PARALLEL NETWORK FLOW ALGORITHMS

by

Gökçehan Kara

B.S., Mechanical Engineering, Koç University, 2010

M.S., Computer Engineering, Boğaziçi University, 2014

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy

Graduate Program in Computer Engineering

Boğaziçi University

2022

PARALLEL NETWORK FLOW ALGORITHMS

APPROVED BY:

Prof. Can Özturan (Thesis Supervisor)	.....
Prof. Haluk O. Bingöl	.....
Assist. Prof. Didem Unat	.....
Prof. Z. Caner Taşkın	.....
Prof. Oya Ekin Karaşan	.....

DATE OF APPROVAL: 21.01.2022

ACKNOWLEDGEMENTS

I would like to thank my supervisor Prof. Can Özturan for keeping me motivated, fellow jury members Prof. Haluk O. Bingöl, Asst. Prof. Didem Unat, Prof. Z. Caner Taşkın, Prof. Oya Ekin Karaşan for all the feedback, my dear family for their endless support, schoolmates for the friendship, Polymer Research Center (PRC) and Telecommunications and Informatics Technologies Research Center (TETAM) in Bogazici University for providing the machines for experiments.

ABSTRACT

PARALLEL NETWORK FLOW ALGORITHMS

Network flows is an active area of research that has applications in a wide variety of fields. Several network flow problems are reduced to either the maximum flow problem or the minimum cost flow problem. Maximum flow problem involves finding the maximum possible amount of flow between two designated nodes on a network with arcs having flow capacities. Minimum cost flow problem tries to determine a flow with the minimum cost on a network with supply and demand nodes. In this thesis, we propose two parallel algorithms for the maximum flow and the minimum cost flow problems respectively. First, we present a shared memory parallel push-relabel algorithm for the maximum flow problem. Graph coloring is used to avoid collisions between threads for concurrent push and relabel operations. In addition, excess values of target nodes are updated using atomic instructions to prevent race conditions. The experiments show that our algorithm is competitive for wide graphs with low diameters. Second, we contribute a parallel implementation of the network simplex algorithm that is used for the solution of minimum cost flow problem. We propose finding the entering arc in parallel as it often takes the majority of the execution time. Scanning all arcs can take quite some time, so it is common to consider only a fixed number of arcs which is referred as the block search pivoting rule. Arc scans can easily be done in parallel to find the best candidate as the calculations are independent of each other. We used shared memory parallelism using OpenMP along with vectorization using AVX instructions. We also tried adjusting block sizes to increase the parallel portion of the algorithm. Our experiments show speedups up to 4 are possible, though they are typically lower.

ÖZET

PARALEL AĞ AKIŞI ALGORİTMALARI

Ağ akışı pek çok sahada uygulaması olan aktif bir araştırma alanıdır. Birçok ağ akışı problemi ya azami akış ya da asgari maliyet akışı problemine indirgenmektedir. Azami akış problemi kenarlarında akış kapasitesi olan bir ağ üzerindeki belirlenmiş iki düğüm arasında olası azami akışı belirlemek üzerinedir. Asgari maliyet problemi arz ve talep düğümleri olan bir ağ üzerinde asgari maliyetli akışı belirlemeye çalışır. Biz bu tezde azami akış ve asgari maliyet akışı problemleri için sırayla iki paralel algoritma sunduk. Birinci olarak azami akış problemi için paylaşımlı hafıza bir paralel itme-etiketleme algoritması sunuyoruz. Eş zamanlı itme ve etiketleme işlemleri için iş parçacıkları arasındaki çarpışmaları önlemek amacıyle çizge renklendirme kullanılmaktadır. Ek olarak hedef düğümlerdeki fazlalık değerleri yarışma durumlarını engellemek için atomik komutlarla güncellenmektedir. Deneyler bizim algoritmamızın geniş ve düşük çaplı ağlarda rekabetçi olduğunu göstermektedir. İkinci olarak asgari maliyet akış probleminin çözümü için ağ simpleks algoritmasının paralel bir uygulamasını sunuyoruz. Genellikle çalışma süresinin çoğunu aldığı için giriş kenarını paralel bir şekilde bulmayı öneriyoruz. Bütün kenarları taramak oldukça vakit alabilir, bu yüzden sadece sabit sayıda kenarı düşünmek yaygındır ki bu da blok arama eksen kuralı olarak adlandırılır. Hesaplamalar birbirinden bağımsız olduğundan kenar taramaları en iyi adayı bulmak için kolaylıkla paralel yapılabilir. Paylaşımlı hafıza paralellik için OpenMP ve bununla beraber vektörleştirme için AVX komutları kullandık. Ayrıca algoritmanın paralel miktarını arttırmak için blok büyüklüklerini ayarlamayı denedik. Deneyler hızlanmanın 4 kata kadar mümkün olduğunu fakat genellikle daha düşük olduğunu göstermektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES viii
LIST OF TABLES x
LIST OF SYMBOLS xii
LIST OF ACRONYMS/ABBREVIATIONS xiv
1. INTRODUCTION 1
1.1. Minimum Cost Flow Problem 2
1.2. Maximum Flow Problem 4
1.3. Shortest Path Problem 5
1.4. Parallel Programming 6
1.5. Contributions 8
2. RELATED WORK 10
2.1. Sequential Maximum Flow Algorithms 10
2.2. Parallel Maximum Flow Algorithms 11
2.3. Sequential Minimum Cost Flow Algorithms 14
2.4. Parallel Minimum Cost Flow Algorithms 15
3. MAXIMUM FLOW PROBLEM 17
3.1. Introduction 17
3.2. Algorithm 23
3.3. Implementation 33
3.4. Experiments 36
3.5. Conclusions 58
4. MINIMUM COST FLOW PROBLEM 60
4.1. Introduction 60
4.2. Background 61
4.2.1. Network Simplex Algorithm 62

4.2.2. Pivoting Operation 64
4.3. Parallel Block Searching 66
4.3.1. Shared Memory Parallelism 69
4.3.2. Vectorization 69
4.3.3. Block Sizes 72
4.4. Implementation 73
4.4.1. Data Alignment 74
4.4.2. Determinism 75
4.5. Experiments 76
4.5.1. Specifications 76
4.5.2. Dataset 77
4.5.3. Distributions 78
4.5.4. Iterations 81
4.5.5. Timings 81
4.5.6. Speedups 84
4.5.7. Vectorization 84
4.6. Conclusions 87
5. CONCLUSIONS 89
REFERENCES 91
APPENDIX A: COPYRIGHT NOTICE 103

LIST OF FIGURES

Figure 1.1. Example of a minimum cost flow problem along with its solution shown as highlighted 4

Figure 3.1. Conflict scenarios in parallel push-relabel algorithms 21

Figure 3.2. Colored Parallel Push-Relabel (cppr) Algorithm 24

Figure 3.3. Discharge Algorithm 26

Figure 3.4. Parallel Global Relabeling Algorithm 28

Figure 3.5. Avoidance of the example push-relabel conflict 30

Figure 3.6. Avoidance of the example relabel-relabel conflict 32

Figure 3.7. Speedups of cppr for vision/DIMACS/KaHIP problems compared to cppr and hpf 50

Figure 3.8. Distribution of number of nodes in color ticks in vision/DIMACS/KaHIP problems 57

Figure 4.1. Network Simplex Algorithm 64

Figure 4.2. Block Searching Algorithm 67

Figure 4.3. Sequential Block Searching Algorithm 68

Figure 4.4. An example entering arc selection 69

Figure 4.5. Parallel Block Searching Algorithm 71

Figure 4.6. Parallel Block Searching Algorithm (cont.) 72

Figure 4.7. Distributions of execution times for each step of the network simplex algorithm for each instance in our dataset 80

Figure 4.8. Relative numbers of iterations as percentages with increasing block size factors 82

Figure 4.9. Speedups of our implementation with increasing number of threads calculated according to lemon-ns implementation 85

Figure 4.10. Speedups of our vectorized implementation with increasing number of threads calculated according to our non-vectorized implementation 86

LIST OF TABLES

Table 3.1. Dimensions and coloring information for vision problems 39

Table 3.2. Dimensions and coloring information for DIMACS problems 40

Table 3.3. Dimensions and coloring information for KaHIP problems 40

Table 3.4. Sequential timings for vision problems 42

Table 3.5. Parallel timings for vision problems 43

Table 3.6. Sequential timings for DIMACS problems 44

Table 3.7. Parallel timings for DIMACS problems 45

Table 3.8. Sequential timings for KaHIP problems 46

Table 3.9. Parallel timings for KaHIP problems 46

Table 3.10. Timing distributions of cppr for different phases in vision problems 47

Table 3.11. Timing distributions of cppr for different phases in DIMACS problems 48

Table 3.12. Timing distributions of cppr for different phases in KaHIP problems 48

Table 3.13. Push counts for vision problems 51

Table 3.14. Relabel counts for vision problems 52

Table 3.15. Global relabel counts for vision problems 53

Table 3.16. Push counts for DIMACS problems 54

Table 3.17. Relabel counts for DIMACS problems 54

Table 3.18. Global relabel counts for DIMACS problems 55

Table 3.19. Push counts for KaHIP problems 55

Table 3.20. Relabel counts for KaHIP problems 56

Table 3.21. Global relabel counts for KaHIP problems 56

Table 4.1. Dimensions of instances used in the experiments 79

Table 4.2. Timings of all instances in our dataset 83

LIST OF SYMBOLS

A	Set of arcs in the graph
$A(x)$	Residual arcs corresponding to the current flow x
b_i	Flow balance of node i
c_a, c_{ij}	Cost of the arc with index a or from node i to node j
c_a^π, c_{ij}^π	Reduced cost of the arc with index a or from node i to node j according to potentials π
$color_i$	Color of node i
$curr$	Current arc index to start searching
d_i	Distance value of node i
e_i	Excess value of node i
$enter$	Entering arc index $.ind$ and its violation $.val$
grq_i	Global relabeling queue of thread i
G	Graph
$G(x)$	Residual network corresponding to the current flow x
k	Block size factor (i.e. $size \approx k \times \sqrt{m}$)
l_{ij}	Lower flow limit of the arc from node i to node j
L	Set of non-tree arcs with flows at lower bounds
m	Number of arcs in the graph (i.e. $m = \|A\|$)
n	Number of nodes in the graph (i.e. $n = \|N\|$)
N	Set of nodes in the graph
p	Number of threads
q_{ij}	Active node queue for color i of thread j
r_{ij}	Residual capacity of the arc from node i to node j
s	Source node in the graph
$size$	Number of arcs in a block
t	Sink node in the graph
tid	Thread id of the executing thread
T	Set of arcs in the current spanning tree

u_{ij}	Upper flow limit of the arc from node i to node j
U	Set of non-tree arcs with flows at upper bounds
x_{ij}	Flow on the arc from node i to node j
π_i	Node potential of node i
ϕ_a	Arc coefficient of the arc with index a for the spanning tree

LIST OF ACRONYMS/ABBREVIATIONS

AVX	Advanced Vector Extensions
FIFO	First-In-First-Out
OpenMP	Open Multi-Processing

1. INTRODUCTION

A *graph* in the most general sense is a model to represent pairwise relations between a given set of objects. *Graph theory* is one of the cornerstones of mathematics which started to gain traction in the 19th century. Since then, graphs have been studied extensively in discrete mathematics and used in various scientific disciplines. The rise of computers increased the popularity of the field even further most recently with internet applications. Today, graphs are everywhere and many applications of the daily modern life rely on graphs to be properly modeled and operated accordingly.

