Furthermore, we managed to process bigger graphs by adopting a 3D matrix as a data structure to store the solutions to the APSP problem. The effectiveness of our method is verified by the experiments which show improvement up to 6x faster than some compared methods.

6.2 Developing the Parallelization Methods for Finding the All-Pairs Shortest Paths in the Distributed Memory Architecture

Shared memory architecture such as a multicore machine is a mainstream parallel processing architecture. The main benefit of the shared memory architecture is its fast communication between the processing cores. However, the size of the graphs that can be processed by a shared memory machine is

limited by the memory size and the number of cores in the machine. In order to alleviate the resource limitation in a single shared memory computer, in chapter 4, we develop a parallelization method (called DM-CNA) for the distributed memory architecture, such as a cluster, to solve the APSP problem. By adding more machines, our method can process even larger scale graphs and achieve a much higher degree of parallelism. In the method, the graph is partitioned into subgraphs, which are assigned to processors. Each processor find the APSP in the assigned subgraph in parallel. We used the Message Passing Interface (MPI) library to implement two communication patterns for propagating local results obtained by one processor to all other processors engaged in the processing. Both patterns efficiently reduced the communication cost and improved the overall performance of the proposed method. Furthermore, we conducted the experiments on a high-performance cluster and evaluated our method with different graph sizes. The experiments show that our algorithm can find 16 million paths in about two seconds and five second to find 64 million paths.

6.3 Accelerating the solving of the APSP problem with GPU

Finding the APSP is computation intensive. In chapter 5, we develop the method called HybridCNA to solve the APSP problem on the GPU. This method is extended from our shared memory method. We discussed the limitation of SM-CNA in combining local results. HybridCNA is a hybrid

method, in which the CPU process the first three steps in CNA and schedule the remaining steps to run on the GPU. Furthermore, we implemented two threads managing approaches, called H-Thread and H-Block, to control the parallelization of GPU threads. The two approaches improved the performance by reducing the data transferring between the CPU and GPU memories and launching a suitable set of threads to combine local results efficiently. Moreover, we extend the method to the computer with multiple GPUs and achieved the further speedup. We conducted the experiments on the Joint Academic Data Science Endeavour (JADE) server, which is equipped with multiple GPUs. The results show outstanding performance with the real-world simulated networks which achieved up to 2.5x speedup compared to running with our CPU version.

6.4 Future Works

In chapter 3, we presented a way to partition the graph by using the multi-level algorithm BFS, which preprocesses the graph to reduce the computation and helps achieve excellent performance when the graph is processed in parallel. This motivates us to investigate more partitioning ways such as using Depth-First Search algorithms and nodes betweenness centrality concept. Another direction of improvement is that in our current method, graph pre-processing is run in sequence. In the future, we plan to parallelize the graph pre-processing, which can potentially further improve the performance of the whole scheme.

In chapter 4, the work is presented in the distributed memory architecture

and achieved the excellent performance. In the future, we plan to extend our method to a hybrid architecture, in which there are multiple computing nodes (distributed memory), each node being a multi-core machine reinforced by GPU (shared memory). Consequently, the hierarchy of parallelism will increase further, which should further speed up the process.

The work presented in this thesis solved the APSP problem on different computer architectures. In the future, we will extend the work to solve more graph problems such as the Single Source Shortest Path (SSSP) problem. In SSSP, the computation is much less than APSP. Since we achieved outstanding performance with the APSP, the methods on the SSSP should also achieve good performance. Furthermore, to increase the impact of this research, we plan to develop a graph processing tool. Depending on the graph type and size, the tool uses a deep-learning algorithm to decide the best way to partition the graph and process it in parallel. An example of an algorithm we can use is Graph Convolutional Networks (GCN). It employs the nodes neighbouring information method which should help us finding the right level of nodes to duplicate between processor. Moreover, the tool will be made available to other researchers to speed up the implementations of their graph processing problems.

Bibliography

- [1] K. Abdelghany, H. Hashemi, and A. Alnawaiseh. Parallel all-pairs shortest path algorithm: Network decomposition approach. *Transportation Research Record*, 2567(1):95–104, 2016.
- [2] L. A. Adamic and B. A. Huberman. Power-law distribution of the world wide web. *science*, 287(5461):2115–2115, 2000.
- [3] U. Agarwal and V. Ramachandran. New and simplified distributed algorithms for weighted all pairs shortest paths. *arXiv preprint arXiv:1810.08544*, 2018.
- [4] U. Agarwal and V. Ramachandran. Distributed weighted all pairs shortest paths through pipelining. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 23–32. IEEE, 2019.
- [5] U. Agarwal, V. Ramachandran, V. King, and M. Pontecorvi. A deterministic distributed algorithm for exact weighted all-pairs shortest paths in $\tilde{O}(n^{3/2})$ rounds. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pages 199–205, 2018.
- [6] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings*