A *flow graph* or more commonly a *flow network* is a special graph to represent a flow between objects. *Network flows* as a subfield of graph theory and operations research started to gain traction in the second half of the 20th century. There is a wide range of engineering applications which make use of network flows to optimize the flow of an entity [1]. The canonical example is the *supply chain optimization* used for the distribution of a good over a given network of supply and demand nodes. The basic version of this application is formulated as the *minimum cost flow problem*.

The other major network flow problem in the literature is the *maximum flow problem* to find the maximum possible flow between two points. In addition, *shortest path problem* is also commonly studied in this field as it is commonly used in either the theoretical works or the implementations of network flow algorithms. These three problems are closely related to each other. It is easily shown that shortest path problem and the maximum flow problem can be converted to an equivalent minimum cost flow problem. However, in practice, this is rarely done as specialized algorithms usually perform better than general solvers.

1.1. Minimum Cost Flow Problem

Minimum cost flow problem is the most general form of the three network flows problems mentioned above. In this problem, each arc has a flow capacity and an associated cost denoting the expense of sending an additional unit of flow along that arc. At the same time, each node has an associated balance value denoting the desired difference between the *inflow* and *outflow* of the node in a *feasible flow* assignment. This is in contrast with the other two problems that usually have only two special nodes (i.e. source and sink) with non-zero associated balances. Having balance values for each node is a more general form and allows more flexible arrangements.

This problem closely mimics the scenario for minimizing the costs of distributing goods from producers to consumers on a network with shipment costs. Some of the terminology is borrowed from the real world for this reason. Nodes that have positive flow balance values are called *supply nodes* whereas those that have negative balance values are called *demand nodes*. Nodes with zero flow balances are called *transshipment nodes*. Not all types of nodes need to exist in a problem instance and these cases are sometimes referred with a special name. For example in *transportation problems* there are no transshipment nodes or in *circulation problems* there are only transshipment nodes.

Minimum cost flow problem is defined on a graph, $G = (N, A)$, where N denotes the set of nodes and A denotes the set of arcs. We refer to the number of nodes and arcs as $n = |N|$ and $m = |A|$ respectively. For each arc $(i, j) \in A$, c_{ij} denotes the cost of sending a unit of flow on an arc, l_{ij} denotes the lower flow limit of an arc, u_{ij} denotes the upper flow limit of an arc, and x_{ij} denotes the current flow of an arc. For each node $i \in N$, b_i denotes the desired difference between the outflow and the inflow of a node, which is positive for supply nodes, negative for demand nodes, and zero for transshipment nodes. There exists network transformation techniques to remove lower bounds from arcs [1]. In the rest of this thesis, lower bounds and zero values are

used interchangeably. With these definitions, linear programming formulation of this problem is given as

$$\text{minimize} \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{1.1a}$$

$$\text{subject to} \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = b_i \qquad \forall i \in N \tag{1.1b}$$

$$l_{ij} \leq x_{ij} \leq u_{ij} \qquad \forall (i,j) \in A. \tag{1.1c}$$

Equation (1.1a) is referred as the *objective function* defined using the total cost of each flow. Equation (1.1b) is the *mass balance constraint* denoting that the difference between the outflow and inflow of a node should be equal to the desired flow difference of that node in a feasible flow. Equation (1.1c) is the *flow bound constraint* denoting that the flow on an arc should be a value in between the lower and the upper flow limits of an arc. An implicit assumption is that the desired flow differences of all nodes should sum up to zero, otherwise there are no feasible solutions. We further assume that all arc and node values are in integer form. *Integrality property* holds that there is always an integer solution in this case.

An example minimum cost flow problem along with its solution is given in Figure 1.1. In the solution, each node satisfies the mass balance constraint and each arc satisfies the flow bound constraint. Also note that some of the arcs have zero flow in the solution meaning that the arc is not used in the optimum flow. We can calculate the total cost of the solution by multiplying the cost with the flow for each arc and sum the resulting values.

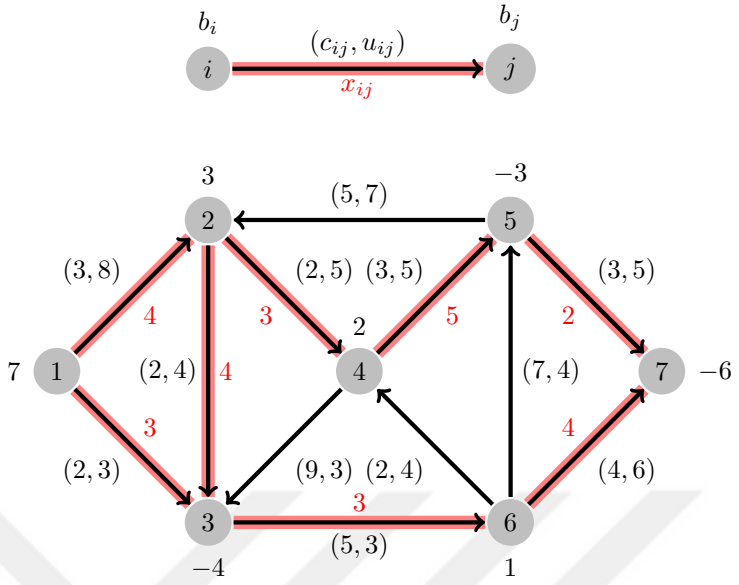


Figure 1.1. Example of a minimum cost flow problem along with its solution shown as highlighted. Mass balance and flow bound constraints are satisfied for all nodes and arcs respectively. Total cost of the solution can be calculated by multiplying the the cost with the flow for each arc to yield 84.

1.2. Maximum Flow Problem

The maximum flow problem is a commonly encountered optimization problem. It can be used to determine the maximum possible steady flow between two places on a network. Example networks include petroleum pipelines, road traffic, telecommunication, and electricity networks. Other than these, there are also less obvious problems that can be modeled as a maximum flow network. These include scheduling in parallel machines, rounding matrices, or selecting political representatives to name a few. Lastly, it is used as a subroutine in many algorithms for the minimum cost flow problem to find an initial feasible flow to work on. The minimum cost flow problem also has many applications from various domains. Refer to the reference book by Ahuja et al. [1] for more information about these applications.

The maximum flow problem is defined on a directed network with lower and upper bounds for the flow on each arc. Graph transformation techniques to remove lower

bounds from arcs for the minimum cost flow problem can be used for the maximum flow problem as well [1]. Two special nodes are given as the *source* and *sink* nodes on this network. The aim is to find a feasible flow assignment for each arc that maximizes the amount of flow between these two nodes. A feasible flow assignment is required to satisfy two types of constraints. First, the flow value on each arc needs to be less than the upper bound of that arc. Second, the inflow and outflow of each node needs to be equal to each other except for the source and sink nodes.

The source node is represented as s and the sink node is represented as t. The amount of flow from the source node to the sink node is represented with v. With these definitions, formulation of this problem is given as

$$\text{maximize } v \tag{1.2a}$$

$$\text{subject to } \sum_{\{j:(j,i)\in A\}} x_{ji} - \sum_{\{j:(i,j)\in A\}} x_{ij} = \begin{cases} -v & i = s \\ v & i = t \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in N \tag{1.2b}$$

$$\text{and } l_{ij} \leq x_{ij} \leq u_{ij} \qquad \forall (i,j) \in A. \tag{1.2c}$$

Using a simple transformation, we can see that maximum flow problem is a special instance of minimum cost flow problem. First, we set flow value of each node and cost of each arc to zero. Then, in order to satisfy the mass balance constraint, we add an extra arc from the sink node to the source node. Upper limit capacity of this extra arc should be infinite or at least larger than the maximum flow. Cost of this arc should be a negative value so the flow on this arc would be maximized in an optimum minimum cost flow. In the final solution, flow value on this arc corresponds to the maximum flow between the source and sink nodes.

1.3. Shortest Path Problem

Shortest path problem is a well-known problem in graph theory. In this problem, arcs have associated lengths and we are required to calculate the shortest route along

two points. In the domain of network flows, lengths are sometimes called costs, and the cheapest route is calculated in that sense. Linear programming formulation of this problem is given as

$$\text{minimize} \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{1.3a}$$

$$\text{subject to} \sum_{\{j:(i,j)\in A\}} x_{ij} - \sum_{\{j:(j,i)\in A\}} x_{ji} = \begin{cases} 1 & i = s \\ -1 & i = t \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in N \tag{1.3b}$$

$$0 \leq x_{ij} \leq 1 \qquad \forall (i,j) \in A. \tag{1.3c}$$

This formulation already shows the similarity between shortest path and minimum cost flow problems. All we need is to determine a source node and a sink node by setting the balance values of nodes accordingly and limit the upper flow bounds of arcs with one.

There is also a more general version of this problem for finding shortest paths from a single node to all other nodes. In the linear programming formulation for this problem we need to make a few changes. In the mass balance constraint, we should have $b_s = n - 1$ and $b_i = -1$ for all other nodes. In the flow bounds constraint, we need to remove the upper bounds by setting them to any number equal or larger than n. In terms of algorithms, this version of the problem is identical to the previous one.

1.4. Parallel Programming

Parallel computation is used to describe a computation in which multiple steps of an execution can be performed at the same time. This term itself has several meanings in practice at different levels of abstractions ranging from parallelism at the lowest hardware level to multiple machines connected over a network to be used for the same computation as in supercomputers. Parallelism is ubiquitous in processors in the form of *bit-level parallelism* and *instruction-level parallelism*. These types of parallelism are

often automatically performed by the computer and requires no explicit effort by the programmer, although carefully designed programs can yield much better performance. More recently, other forms of higher level parallelism have become more popular as the single core performance improvements started to stagnate. Two such common forms are *multi-core parallelism* in which multiple cores on the same chip are used and *multi-processor parallelism* in which multiple chips are used either on the same machine or separate machines. These forms of parallelization often requires explicit effort by the programmer.

Parallel programming is the study of designing algorithms and programs for parallel computation. Parallel programming requires careful examination of the underlying algorithm to avoid unintended behavior due to different execution orders of steps. Two such common errors are *race conditions* in which different execution orders produce different outputs and *deadlocks* in which multiple executions wait for each other to finish indefinitely. Such errors may be non-deterministic in practice meaning that they may not manifest themselves in all runs which can make them difficult to identify. Therefore, formal analysis may be desired in some cases to prove the correctness of the algorithm.

Race conditions happen when a shared resource is used concurrently by multiple executions. Part of a program in which a shared resource is used is referred as a *critical region*. Race conditions can be avoided by the use of *mutual exclusion* to access the critical region so that only a single execution can be in the critical region at any point. For shared-memory parallelism, mutual exclusion is commonly implemented either by the use of *locks* or *atomic instructions*. Locks are an abstraction often provided by the operating system as a coarse-grained mechanism to provide mutual exclusion to an arbitrary part of the program. On the other hand, atomic instructions are provided by processors with a guarantee of indivisible execution and they can be used as a fine-grained mechanism for mutual exclusion. Locks can be more expensive in terms of performance overhead compared to atomic instructions. The use of locks for synchronization also introduces the possibility of deadlocks in the program. Therefore, the

use of atomic instructions over locks are often preferred whenever possible. However, the use of atomic instructions for mutual exclusion can be limiting at times and may not always be possible. Algorithms without any lock usage are referred as *lock-free algorithms*.

OpenMP (Open Multi-Processing) is an application programming interface designed to support shared-memory parallelism with C, C++, and Fortran languages [2]. It consists of compiler directives to mark parallel sections in the code along with library functions and environment variables to control the runtime behavior of the program. Parallel sections are automatically converted to multithreaded code by the compiler. It is possible to maintain the sequential and parallel versions of a program in the same codebase by arranging the compiler to ignore OpenMP directives and using preprocessor macros for the rest of the library functions.

1.5. Contributions

Contributions of this thesis are as follows:

- We present two parallel algorithms for the two common network flow problems namely the maximum flow problem and the minimium cost flow problem. There are many algorithms proposed for the solutions of these problems. We have selected algorithms which are already competitive with their sequential implementations and suitable for parallelization. We parallelized the push-relabel algorithm for the maximum flow problem and the network simplex algorithm for the minimum cost flow problem.
- We have implemented these parallel algorithms and compared their performance to other existing sequential and parallel implementations. Our implementations are available online as open source with permissive licenses to help with future research on this topic [3, 4].
- Our implementations are robust in practice and they produce correct results for all examples in our experiments. We provide a formal correctness proof our

algorithm for the maximum flow problem. Our experience with other existing parallel implementations showed us that concurrency and memory bugs can be easily found in these types of algorithms. We have not observed any such bugs in our implementations during our experiments.

- We include graph instances having up to a billion arcs in our experiments. To our knowledge, there is no other prior study experimenting with the challenging billion arc problem.

Contents of this thesis has been published in two papers. Maximum flow algorithm as presented in Chapter 3 is published in [5], and the minimum cost flow algorithm as presented in Chapter 4 is published in [6]. Related work from these papers are given together in Chapter 2. Conclusions to the thesis are given in Chapter 5.

2. RELATED WORK

In this chapter, we review the literature for sequential and parallel network flow algorithms. Our intention here is to focus on the experimental work in the field instead of algorithmic complexity bounds. The famous book by Ahuja et al. [1] and an old survey for the maximum flow algorithms by Goldberg [7] can be consulted for the history of network flow algorithms given here and improvements in algorithmic bounds in time.

2.1. Sequential Maximum Flow Algorithms

There are three common classes of algorithms to solve the maximum flow problem. First, there is the *augmenting path algorithm* first introduced by Ford and Fulkerson [8]. This algorithm starts with a zero flow, and then tries to identify augmenting paths from the source node to the sink node. Maximum possible amounts of flow are sent along these paths iteratively. The algorithm finishes when there are no augmenting paths left from the source node to the sink node. The original analysis of the augmenting path algorithm suggests that the algorithm runs in pseudopolynomial time. However, further modifications have been proposed to improve this bound since then. Edmonds and Karp [9] show that the algorithm runs in polynomial time when the flow is augmented along shortest paths. Dinitz [10] independently presents a similar result using layered networks. Ahuja and Orlin [11] propose using distance labels instead of layered networks. Recent *Boykov-Kolmogorov algorithm* [12] is an augmenting path algorithm specialized for grid graphs. It has been shown experimentally that this algorithm is competitive for problems in computer vision domain [13].

Some other algorithms are based on *preflows*. The best known algorithm using preflows is the *push-relabel algorithm* by Goldberg and Tarjan [14, 15]. Two common implementations of the push-relabel algorithm involve using different strategies to select the next active node to be processed [16]. The *queue implementation* processes

active vertices using a first-in-first-out (FIFO) queue. The *highest label implementation* always selects the active node with the maximum distance using an array of sets for all possible distance values. There is also a third hybrid version namely the *wave implementation* which also maintains an array of sets but processes distances from higher to lower repeatedly instead of picking the highest available one each time. There are two common heuristics used to reduce the number of operations in the push-relabel algorithm. The *global relabeling heuristic* is used to periodically update distance labels to their exact values. The *gap relabeling heuristic* is used to relabel redundant nodes in bulk when a gap is found in between distance labels. Further improvements have been proposed to reduce execution times by pushing flow along a path of multiple arcs instead of a single arc [17–19].

Lastly, there is the *pseudoflow algorithm* introduced recently by Hochbaum [20, 21]. This algorithm and its simplex version are used to solve the maximum blocking cut problem which is equivalent to the minimum cut problem. The corresponding maximum flow can then be obtained from this solution if desired. Hochbaum and Orlin [22] later proposed a simplification further reducing the complexity of the pseudoflow algorithm. Chandran and Hochbaum [23] carry out an experimental study comparing the pseudoflow algorithm to the best known push-relabel algorithm. The results show that the pseudoflow algorithm performs better for most problem instances used in the experiments. A recent study suggests that the pseudoflow algorithm can also outperform the Boykov-Kolmogorov algorithm in most cases for computer vision problems [24].

2.2. Parallel Maximum Flow Algorithms

Anderson and Setubal [25, 26] propose an asynchronous multithreaded push-relabel algorithm based on locks. Active vertices are stored in a global queue, and processed in FIFO order. All threads participate in global relabelings which are run concurrently with push and relabel operations. To avoid collisions, a wave number is used that denotes the number of global relabeling visits on the node. A push operation requires the two nodes to be in the same wave. All synchronization is achieved through

the use of locks. The experiments show speedups of up to 9 are possible when using 16 processors. We note that these results are not likely reproducible anymore as the performance overhead of locks often makes them impractical nowadays and the state of the art research in the field mostly relies on lock-free algorithms.

Bader and Sachdeva [27] implement a cache-aware asynchronous multithreaded push-relabel algorithm based on locks. This work is an adaptation of Anderson and Setubal's previous work to the modern hardware of its time. Their cache-aware implementation uses contiguous allocation of memory for pair arcs to reduce cache-misses. In addition, they make use of the highest label node selection strategy instead of using a FIFO queue to process active nodes. The gap relabeling heuristic is also implemented concurrently to further improve the performance with this new strategy. The experiments show speedups around 2 to 4 are possible when using 8 processors. Similar to the previous work, this algorithm also use locks for synchronization which may not perform well in today's hardware anymore.

Hong and He [28, 29] introduce an asynchronous multithreaded push-relabel algorithm with a nonblocking global relabeling heuristic. The major contribution of this study is the use of atomic instructions to replace other synchronization methods such as locks and barriers. In order to handle concurrent push and relabel operations on the same node, they use an alternative termination criteria. It is shown by case analysis that arcs with invalid distance labels that are accidentally formed by these concurrent operations are all eliminated before the termination. For this purpose, they also pick the lowest neighbour for push operations instead of an admissible arc. The experiments show that it scales better than its lock based counterparts. They also experiment with an implementation of this algorithm on CPU-GPU-Hybrid systems [30].

Baumstark et al. [31] propose a synchronous multithreaded push-relabel algorithm. The main strategy in this work is to avoid explicit synchronization with locks and atomic accesses as much as possible. Instead, a deterministic winning criteria is used when two active nodes share a common arc for pushes and relabels. This is used

to avoid having wrong distance labels when a concurrent push and relabel operation is performed on the same node. To achieve this, processing is done using local variables denoting the value of excess and distance values at the last iteration. These values are then updated globally at the end of the iteration. Global relabelings are carried out in parallel in between certain iterations with the same frequency as Goldberg's highest label push-relabel implementation. The algorithm is implemented with C++ and OpenMP. Speedup ratios up to 12 are reported when using 40 threads on a self formed dataset. We note that there were some issues with this implementation in our experiments resulting in different solutions for a few instances likely due to race conditions and crashes for some other instances likely due to memory leaks so these results should be taken with a grain of salt.

The use of parallelization methods other than regular shared memory parallelism are rare. Halim et al. [32] use MapReduce to parallelize the augmenting path algorithm on distributed memory. This is used for extremely large sparse graphs from the internet with billions of arcs. The experiments show that the algorithm can finish in a reasonable amount of time. Caragea and Vishkin [33] implement the push-relabel algorithm targeting the XMT platform which is a PRAM-inspired many-core architecture. The experiments show speedups up to 4 are possible compared to Goldberg's highest label push-relabel implementation.

The minimum cut problem, as an equivalent to the maximum flow problem, is commonly used in computer vision, graphics, and medical imaging. Many problems in these domains are reduced to energy minimization, and graph cuts are calculated to solve these problems. Liu and Sun [34] implement a parallel Boykov-Kolmogorov algorithm. Delong and Boykov [35] propose a generalization of the push-relabel algorithm to work on regions on a graph which is suitable for parallelization. This method has been used to implement distributed memory algorithms [36, 37]. There exists studies that implement parallel push-relabel algorithms on GPUs [38–40]. All of these studies assume a grid graph, and implement specific techniques to exploit this structure. Typically they are aimed to solve problems on commodity hardware with limited memory.

Experiments often involve small problems that are to be solved within a time restriction in the order of milliseconds. Bigger examples are usually meant to surpass the available memory, and perform either disk input/output or swapping. Thus, it is difficult to make a direct comparison with general purpose maximum flow algorithms.

2.3. Sequential Minimum Cost Flow Algorithms

There is a wide range of minimum cost flow algorithms proposed in the literature. Sifaleras [41] provides a review for the minimum cost flow algorithms and available software for this problem. Kovács [42] conducts a comprehensible experimental study comparing the performance for all available solvers with various network instances. Results suggest that *primal network simplex algorithm* and *cost scaling algorithm* are amongst the most competitive algorithms for this problem. Their own implementation is often the most performant network simplex algorithm in the experiments. This implementation is available in LEMON graph template library as open source along with most other algorithms [43]. We used this implementation as a reference for comparison in our experiments and simply refer it as `lemon-ns` throughout the chapter. Refer to these papers for the state-of-the-art experimental knowledge about the minimum cost flow problem.

There has been a few recent experimental studies evaluating the performance of various minimum cost flow algorithms for a specific domain and/or a different problem variation. Vieira et al. [44] evaluate four algorithms for the minimum-cost flow problem on road networks and find that the network simplex is the best performing algorithm. Dong et al. [45] investigate multiple algorithms for the transportation problem from various domains and show the combinatorial methods such as network simplex and augmenting path based algorithms can consistently outperform numerical matrix-scaling based methods.

There are many studies using various approximation, heuristic or meta-heuristic methods to solve the more difficult variants of the minimum cost flow problem. These

variants include fixed costs, non-linear costs as convex or non-convex functions, multi-commodity flows, and constrained problems [46–49]. In this chapter, we only consider the basic minimum cost flow problem with single commodity and linear costs [50, 51].