- of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 349–360, 2013.
- [7] Y. Arfat, S. Suma, R. Mehmood, and A. Albeshri. Parallel shortest path big data graph computations of us road network using apache spark: survey, architecture, and evaluation. In *Smart Infrastructure and Applications*, pages 185–214. Springer, 2020.
- [8] J. Barnat, L. Brim, and J. Chaloupka. Parallel breadth-first search ltl model-checking. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, pages 106–115. IEEE, 2003.
- [9] C. Barrett, R. Jacob, and M. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.
- [10] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 46–59. SIAM, 2007.
- [11] A. Bernstein and D. Nanongkai. Distributed exact weighted all-pairs shortest paths in near-linear time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 334–342, 2019.
- [12] U. Bondhugula, A. Devulapalli, J. Dinan, J. Fernando, P. Wyckoff, E. Stahlberg, and P. Sadayappan. Hardware/software integration for fpga-based all-pairs shortest-paths. In *2006 14th Annual IEEE Sym-*

- posium on Field-Programmable Custom Computing Machines*, pages 152–164. IEEE, 2006.
- [13] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan. Parallel fpga-based all-pairs shortest-paths in a directed graph. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.
- [14] A. Brodnik and M. Grgurovič. Solving all-pairs shortest path by single-source computations: Theory and practice. *Discrete applied mathematics*, 231:119–130, 2017.
- [15] A. R. Brodtkorb, T. R. Hagen, and M. L. Sætra. Graphics processing unit (gpu) programming strategies and trends in gpu computing. *Journal of Parallel and Distributed Computing*, 73(1):4–13, 2013.
- [16] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.
- [17] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent advances in graph partitioning. In *Algorithm Engineering*, pages 117–158. Springer, 2016.
- [18] E. Cantú-Paz. A summary of research on parallel genetic algorithms. 1995.

- [19] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.
- [20] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: Theory and experimental evaluation. *Mathematical programming*, 73(2):129–174, 1996.
- [21] J. W. Choi and R. W. Vuduc. How much (execution) time and energy does my algorithm cost? *XRDS: Crossroads, The ACM Magazine for Students*, 19(3):49–51, 2013.
- [22] C. Cooper and A. Frieze. A general model of web graphs. *Random Structures & Algorithms*, 22(3):311–335, 2003.
- [23] T. H. Cormen. *Introduction to algorithms*. MIT press, 2009.
- [24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.
- [25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.
- [26] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.
- [27] K. Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.
- [28] N. Developer. Cuda toolkit documentation. <https://docs.nvidia.com/cuda/>. (Accessed on 06/19/2020).

- [29] E. W. Dijkstra et al. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.
- [30] H. Djidjev, G. Chapuis, R. Andonov, S. Thulasidasan, and D. Lavenier. All-pairs shortest path algorithms for planar graph for gpu-accelerated clusters. *Journal of Parallel and Distributed Computing*, 85:91–103, 2015.
- [31] H. Djidjev, G. Chapuis, R. Andonov, S. Thulasidasan, and D. Lavenier. All-pairs shortest path algorithms for planar graph for gpu-accelerated clusters. *Journal of Parallel and Distributed Computing*, 85:91–103, 2015.
- [32] J. Dümmler and S. Egerland. Interval-based performance modeling for the all-pairs-shortest-path problem on gpus. *The Journal of Supercomputing*, 71(11):4192–4214, 2015.
- [33] H. El-Rewini and M. Abd-El-Barr. *Advanced computer architecture and parallel processing*, volume 42. John Wiley & Sons, 2005.
- [34] S. Even. *Graph algorithms*. Cambridge University Press, 2011.
- [35] Y. Fang, X. Huang, L. Qin, Y. Zhang, W. Zhang, R. Cheng, and X. Lin. A survey of community search over big graphs. *The VLDB Journal*, 29(1):353–392, 2020.
- [36] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

- [37] S. Friedrichs and C. Lenzen. Parallel metric tree embedding based on an algebraic view on moore-bellman-ford. *Journal of the ACM (JACM)*, 65(6):1–55, 2018.
- [38] F. Garin and L. Schenato. A survey on distributed estimation and control applications using linear consensus algorithms. In *Networked control systems*, pages 75–107. Springer, 2010.
- [39] M. Ghaffari and J. Li. Improved distributed algorithms for exact shortest paths. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 431–444, 2018.
- [40] A. Goldberg and T. Radzik. A heuristic improvement of the bellman-ford algorithm. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1993.
- [41] W. Gropp, R. Thakur, and E. Lusk. *Using MPI-2: Advanced features of the message passing interface*. MIT press, 1999.
- [42] K. Group. Opencl overview - the khronos group inc. <https://www.khronos.org/opencl/>. (Accessed on 06/19/2020).
- [43] H. Guo, L. Huang, Y. Lü, J. Ma, C. Qian, S. Ma, and Z. Wang. Accelerating bfs via data structure-aware prefetching on gpu. *IEEE Access*, 6:60234–60248, 2018.
- [44] K. Gupta, J. A. Stuart, and J. D. Owens. *A study of persistent threads style GPU programming for GPGPU workloads*. IEEE, 2012.