There has been many recent studies adapting the network simplex algorithm to different fields. Modulo network simplex heuristic is used to solve the periodic event scheduling problem [52–56]. Holzhauser et al. [57] present a specialized network simplex algorithm for the budget-constrained minimum cost flow problem. Ryan et al. [58] introduce a simplex algorithm to solve uncapacitated pure-supply infinite network flow problems. Beckenbach [59] describe a network simplex algorithm for the minimum cost flow problem on graph-based hypergraphs. Zheng et al. [60] propose a two-layer virtual network mapping algorithm based on node attribute and network simplex for virtual network mapping. Nie and Wang [61] use network simplex algorithm to solve the continuous convex piecewise linear network flow problem. Lin et al. [62] use network simplex algorithm for balanced clustering. Among the theoretical work, Disser and Skutella [63] show the network simplex method can be used to solve, with polynomial overhead, any problem in NP implicitly during the algorithm's execution.

2.4. Parallel Minimum Cost Flow Algorithms

There has been a number of studies to parallelize the network simplex algorithm. Peters [64] experiments with different methods for parallel pricing on shared memory processors. Miller et al. [65] propose running multiple pivot searches in parallel and then performing pivotings sequentially. Thulasiraman et al. [66] present a method for concurrent pivoting to solve dual shipment problems. Barr and Hickman [67] report an implementation for running both pivoting and pricing operations in parallel. These studies are rather old and use lock-based synchronization which usually does not perform well in modern processors anymore. We found no trace of their implementations available online.

Jiang et al. [68] propose a multi-granular approach for parallel network simplex algorithm. Fine and coarse grained parallel strategies are to be used for shared and distributed memory parallelism respectively. A maximum speedup of 18.7 is reported in the paper with a mix of both strategies. While this study has a similar extent as ours, some of the details are not apparent in the experiments section. There is mention of different pivoting techniques in the paper, but it is not always clear which one is used for the presented results. We note that, best eligible pivoting is much more applicable for parallelization even though performance is not competitive compared to other efficient pivoting rules. Also, speedup values are calculated using the author's own sequential implementation and not by using the best and widely used existing sequential implementation in the literature. Additionally, there are no other performance figures of existing implementations in the results for comparison. Since the code is not available, we have no way of confirming these results and compare it with our own implementation. Experiments are only performed with 3 relatively small graphs which we expect to be solvable in a few seconds at most with a state-of-the-art performant solver. In our study, we restrained ourselves from presenting results for best eligible strategy even though it would show greater speedup values. We also calculate speedup values using the popular `lemon-ns` solver with the default block pivoting strategy, which is compared to other existing implementations in the original paper [42], so our implementation can also be transitively compared to other implementations. We also made our implementation available online to make sure results are easily reproducible [4].

To our knowledge, there are no other parallel network simplex algorithm study in the literature. Some of the other references specific to the discussion are given in later sections.

3. MAXIMUM FLOW PROBLEM

In this chapter, we present a graph coloring based parallel push-relabel algorithm for the maximum flow problem.

3.1. Introduction

Given a network and a flow on this network we can define a *residual network* to show remaining available capacities on arcs. Residual networks are represented as $G(x) = (N, A(x))$ where x is a flow on the network G. Residual networks have a one-to-one correspondence to a flow on the original network, and they can easily be converted to each other. Many algorithms are designed to manipulate these residual networks, and the corresponding optimum flow on the original network is calculated afterwards. We use the assumption that if $(i, j) \in A$ then $(j, i) \notin A$ without a loss of generality. This is only a notational convention, and most graph representations are able to handle parallel arcs. For each arc in the original network, we define at most two arcs on the residual network. A *forward arc* shows the available capacity left on the original arc with a residual capacity $r_{ij} = u_{ij} - x_{ij}$. A *backward arc* shows the flow on the original arc to be cancelled out if desired with a residual capacity $r_{ji} = x_{ij}$. These two arcs are referred to as *pair arcs* of each other. Residual networks consist of those arcs with a positive residual capacity.

A common algorithm for the maximum flow problem is the *push-relabel algorithm* introduced by Goldberg and Tarjan [14, 15]. This algorithm maintains *preflows* during the execution by relaxing mass balance constraints given in Equation (1.2b). The left hand side of this equation shows the difference between the inflow and outflow of a node. This difference is called the *excess value* of the node and it is represented as e_i for a given node i. Preflows allow $e_i \geq 0$ for all $i \neq s$. A given node i where $i \neq t$ is called an *active node* when $e_i > 0$. All active nodes are expected to be eliminated before

the algorithm ends to find a feasible flow. To achieve this, excess flow is repeatedly pushed to neighbouring nodes along individual arcs.

The push-relabel algorithm also assigns a *distance value* to each node which is represented as d_i for a given node i. Distances are sometimes also called heights to reflect the concept of flowing from a higher node to a lower node. A valid distance label assignment by definition needs to satisfy two types of conditions namely $d_t = 0$ and $d_i \leq d_j + 1$ for all $(i, j) \in A(x)$. Distance labels denote a lower bound rather than the exact value for the distance between the corresponding node and the sink node. Arcs that satisfy the condition $d_i = d_j + 1$ are named *admissible arcs*, and paths that consist of only admissible arcs are called *admissible paths*. An admissible path to the sink node is always a shortest path on the underlying unit weighted graph. When distance labels are exact, all admissible arcs are on a shortest path to the sink node. Since distance labels are not always exact, admissible arcs are only estimations of those arcs that are on shortest paths. For this reason, it is important to keep distance labels as close to their exact values as possible for better estimations.

The push-relabel algorithm consists of two basic operations.

Push operation increases the flow on an admissible arc that belongs to an active node. The amount of flow δ to be sent from an active node i on an arc $(i, j) \in A(x)$ is calculated with $\delta = \min\{e_i, r_{ij}\}$. A push operation can be *saturating* or *unsaturating*. In the former case, $\delta = r_{ij}$, and all available capacity on the arc is used. In the latter case, $\delta < r_{ij}$, and only the excess on the node is pushed along the arc.

Relabel operation is applied to an active node only when it has no outgoing admissible arcs. The distance value d_i of a given node i is updated to $d'_i = \min\{d_j \mid (i, j) \in A(x)\} + 1$ where d'_i represents the new distance value. Relabel operation ensures that $d'_i > d_i$ since admissible arcs are the only arcs which point to a neighbour with a lower distance value due to distance label conditions, and there are no admissible arcs when the node is eligible for relabeling.

There is some flexibility in choosing the order of push and relabel operations. In efficient implementations, each node has an arc list in a fixed order that is determined arbitrarily at the start of the algorithm. A *current arc* shows the current candidate on this list for a push operation. When a push operation is performed or it is not applicable, the current arc is changed to the next arc in the list. When the current arc reaches the end of the list, it is reset back to the beginning of the list. For this method, basic operations are usually combined into a single operation.

Discharge operation iterates over the arc list of a node, and performs all applicable push operations until the excess is zero or the current arc reaches the end of the list. If the current arc reaches the end of the list, then it is reset back to the beginning of the list, and a relabel operation is performed afterwards.

Discharge operations are applied repeatedly on active nodes during the algorithm. Initialization is slightly different in that the source node is discharged as if it has unlimited excess. Then the distance label of the source node is set to n to ensure that excess that can possibly flow to the sink node can never accidentally flow back to the source node.

Most parallel push-relabel algorithms aim to distribute nodes to threads, and run discharge operations in parallel. This approach has the following conflict scenarios to consider.

Push-push conflict occurs when two nodes sharing a common node try to push at the same time. In this case, race condition happens when these two nodes update the excess value on the common node. Depending on the execution order of instructions, the excess value might be set to an incorrect value. There are actually three variants of this conflict, two nodes share the same target node, one node shares the source node with the target node of the other, and two nodes shares the same source node. However, since we only consider node level parallelization, the last case is properly ordered, and there is no conflict.

Push-relabel conflict occurs when a push and relabel operation act concurrently on the same node. For this case, there must be no residual arc from the node that the flow is pushed to the node that pushes the flow. When the node that the flow is pushed is relabeled at the same time, node that pushes the flow might be overlooked, and distance label conditions might be violated after the push.

Relabel-relabel conflict occurs when two neighbour nodes are relabeled concurrently. This can happen when either one of the nodes have an arc to the other or both nodes have arcs to each other. During the relabel operation, a node can read the old distance value of the neighbour node that is also being relabeled, and possibly come up with a lower value.

Examples for these scenarios are shown in Figure 3.1. Note that relabel-relabel conflicts are actually benign in the sense that they do not cause any distance label violations. However, they can still be avoided to decrease the number of relabel operations performed. Push-push conflicts are usually avoided using atomic instructions in multithreaded algorithms. Push-relabel conflicts are the most complicated of all these scenarios.

One of the most common graph coloring methods is graph vertex coloring. Vertex coloring assigns colors to nodes in such a way that no two nodes that share an arc have the same color. Finding an assignment with the minimum number of colors is an NP-complete problem. In practice, there are fast polynomial algorithms to find approximate solutions using reasonable numbers of colors for most graphs. In the literature, vertex coloring is a common technique that is used to avoid conflicts in problems involving graph structures from various domains [69–72]. For parallel algorithms, coloring can be used for synchronization to properly order processing of nodes and arcs by conforming to a global color order at one point during the execution. Such a coloring strategy can be useful to prevent common concurrency problems such as race conditions and deadlocks.

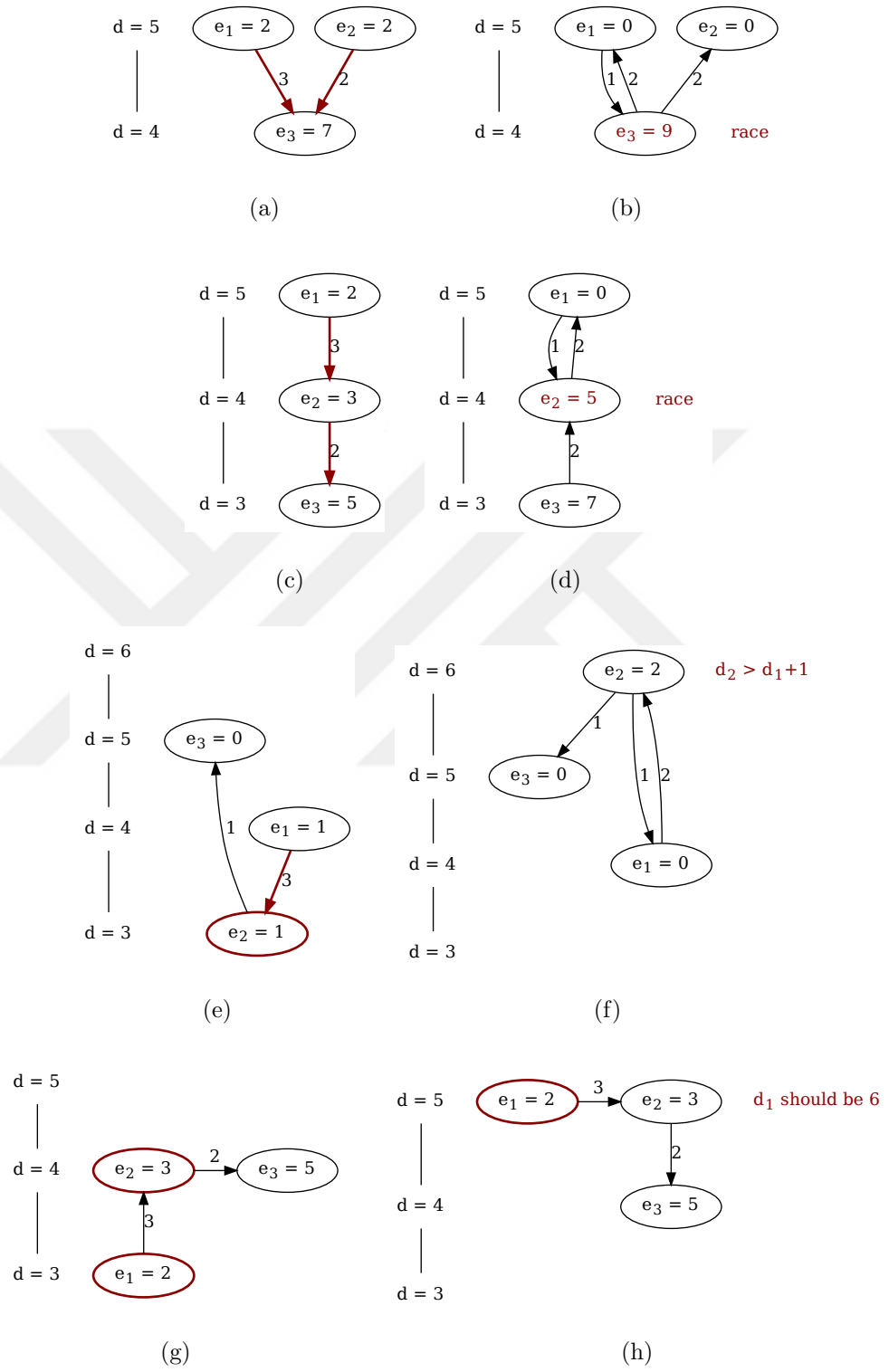


Figure 3.1. Conflict scenarios in parallel push-relabel algorithms, target-target push-push conflict (a, b), source-target push-push conflict (c, d), push-relabel conflict (e, f), and relabel-relabel conflict (g, h) (benign).

In this study, we present a multithreaded push-relabel algorithm. The novelty of our method is the use of graph vertex coloring to synchronize node processing. At one point during the execution, only vertices that belong to a single color are processed, and thus push-relabel conflicts are avoided. This approach also prevents relabel-relabel conflicts. For push-push conflicts, atomic instructions are used during target excess updates to prevent race conditions. There is no need to use atomic instructions for source excess updates since source-target push-push conflicts are also avoided with coloring. We refer to the algorithm as *colored parallel push-relabel (cppr)*, and the implementation is available online [3]. Our experiments show that high speedup rates are possible when there are enough active vertices for parallelization during the execution.

Novel features of our contributed algorithm are as follows:

- To our knowledge, graph coloring has not been used so far in the literature for the parallelization of the push-relabel algorithm. Our algorithm is lock-free and deterministic, and it uses only a single atomic instruction per push operation while discharging and similarly per distance update during global relabeling.
- We present formal correctness and termination proof of our algorithm. We have followed the algorithm thoroughly in our code. Our implementation is robust and completes with a correct answer for all examples we used in our experiments. In some examples, all other parallel implementations we tested [14, 29, 31, 37] either terminate prematurely without an answer, take too long to finish, or come up with wrong answers.
- Our experiments compare performances of sequential and parallel implementations for general purpose and computer vision specific algorithms that are often studied separately in the literature. We have experimented using examples from three different domains with graphs having up to a billion arcs.

This chapter is organized as follows. The use of graph coloring in our algorithm is discussed in Section 3.2. We discuss some of the implementation details in Section 3.3.

The results of our experiments from different domains are presented in Section 3.4. Some of the scalability issues and future works are discussed as conclusions in Section 3.5.

3.2. Algorithm

In this section, we present the colored parallel push-relabel algorithm, and show that it handles all conflicting cases properly. The highest label node selection is difficult to implement as an efficient parallel algorithm. Therefore, we used a queueing mechanism for active node selection in our algorithm. More information about data structures and other implementation details are discussed in the next section.

The general overview of the colored parallel push-relabel algorithm is given in Figure 3.2. The first thing we do is to color the vertices of the input residual graph (line 1). We used a regular sequential greedy vertex coloring algorithm for this purpose. Coloring is only done once at the start of the algorithm by considering all arcs in the residual network including the ones with zero capacity (i.e. on the underlying undirected graph that does not consider arc directions). Colors are simply denoted with integers. We check colors of neighbours for each node in the graph. If there is an available color that has been used before and none of the neighbours is assigned to this color yet, then we assign this color for the node. If there is no such available color, we increase the number of colors used, and assign the new color to the node. This algorithm runs in a fraction of the total execution time, and comes up with a few number of colors for most problems we used in our experiments.

Input: $G(x)$ where $x = 0$.

Output: The maximum flow and the corresponding residual graph $G(x)$.

```
1  Perform graph vertex coloring on G(x);
2  q_ij ← {} for all i ∈ {1, 2, ..., color_max} and j ∈ {1, 2, ..., tid_max};
3  foreach (s, j) ∈ A(x) do // initialization
4      δ ← r_sj;
5      r_sj ← r_sj − δ;
6      r_js ← r_js + δ;
7      e_s ← e_s − δ;
8      e_j ← e_j + δ;
9      if e_j = δ and j ≠ t then
10         q_{color_j,1} ← q_{color_j,1} ∪ j;
11     end
12 end
13 relabels ← n;
14 grq_i ← {} for all i ∈ {1, 2, ..., tid_max};
15 while q ≠ ∅ do // main loop
16     if relabels ≥ n then
17         relabels ← 0;
18         ParallelGlobalRelabeling(G(x), d, grq);
19     end
20     elems ← ⋃_{j=1}^{tid_max} q_ij where i is the next color;
21     parallel foreach i ← elems reduction (+ : relabels) do // iteration
        (color tick)
22         Discharge(G(x), i, d, e, q, relabels);
23     end
24 end
25 return e_t;
```

Figure 3.2. Colored Parallel Push-Relabel (cppr) Algorithm.

The queues are initialized at line 2, and then the source node is discharged while disregarding its excess value at lines 3-12. All arcs coming out from the source node are saturated, and all neighbouring nodes are activated. Since there is only one node being discharged, this phase is implemented sequentially to avoid parallelization overhead. All activated nodes are placed in the queue of the first thread. The global relabeling queues of threads are also initialized at line 14. After these initializations, the main loop starts (lines 15-24). We first trigger a global relabeling if the condition is met (lines 16-19). Global relabelings are triggered when the number of relabel operations from the last time exceed the number of nodes in the graph. To avoid race conditions in relabel counts, we use the reduction operation to sum thread local counts at the end of iterations. Initially we set this value to n to run a global relabeling at the beginning of the algorithm (at line 13). Active nodes are processed afterwards (lines 20-23). A global color value shows the current color of nodes to be processed. Active nodes of the current color are selected and distributed to all available threads. These nodes are discharged in parallel. Then the global color value is changed to the next color. This is counted as one iteration. We also call this color change a *color tick*. Iterations are performed until no active nodes are left of any colors.

The discharge operation is shown in Figure 3.3 in detail. This is implemented similar to the sequential version. The only difference occurs when the excess of a target node is increased (line 9). Since there might be several nodes that push to the same target node, the increment operation for the target excess value is implemented with an atomic instruction. During this update, the old excess value is kept in order to determine which thread pushed to the node before the others in case the node is activated. If the old value is zero, we say that the thread owns that target node and has the right to push it to its queue. Note that this happens only for a single thread due to the use of atomic instructions. This is used to avoid duplicate nodes when local queues are merged at the beginning of iterations. We also store the minimum distance value of neighbours during push operations to avoid going over arcs again for a possible relabel operation afterwards (line 17).

Input: $G(x)$, node to be discharged i, d, e, q, relabel count *relabels* (reduction variable).

Output: $A(x)$, r, d, e, q, and *relabels* are updated.

```
1  while true do
2      dmin ← ∞;
3      foreach (i, j) ∈ A(x) do
4          if d_i = d_j + 1 then // admissible arc
5              δ ← min{e_i, r_ij};
6              r_ij ← r_ij − δ;
7              r_ji ← r_ji + δ;
8              e_i ← e_i − δ;
9              atomic (old ← e_j; e_j ← e_j + δ);
10             if old = 0 and j ≠ t then // node is activated by the
                current thread
11                 q_{color_j,tid} ← q_{color_j,tid} ∪ j;
12             end
13             if e_i = 0 then
14                 return;
15             end
16         else
17             dmin ← min{dmin, d_j};
18         end
19     end
20     relabels ← relabels + 1;
21     d_i ← dmin + 1;
22 end
```

Figure 3.3. Discharge Algorithm.

For heuristics, we have only used global relabeling in our implementation. Gap relabeling is more useful for the highest label node selection [16]. Also, it is difficult to implement in parallel algorithms [31]. The global relabeling heuristic is implemented as a parallel reverse breadth-first search algorithm which is shown in Figure 3.4. Each level of the graph is processed in a separate iteration. Discovered nodes of all threads are merged, and then distributed back to threads at the beginning of each iteration. Distance updates are implemented as atomic instructions to determine which thread owns an undiscovered node (line 13). Distance value of a node is atomically set to the current level of the graph possibly by multiple threads. Meanwhile, the old distance value is stored to determine which thread discovered the node before others. If the old distance value is equal to the initial distance value, then the thread has the right to put the node in its queue. This technique is similar to what we have for the discharge operation, and it is used to avoid duplicates in the merged queue at the beginning of iterations. We allow threads to read the distance value non-atomically to determine whether a node is discovered or not since it is already corrected afterwards (line 12). This helps us avoid the cost of atomic instructions each time we check distance values.

As we mentioned before, there are some conflict scenarios to consider in parallel push-relabel algorithms. We now formally define these scenarios, and show that the colored parallel push-relabel algorithm handles these properly.