- [45] M. B. Habbal, H. N. Koutsopoulos, and S. R. Lerman. A decomposition algorithm for the all-pairs shortest path problem on massively parallel computer architectures. *Transportation Science*, 28(4):292–308, 1994.
- [46] G. Hajela and M. Pandey. Parallel implementations for solving shortest path problem using bellman-ford. *International Journal of Computer Applications*, 95(15), 2014.
- [47] B. Hendrickson and T. G. Kolda. Graph partitioning models for parallel computing. *Parallel computing*, 26(12):1519–1534, 2000.
- [48] B. Hendrickson and R. W. Leland. A multi-level algorithm for partitioning graphs. *SC*, 95(28), 1995.
- [49] P. Holme and B. J. Kim. P. holme and bj kim, phys. rev. e 65, 026107 (2002). *Phys. Rev. E*, 65:026107, 2002.
- [50] S. Holzer and R. Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, pages 355–364, 2012.
- [51] Q.-S. Hua, H. Fan, L. Qian, M. Ai, Y. Li, X. Shi, and H. Jin. Brief announcement: A tight distributed algorithm for all pairs shortest paths and applications. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 439–441, 2016.
- [52] J.-F. Jenq and S. Sahni. All pairs shortest paths on a hypercube multiprocessor. 1987.

- [53] P. Jesus, C. Baquero, and P. S. Almeida. A survey of distributed data aggregation algorithms. *IEEE Communications Surveys & Tutorials*, 17(1):381–404, 2014.
- [54] M. Jian, L. Ke-ping, and L.-y. Zhang. A parallel floyd-warshall algorithm based on tbb. In *2010 2nd IEEE International Conference on Information Management and Engineering*, pages 429–433. IEEE, 2010.
- [55] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.
- [56] G. J. Katz and J. T. Kider Jr. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55. Eurographics Association, 2008.
- [57] J. W. Kim, H. Choi, and S.-H. Bae. Efficient parallel all-pairs shortest paths algorithm for complex graph analysis. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, page 5. ACM, 2018.
- [58] R. E. Korf and P. Schultze. Large-scale parallel breadth-first search. In *AAAI*, volume 5, pages 1380–1385, 2005.
- [59] V. Kumar and V. Singh. Scalability of parallel algorithms for the all-pairs shortest-path problem. *Journal of Parallel and Distributed Computing*, 13(2):124–138, 1991.

- [60] V. Kumar and V. Singh. Scalability of parallel algorithms for the all-pairs shortest-path problem. *Journal of Parallel and Distributed Computing*, 13(2):124–138, 1991.
- [61] S. M. Kumari and N. Geethanjali. A survey on shortest path routing algorithms for public transport travel. *Global Journal of Computer Science and Technology*, 9(5):73–76, 2010.
- [62] J. Kunegis. Konect - the koblenz network collection, 2013. URL <http://konect.uni-koblenz.de/>.
- [63] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 451–460, 2010.
- [64] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. <http://snap.stanford.edu/data>, June 2014.
- [65] Y. Lu, J. Cheng, D. Yan, and H. Wu. Large-scale distributed graph computing systems: An experimental evaluation. *Proceedings of the VLDB Endowment*, 8(3):281–292, 2014.
- [66] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

- [67] B. Lund and J. W. Smith. A multi-stage cuda kernel for floyd-warshall. *arXiv preprint arXiv:1001.4108*, 2010.
- [68] K. Madduri, D. A. Bader, J. W. Berry, and J. R. Crobak. An experimental study of a parallel shortest path algorithm for solving large-scale graph instances. In *2007 Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 23–35. SIAM, 2007.
- [69] A. Madkour, W. G. Aref, F. U. Rehman, M. A. Rahman, and S. Basalamah. A survey of shortest-path algorithms. *arXiv preprint arXiv:1705.02044*, 2017.
- [70] S.-T. Mak and K.-P. Lam. Serial-parallel tradeoff analysis of all-pairs shortest path algorithms in reconfigurable computing. In *2002 IEEE International Conference on Field-Programmable Technology, 2002.(FPT). Proceedings.*, pages 302–305. IEEE, 2002.
- [71] K. Matsumoto, N. Nakasato, and S. G. Sedukhin. Blocked united algorithm for the all-pairs shortest paths problem on hybrid cpu-gpu systems. *IEICE TRANSACTIONS on Information and Systems*, 95(12):2759–2768, 2012.
- [72] A. McLaughlin and D. A. Bader. Fast execution of simultaneous breadth-first searches on sparse graphs. In *Parallel and Distributed Systems (ICPADS), 2015 IEEE 21st International Conference on*, pages 9–18. IEEE, 2015.