Definition 3.1 (Target-target push-push conflict)**.** *Let i, j, and k be three nodes on the residual network and (i,k) and $(j,k) \in A(x)$. Assume i and j are active nodes that are assigned to threads tid_i and tid_j respectively for discharging. A push along the arc (i,k) should decrease e_i and increase e_k by the same value. Similarly, a push along the arc (j,k) should decrease e_j and increase e_k at the same time. In that case, the order of increments on e_k are not defined, and a race condition occurs. This is referred to as a target-target push-push conflict.*

Lemma 3.2. *Target-target push-push conflict always ends up with the same outcome in the colored parallel push-relabel algorithm.*

```
   Input: G(x), d, grq.
   Output: d is updated.
 1 parallel foreach i ← 1 to n do // initialization
 2 |   d_i ← n;
 3 end
 4 d_t ← 0;
 5 grq_1 ← grq_1 ∪ t;
 6 level ← 0;
 7 while grq ≠ ∅ do // main loop
 8 |   level ← level + 1;
 9 |   elems ← ⋃_{i=1}^{tid_max} grq_i;
10 |   parallel foreach i ← elems do // iteration (wave)
11 |   |   foreach (i, j) ∈ A(x) do
12 |   |   |   if d_j = n then // undiscovered node
13 |   |   |   |   atomic (old ← d_j; d_j ← level);
14 |   |   |   |   if old = n then // node is discovered by the current
   |   |   |   |    thread
15 |   |   |   |   |   grq_tid ← grq_tid ∪ j;
16 |   |   |   |   end
17 |   |   |   end
18 |   |   end
19 |   end
20 end
```

Figure 3.4. Parallel Global Relabeling Algorithm.

Proof. Values handled by thread tid_i are e_i, e_k, r_{ik}, and r_{ki}. Similarly, values handled by thread tid_j are e_j, e_k, r_{jk}, and r_{kj}. The only common element on both of these lists is e_k. The colored parallel push-relabel algorithm uses atomic instructions to update target excess values. Therefore e_k increments are atomic instructions, and no race is possible during a target-target push-push conflict. When $e_k = 0$, either thread tid_i or tid_j may activate node k, and node k may be placed in either $q_{color_k tid_i}$ or $q_{color_k tid_j}$ depending on the execution order. However, thread queues are already merged together before iterations, therefore this difference is eliminated. □

Definition 3.3 (Source-target push-push conflict)**.** *Let i, j, and k be three nodes on the residual network, and (i,j) and $(j,k) \in A(x)$. Assume i and j are active nodes that are assigned to threads tid_i and tid_j respectively for discharging. A push along the arc (i,j) should decrease e_i and increase e_j by the same value. Similarly, a push along the arc (j,k) should decrease e_j and increase e_k at the same time. In that case, the order of updates on e_j are not defined, and a race condition occurs. This is referred to as a source-target push-push conflict.*

Lemma 3.4. *There is no source-target push-push conflict in the colored parallel push-relabel algorithm.*

Proof. Graph coloring assigns different colors to i and j since $(i,j) \in A(x)$. In the colored parallel push-relabel algorithm, only nodes from the same color are processed in a single iteration. Hence, i and j can not be processed in the same iteration, and the source-target push-push conflict is avoided. □

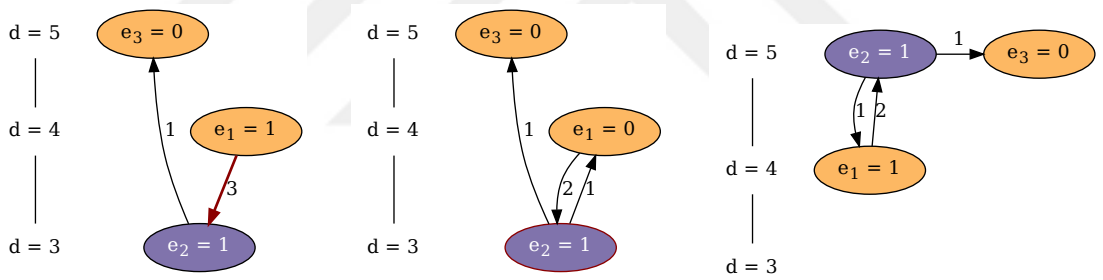
Definition 3.5 (Push-relabel conflict)**.** *Let i, j, and k be three nodes on the residual network, and (i,j) and $(j,k) \in A(x)$ but $(j,i) \notin A(x)$. In addition, let the distance values d_i and d_k satisfy $d_i = d_j + 1$ and $d_k = \min\{d_w \mid \forall (j,w) \in A(x)\}$ with $d_k > d_i$. Assume i and j are active nodes that are assigned to threads tid_i and tid_j respectively, and i is performing pushes while j is being relabeled at the same time. Since (i,j) is an admissible arc, i is allowed to push on this arc. At the same time, d_j is relabeled to $d'_j = d_k + 1$. After the push operation (j,i) is added to $A(x)$. Since $d_k > d_i$ and*

$d'_j = d_k + 1$, *we can infer* $d'_j \not\leq d_i + 1$ *meaning that the distance label condition for the new arc* $(j, i) \in A(x)$ *is violated. This is called a push-relabel conflict.*

Lemma 3.6. *There is no push-relabel conflict in the colored parallel push-relabel algorithm.*

Proof. With the same arguments given in the proof of Lemma 3.4, nodes i and j cannot be processed in the same iteration, and the push-relabel conflict is avoided. □

An example of how a push-relabel conflict can arise was shown in Figure 3.1. Figure 3.5 shows how this example push-relabel conflict is avoided by the use of graph coloring.



(a) (b) (c)

Figure 3.5. Avoidance of the example push-relabel conflict shown in Figure 3.1 (e, f). Graph coloring is used to properly order push and relabel operations. All such possible push-relabel conflicts are avoided by conforming to a global color order to process nodes.

Definition 3.7 (Relabel-relabel conflict)**.** *Let* i, j, *and* k *be three nodes on the residual network, and* (i, j) *and* $(j, k) \in A(x)$. *Also, let the distance values* d_k *and* d_j *satisfy* $d_k = \min\{d_w \mid \forall (j, w) \in A(x)\}$ *and* $d_j = \min\{d_w \mid \forall (i, w) \in A(x)\}$ *with* $d_j < d_w$ *for all* $(i, w) \in A(x)$ *where* $w \neq j$. *Assume that* i *and* j *are active nodes that are assigned to threads* tid_i *and* tid_j *respectively, and both of these nodes are being relabeled at the same time. After node* i *is relabeled, the new distance value is set to* $d'_i = d_j + 1$. *Given our assumptions, we can infer* $d'_i < d_w + 1$ *for all* $(i, w) \in A(x)$ *where* $w \neq j$. *Similarly,*

node j is relabeled to a new distance value $d'_j = d_k + 1$, and this new label should satisfy $d'_j > d_j$ by definition. Since $d'_i = d_j + 1$ and $d'_j > d_j$, we can infer $d'_i < d'_j + 1$. Hence, $d'_i < d_w + 1$ for all $(i, w) \in A(x)$ after these operations, which means that node i still has no admissible arcs, and it requires another relabeling. This is called a relabel-relabel conflict.

Lemma 3.8. *Relabel-relabel conflict is benign.*

Proof. These are two relabel operations where $A(x)$ does not change. Therefore, we only check if any existing arc violates its distance label condition with the new distance values. Since other distances are not changed, we only consider incoming and outcoming arcs of i and j. It is easy to show that all arcs other than (i, j) and (j, k) satisfy distance label conditions since we have $d'_i > d_i$ and $d'_j > d_j$ by definition. For arc (i, j), we have already inferred $d'_i < d'_j + 1$, and therefore the distance label condition still holds. For arc (j, k), we update the distance value of j to $d'_j = d_k + 1$ due to our assumptions, and therefore the distance label is satisfied. Hence, there are no distance label violations, and the relabel-relabel conflict is benign. □

Even though a relabel-relabel conflict is benign, it creates a redundant relabeling operation each time it occurs. Next, we show that our algorithm handles this conflict properly.

Lemma 3.9. *There is no relabel-relabel conflict in the colored parallel push-relabel algorithm.*

Proof. For a relabel-relabel conflict, we need to have (i, j) and $(j, k) \in A(x)$, therefore i and j should be assigned to different colors. Since i and j have different colors, they are processed in separate iterations, and no concurrent relabel operations are performed. Hence, relabel-relabel conflict is avoided in the colored parallel push-relabel algorithm. □

An example relabel-relabel conflict was shown in Figure 3.1. The use of graph coloring to avoid this case is shown in Figure 3.6.

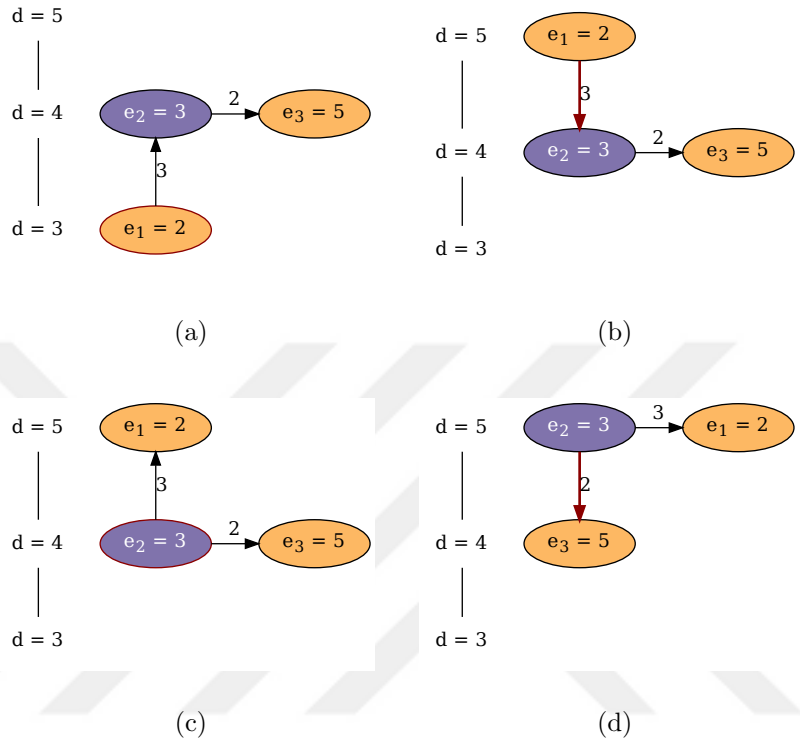


Figure 3.6. Avoidance of the example relabel-relabel conflict shown in Figure 3.1 (g, h). In the colored version, relabels of neighbour nodes are ordered properly. All such possible relabel-relabel conflicts are avoided by conforming to a global color order to process nodes.

Theorem 3.10. *The colored parallel push-relabel algorithm terminates with a feasible and optimal flow.*

Proof. Following Lemmas 3.2, 3.4, 3.6 and 3.9, the colored parallel push-relabel algorithm is reduced to the original push-relabel algorithm. By the correctness proof of the original algorithm, the colored parallel push-relabel algorithm terminates correctly with a feasible and optimal flow. □

Since we avoid push-relabel and relabel-relabel conflicts, our algorithm is deterministic in the sense that it always ends up with the same outcome and numbers of

operations performed at the end of a color iteration which is independent of the underlying scheduling. The work by Baumstark et al. [31] also has this deterministic property. This is usually an advantage, as there is no increase in the number of performed operations due to conflicting operations that arise because of using multiple threads.

Corollary 3.11. *The colored parallel push-relabel algorithm is deterministic.*

Proof. We only need to consider the push-push conflict since other conflicts are not possible. In one color iteration (lines 20-23 in Figure 3.2), multiple threads may push flow from the nodes with the same color to the same target node. Since, associative addition operation is performed when updating the excess value of the target node, the resulting excess value is independent of the order of execution of threads as well as the number of threads. Also, a push operation may create/update a reverse arc. Again since the active node, from which the push operation will be carried out, is assigned to one thread only, the result is deterministic. Whichever thread the active node is assigned, if a reverse arc is to be created/updated, it will be created/updated independent of the execution order of the threads and their numbers. Hence, the new residual graph that results after the color iteration is completed will be the same no matter how many threads we have. Hence, we conclude that the colored parallel push-relabel algorithm is deterministic. □

3.3. Implementation

The implementation is done with C++ using OpenMP for the parallelization. Atomic capture instructions are used in both distance updates during global relabeling and target excess updates during push operations. These instructions update the value of a given memory location atomically while also capturing the old stored value. Summing relabel counts at the end of iterations is implemented by a reduction clause with plus operator. The reduction clause creates local variables for each thread and

then combines these variables at the end of the loop. In our case, this is used to avoid the cost of atomic instructions to update a global count.

For the graph representation, we have experimented with both adjacency list and star graph representations. The adjacency list representation uses separate arrays to store adjacent arcs for each node. These arrays are usually spread to different locations in the memory since they are separate. In the star graph representation, there is only a single contiguous array to store all arcs in the graph. Arcs of a node are represented with a begin and an end iterator on this array. While the difference is small, the star graph representation almost always performs better than the adjacency list representation. Therefore, we used the star graph representation in our experiments. For each original arc in the input, we have a pair of residual arcs with opposite directions between the given nodes. Pair of an arc is stored as a pointer in the arc structure for easier update while pushing flow along the arc. We always keep all residual arcs even if they have zero residual capacities, and check if the capacity of an arc is positive during processing. Both of these representations are able to handle parallel arcs in the input graph between the same two nodes in either the same or opposite directions.

During discharging, we keep new activated nodes apart from others to be processed in subsequent iterations. Threads have local arrays to avoid synchronization overhead. In addition, nodes having different colors are held in separate arrays. In the end, we have a 2-dimensional array of arrays as the active node queue. At the start of each iteration, the next color is picked, and local arrays for that color are merged to a single global array. This array is then distributed evenly among all threads for processing. Dynamically growing arrays are used to avoid overflowing. Some amount of size is reserved for each array in the beginning to prevent most reallocations. The total amount of reserved size for all arrays is set to be equal to the number of nodes in the graph.

For global relabelings, the same queue is used for all runs. Similar to the discharging queue, threads have their own local arrays. These are merged to a single

global array, and then distributed among all threads at the start of each wave. Similarly, dynamically growing arrays are used in the global relabeling queue, and a total amount of memory with size equal to the number of nodes is reserved in the beginning. Unlike the discharging queue, we only have a 1-dimensional array since colors are not used in global relabelings.

We divide the execution into two phases to calculate the minimum cut first and then optionally get the maximum flow later. First phase is implemented simply by skipping nodes with distances bigger than or equal to the number of nodes in the graph. In the second phase, all the remaining active nodes are discharged regardless of their distances to push the excess flow back into the source node. Global relabeling is also a little different for the first and second phases. In the first phase, all node distances except for the source and sink nodes are initialized to the number of nodes in the graph and relabeling starts with the sink node. In the second phase, these distances are initialized to the double of the number of nodes and relabeling starts with the source node. In both cases, the source node is initialized to the number of nodes and the sink node is initialized to zero.

We use a round-robin order to process colors. In this schema, colors are processed in the same order repeatedly until no active nodes are left of any colors. If there is no active node of the current color, it is simply skipped to be checked again in the next round. We have also experimented with other strategies such as picking the color having the most active nodes, or the color having the highest total distance of its active nodes. Performance of these strategies differ from example to example but none of them is strictly superior to others. Therefore, we settled on using the robin-robin scheduling.

We used the default OpenMP static scheduling method for all parallel loops. We have also performed experiments with different OpenMP scheduling methods to have better load balancing. Our experiments show that the regular static scheduling is faster than other dynamic methods. The reason for this is that usually the work per

node is small enough that the overhead of dynamic scheduling methods becomes more significant. Also when there are enough active nodes for parallelization, differences among the number of arcs per node are easily averaged out when nodes are distributed to threads. On the other hand, parallelization in the arc level in addition to the node level complicates the synchronization of parallel algorithms significantly and it is unlikely to scale except for very specific examples designed for this purpose. These observations are in line with previous parallel push-relabel studies [73].

3.4. Experiments

Experiments were performed on a HPE ProLiant BL460c Gen9 Server Blade. There are two Intel Xeon CPUs E5-2650 v3 running at 2.30GHz attached to the machine. Each CPU has 10 physical cores, and 20 logical cores with Hyper-Threading. In our parallel runs, we used 20 threads at most which is equal to the total number of physical cores. Scalable memory allocator from Intel Threading Building Blocks (TBB) library is used to prevent false sharing. Our machine had 8x16GB DDR4 type RAM running at 2133MHz totaling up to 128GB which was enough to hold all tests in memory without swapping. Linux operating system is installed on this machine with kernel version 2.6.32. Programs used in our experiments are compiled with the highest optimization level using GCC version 5.4.0 which implements OpenMP version 4.0 using Posix threads. We enabled OpenMP processor binding option with automatic placement to set processor affinities.

We have also performed experiments with various sequential and parallel maximum flow solvers available. For the sequential algorithms, we used `fprf` and `hipr` implementations of the original push-relabel algorithm [16]. `fprf` uses a FIFO queue for active node selection, and it implements the global relabeling heuristic. This is the closest implementation to the sequential version of our algorithm. Similarly `hipr` uses the highest label node selection, and it implements both global relabeling and gap relabeling heuristics. Lastly, `hpf` is the original pseudoflow algorithm implementation

by the authors [23]. We used `pseudo_fifo` implementation which is specified as the fastest in general out of four available variants from the latest available version 3.23.

For our `cppr` implementation, we added results from both sequential and parallel versions. The sequential version is a standard non-colored push-relabel algorithm using the same discharging routine as the parallel version except for atomic instructions. It uses a regular sequential reverse breadth-first search algorithm for global relabeling. This version is included to reflect the parallelization overhead when all other things are equal. For the parallel version, we included results from two cases, running with only 1 thread, and running with 20 threads. For operation counts, we only show a single case for the parallel version since our algorithm is deterministic. Push-relabel timings are measured from the time when the residual graph is ready on the memory to the time when the algorithm comes up with a maximum flow value and assignment. Residual graphs are not used in `hpf` so we measure from the point when the graph is read to the memory.

For parallel solvers, we obtained the source code of the algorithm by Baumstark et al. [31] which we refer as `sync`. Authors also provided implementations for two other algorithms. First is Goldberg's original parallel push-relabel algorithm which we refer as `goldberg` [14]. Second is the implementation of the algorithm by Hong and He which we refer as `bohong` [29]. Lastly, we used `p-ard` algorithm for comparison which was shown to be among the most performant parallel algorithms for computer vision problems [37]. However, parallelization of this algorithm requires a preprocessing step to split the original graph into parts which we show separately from the execution time. This process can take a significant amount of time due to I/O operations.

Sequential solvers in our experiments use two-phase implementations that calculate the minimum cut in the first phase, and then get the maximum flow in the second phase using a more efficient method. Parallel solvers in our experiments calculate only the minimum cut but not the maximum flow. Our implementation uses the same method for the first and second phases. We present minimum cut and maximum flow

timings separately for a fair comparison. We used our maximum flow runs in the rest of the results besides timings.

Our first set of examples are maximum flow examples that are encountered in the computer vision domain. These consist of various types of problems that we have obtained online [74]. `camel` and `gargoyle` are multiview reconstruction instances, `bunny` is a surface fitting instance, and the rest are segmentation instances. Dimensions of these instances along with the number of colors used in greedy coloring and the number of iterations in our algorithm is shown in Table 3.1. Node degrees statistics are given in terms of mean (μ) and standard deviation (σ). Distance between the source and the sink nodes ($s - t$ distance) is the shortest distance measured on the initial residual graph with unit arcs.

Our second set of experiments are created using synthetic generators used in DIMACS challenge [75]. For `genrmf` examples, we have used dimensions of 8 and 4194304 for the long graph, and 5120 and 8 for the wide graph. Arc capacities are set to the range of 1 to 10000. For `washington` examples, we have used dimensions of 32 and 1048576 for the long graph and its reverse for the wide graph. Generating bigger examples turned out to be difficult without increasing the smaller dimension as generations already takes too long. Dimensions of the `washington-line-mod` instance is set to 262144 and 4. Arc capacities are set to the same range of 1 to 10000. Dimensions of generated instances along with coloring information is shown in Table 3.2.

Our third set of experiments are maximum flow problems converted from graph partitioning problems by the authors of KaHIP [76]. These problems are provided in their website for evaluation purposes [77]. For problems that have multiple instances with different dimensions, we used the biggest ones that fit into the memory of our test machine for all programs we used in our experiments. Among these problems, `del_strip` is a graph generated as Delaunay triangulations of random points in the unit square, `europe.osm` is a real world Open Street Map road network, `grid_strip` is a generated grid graph, `nlpkkt` is a graph generated from a sparse matrix example used

in nonlinear programming, and `rgg_strip` is a graph generated from random points in the unit square connected when the euclidean distance between two points is lower than a given threshold. Dimensions of these instances along with coloring information is shown in Table 3.3.

Table 3.1. Dimensions and coloring information for vision problems.

Problem	n	m	**degree** $(\mu \pm \sigma)$	$s-t$ **distance**	**# colors**	**# color ticks**
abdomen_long .n6c10	144M	867M	6.00 ± 0.07	23	4	549K
abdomen_short .n6c10	144M	867M	6.00 ± 0.07	19	4	15K
adhead.n26c100	12M	327M	26.02 ± 0.15	36	12	11K
babyface .n26c100	5M	131M	26.00 ± 0.03	14	16	781K
BL06-camel-lrg	18M	93M	4.35 ± 0.52	2	5	21K
BL06-gargoyle-lrg	17M	86M	4.05 ± 0.26	2	5	195K
bone.n26c100	7M	202M	26.01 ± 0.12	79	16	17K
LB07-bunny-lrg	49M	300M	6.03 ± 0.19	3	5	6153K
liver.n26c100	4M	108M	26.03 ± 0.19	6	13	155K

Table 3.2. Dimensions and coloring information for DIMACS problems.

Problem	n	m	**degree** $(\mu \pm \sigma)$	$s-t$ **distance**	**# colors**	**# color ticks**
genrmf-long	268M	1207M	4.50 ± 0.61	4194304	4	17258K
genrmf-wide	209M	1022M	4.87 ± 0.33	16	4	16K
washington-line-mod	1M	134M	127.96 ± 1.64	2058	55	252K
washington-rlg-long	33M	100M	3.00 ± 0.00	1048577	7	5393K
washington-rlg-wide	33M	99M	2.93 ± 0.34	33	7	98K

Table 3.3. Dimensions and coloring information for KaHIP problems.