- [73] H. Meyerhenke, P. Sanders, and C. Schulz. Parallel graph partitioning for complex networks. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1055–1064. IEEE, 2015.
- [74] M. Nakao, H. Murai, and M. Sato. Parallelization of all-pairs-shortest-path algorithms in unweighted graph. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, pages 63–72, 2020.
- [75] D. Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 565–573, 2014.
- [76] U. of Oxford. Jade.ac.uk. Online, 2020. URL <https://www.jade.ac.uk/>.
- [77] U. of Warwick. High performance computing orac. Online, 2017. URL <https://warwick.ac.uk/research/rtp/sc/hpc>.
- [78] T. Okuyama, F. Ino, and K. Hagihara. A task parallel algorithm for computing the costs of all-pairs shortest paths on the cuda-compatible gpu. In *2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 284–291. IEEE, 2008.
- [79] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano. The all-pair shortest-path problem in shared-memory heterogeneous systems. *High-Performance Computing on Complex Environments*, pages 283–299, 2013.

- [80] W. Peng, X. Hu, F. Zhao, and J. Su. A fast algorithm to find all-pairs shortest paths in complex networks. *Procedia Computer Science*, 9: 557–566, 2012.
- [81] W. Peng, X. Hu, F. Zhao, and J. Su. A fast algorithm to find all-pairs shortest paths in complex networks. *Procedia Computer Science*, 9: 557–566, 2012.
- [82] J. C. Platt. Fast embedding of sparse similarity graphs. In *Advances in neural information processing systems*, pages 571–578, 2004.
- [83] A. Pradhan and G. Mahinthakumar. Finding all-pairs shortest path for a large-scale transportation network using parallel floyd-warshall and parallel dijkstra algorithms. *Journal of Computing in Civil Engineering*, 27(3):263–273, 2013.
- [84] K. U. K. Reddy. A survey of the all-pairs shortest paths problem and its variants in graphs. *Acta Universitatis Sapientiae, Informatica*, 8(1): 16–40, 2016.
- [85] G. Rétvári, J. J. Bíró, and T. Cinkler. On shortest path representation. *IEEE/ACM Transactions on Networking*, 15(6):1293–1306, 2007.
- [86] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization, 2015. URL <http://networkrepository.com>.
- [87] A. Salwicki. Av aho, je hopcroft, jd ullman; the design and analysis of computer algorithms. *Mathematica Applicanda*, 14(28):139–142, 2016.

- [88] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming, portable documents*. Addison-Wesley Professional, 2010.
- [89] P. Sao, R. Kannan, P. Gera, and R. Vuduc. A supernodal all-pairs shortest path algorithm. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 250–261, 2020.
- [90] S. S. Skiena. The algorithm design manual,(2008). URL: <http://dx.doi.org/10.1007/978-1-84800-070-4>.
- [91] S. S. Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.
- [92] E. Solomonik, A. Buluç, and J. Demmel. Minimizing communication in all-pairs shortest paths. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 548–559. IEEE, 2013.
- [93] O. Taştan, O. C. Eryüksel, and A. Temizel. Accelerating johnson’s all-pairs shortest paths algorithm on gpu.
- [94] A. Tutor. Graph representation: Adjacency list and matrix. Online, 2020. URL <https://algorithmtutor.com/Data-Structures/Graph/Graph-Representation-Adjacency-List-and-Matrix/>.
- [95] S. M. Van Dongen. *Graph clustering by flow simulation*. PhD thesis, 2000.

- [96] C. Xie, L. Yan, W.-J. Li, and Z. Zhang. Distributed power-law graph computing: Theoretical and empirical analysis. In *Advances in Neural Information Processing Systems*, pages 1673–1681, 2014.
- [97] Q. Xu, H. Jeon, and M. Annavaram. Graph processing on gpus: Where are the bottlenecks? In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 140–149. IEEE, 2014.
- [98] Z. Yan and Q. Song. An implementation of parallel floyd-warshall algorithm based on hybrid mpi and openmp. In *Proceedings of the 2012 International Conference on Electronics, Communications and Control*, pages 2461–2466, 2012.
- [99] H. Yanagisawa. A multi-source label-correcting algorithm for the all-pairs shortest paths problem. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.
- [100] Y. Zhang and L. Wu. A novel algorithm for apsp problem via a simplified delay pulse coupled neural network. *Journal of Computational Information Systems*, 7(3):737–744, 2011.