Problem	n	m	**degree** $(\mu \pm \sigma)$	$s-t$ **distance**	**# colors**	**# color ticks**
del_strip26	40M	241M	5.99 ± 1.33	1354	7	58598K
europe.osm	15M	32M	2.12 ± 0.48	2712	5	13962K
grid_strip26	53M	214M	3.99 ± 0.02	6550	5	270124K
nlpkkt240	8M	222M	26.51 ± 2.32	44	10	2123K
rgg_strip26	40M	689M	17.12 ± 4.14	2396	25	3990K

Table 3.4 and Table 3.5 show timing results for vision problems. `fprf` is often slower than `hipr` but not with large margins. Our sequential implementation is usually faster than `hipr` for minimum cut calculation. However, it can get behind others for maximum flow calculations since we do not use an efficient second phase implementation. `hpf` is almost always the fastest sequential algorithm for these problems. For most problems, parallel implementations except for `bohong` can outperform `hpf`. Splitter of `p-ard` can take very large amounts of time especially for bigger examples. `goldberg` is faster than `sync` for all problems. Also `sync` crashes in two examples likely due to memory leaking, and comes up with slightly different solutions for two other examples due to race conditions. Between our algorithm and `goldberg`, there is no clear winner though `goldberg` is sometimes much faster. The parallelization overhead of our algorithm is around 28% for this set of problems.

Table 3.4. Sequential timings (in seconds) for vision problems (mincut/maxflow).

Problem	**cppr (seq)**	**cppr** ($p = 1$)	**fprf (seq)**	**hipr (seq)**	**hpf (seq)**
abdomen_long .n6c10	209/250	306/365	224/233	216/224	109/114
abdomen_short .n6c10	142/190	172/227	267/278	122/130	81/86
adhead.n26c100	63/75	85/100	125/134	120/125	41/51
babyface .n26c100	66/72	90/98	95/107	106/111	66/83
BL06-camel-lrg	130/147	165/184	161/164	198/200	28/44
BL06-gargoyle-lrg	138/158	188/212	172/174	129/130	31/40
bone.n26c100	38/45	52/62	82/86	46/48	5/8
LB07-bunny-lrg	151/180	188/222	291/302	349/359	26/101
liver.n26c100	56/61	60/65	70/72	49/50	19/20

Table 3.5. Parallel timings (in seconds) for vision problems (20 threads) (mincut/maxflow or only mincut) (asterisk (*) denotes different solution and dash (-) denotes crash).

Problem	**cppr** ($p = 20$)	**goldberg**	**bohong**	**sync**	**p-ard (+splitter)**
abdomen_long .n6c10	39/43	47	5490	-	38 (+501)
abdomen_short .n6c10	20/24	17	3860	-	15 (+504)
adhead.n26c100	9/10	7	977	11	55 (+172)
babyface .n26c100	21/22	6	710	10	77 (+84)
BL06-camel-lrg	13/15	15	1550	17*	15 (+40)
BL06-gargoyle-lrg	18/21	18	1760	21*	27 (+41)
bone.n26c100	5/5	3	395	14	15 (+102)
LB07-bunny-lrg	140/143	31	2150	41	3 (+175)
liver.n26c100	8/9	4	407	6	8 (+39)

Table 3.6 and Table 3.7 show timing results for DIMACS problems. In these experiments, our sequential implementation is slower than `fprf` especially for long graphs. `hipr` is much faster than `fprf` for long graphs. `hpf` is again the fastest sequential implementation except for `genrmf-long` example. For `genrmf` examples, parallel implementations other than our own have problems calculating the solution. These are big examples consisting of more than a billion arcs each. For `genrmf-wide` example, our algorithm with 20 threads can outperform sequential `hpf` algorithm, but this is not the case for `genrmf-long` example. For `washington-line-mod` example, our algorithm is slower than `goldberg` and `sync`, and these two are slower than `hpf`. For `washington-rlg` examples, our algorithm is faster or equal to other parallel al-

gorithms. Similar to `genrmf` examples, for `washington-rlg-wide` example, some of the parallel implementations including our own is able to outperform `hpf`, but not for `washington-rlg-long`. The average parallelization overhead for DIMACS examples is around 51%.

Table 3.6. Sequential timings (in seconds) for DIMACS problems (mincut/maxflow).

Problem	**cppr (seq)**	**cppr** ($p = 1$)	**fprf (seq)**	**hipr (seq)**	**hpf (seq)**
genrmf-long	110248/ 115027	222453/ 231240	87173/ 87222	167/204	128/ 15064
genrmf-wide	10806/ 10806	13532/ 13533	9821/ 9833	3730/ 3738	1425/ 1428
washington-line-mod	5/7	7/9	5/8	7/10	2/3
washington-rlg-long	1156/ 1177	1708/ 1774	186/190	23/26	12/18
washington-rlg-wide	554/562	855/866	609/611	508/510	115/118

Table 3.7. Parallel timings (in seconds) for DIMACS problems (20 threads) (mincut/maxflow or only mincut) (dash (-) denotes crash).

Problem	**cppr** ($p = 20$)	**goldberg**	**bohong**	**sync**	**p-ard (+splitter)**
genrmf-long	16770/17484	-	-	-	- (-)
genrmf-wide	864/864	-	13600	-	- (+770)
washington-line-mod	5/6	3	13	3	44 (+82)
washington-rlg-long	284/330	492	1220	473	24197 (+57)
washington-rlg-wide	73/74	70	6260	73	20160 (+84)

Table 3.8 and Table 3.9 show timing results for KaHIP problems. Our sequential implementation is faster than `fprf` which is faster than `hipr` but the fastest sequential implementation is again `hpf`. `goldberg` exits with an error for three problems in this set and it is slower than `sync` for the other two problems that it can run. Our implementation has slowdowns with increasing number of threads for three problems in this set. For these problems, our algorithm performs slower than `goldberg` and `sync` by a large margin and all parallel implementations are slower than `hpf`. For the other two problems, our algorithm have some speedups with increasing number of threads and it can outperform `hpf` but it is still slower than `sync`. The average parallelization overhead for KaHIP examples is around 14%.

Table 3.8. Sequential timings (in seconds) for KaHIP problems (mincut/maxflow).

Problem	**cppr (seq)**	**cppr** ($p = 1$)	**fprf (seq)**	**hipr (seq)**	**hpf (seq)**
del_strip26	462/494	478/507	685/696	2092/2099	216/282
europe.osm	25/30	38/47	37/38	263/264	10/12
grid_strip26	537/543	656/663	729/733	9608/9611	213/213
nlpkkt240	340/350	345/355	445/452	531/535	119/123
rgg_strip26	705/723	704/723	1416/1444	3847/3866	637/646

Table 3.9. Parallel timings (in seconds) for KaHIP problems (20 threads) (mincut/maxflow or only mincut) (dash (-) denotes crash).

Problem	**cppr** ($p = 20$)	**goldberg**	**bohong**	**sync**	**p-ard (+splitter)**
del_strip26	1266/1276	-	3550	82	1148 (+127)
europe.osm	185/274	23	236	14	57 (+19)
grid_strip26	5129/5130	349	6420	233	- (-)
nlpkkt240	69/70	-	3090	42	3198 (+126)
rgg_strip26	116/118	-	5480	75	756 (+314)

Timing distributions of our algorithm for different phases are given in Table 3.10, Table 3.11, and Table 3.12 respectively. For vision problems, coloring usually takes a small amount of time except for a few examples. Global relabel and discharge times are close to each other for most of these problems. For DIMACS problems, coloring times are mostly insignificant compared to total execution times except for `washington-line-mod` example which has the shortest execution time in this set. Most of the time is spent with discharging for these problems. For KaHIP problems, coloring

times are again insignificant especially for the three problems in which our algorithm shows slowdowns with increasing number of threads. For these three problems, almost all of the execution time is spent with discharging. For the other two problems, coloring phases are a little longer but they are still pretty short compared to total execution times. Discharge times are still dominant compared to global relabeling times for these two problems.

Table 3.10. Timing distributions (in seconds) of cppr ($p = 20$) for different phases in vision problems.

Problem	**coloring (seq)**	**discharging (par)**	**global relabeling (par)**
abdomen_long .n6c10	5.84 (14%)	19.93 (46%)	17.51 (40%)
abdomen_short .n6c10	6.40 (26%)	5.47 (23%)	12.24 (51%)
adhead.n26c100	1.93 (20%)	3.27 (33%)	4.68 (47%)
babyface .n26c100	0.73 (3%)	17.63 (80%)	3.76 (17%)
BL06-camel-lrg	0.75 (5%)	8.85 (59%)	5.48 (36%)
BL06-gargoyle-lrg	0.69 (3%)	15.07 (73%)	5.01 (24%)
bone.n26c100	1.21 (23%)	1.64 (31%)	2.45 (46%)
LB07-bunny-lrg	2.34 (2%)	132.06 (92%)	8.51 (6%)
liver.n26c100	0.76 (9%)	4.90 (55%)	3.21 (36%)

Table 3.11. Timing distributions (in seconds) of cppr ($p = 20$) for different phases in DIMACS problems.

Problem	**coloring (seq)**	**discharging (par)**	**global relabeling (par)**
genrmf-long	10.67 (0%)	13037.00 (75%)	4435.82 (25%)
genrmf-wide	20.06 (2%)	578.95 (67%)	265.17 (31%)
washington-line-mod	0.70 (11%)	5.15 (80%)	0.57 (9%)
washington-rlg-long	1.09 (0%)	232.70 (71%)	96.55 (29%)
washington-rlg-wide	2.14 (3%)	57.68 (78%)	13.99 (19%)

Table 3.12. Timing distributions (in seconds) of cppr ($p = 20$) for different phases in KaHIP problems

Problem	**coloring (seq)**	**discharging (par)**	**global relabeling (par)**
del_strip26	2.66 (0%)	1258.75 (99%)	15.11 (1%)
europe.osm	0.36 (0%)	271.17 (99%)	2.81 (1%)
grid_strip26	2.06 (0%)	5105.41 (100%)	23.25 (0%)
nlpkkt240	2.12 (3%)	55.73 (80%)	12.04 (17%)
rgg_strip26	5.87 (5%)	94.42 (80%)	17.35 (15%)

Speedup plots are given in Figure 3.7. These are calculated using the parallel version executed with one thread and `hpf` as the best sequential algorithm as references. For vision problems, the speedup trend seems to saturate as the number of threads increase. In two examples, `babyface` and `LB07-bunny-lrg`, we also see slowdowns after 8 threads. The average speedup ratio of the parallel implementation using 20 threads is around 7.3 compared to the parallel implementation running with a single thread, 5.7 compared to the sequential implementation, and 2.4 compared to hpf. For DIMACS problems, the scaling trend is more regular even after high numbers of threads, though there are two examples `washington-line-mod` and `washington-rlg-long` which shows slowdowns after some point. Average speedup rate is about 7.2 compared to the parallel implementation running with a single thread, 4.7 compared to the sequential implementation, and 0.6 compared to hpf. For only `genrmf-wide` and `washington-rlg-wide` examples in which our algorithm with 20 threads can outperform the sequential `hpf` algorithm, these numbers become 13.6, 9.8, and 1.6 respectively. For KaHIP problems, slowdowns start earlier than other problems. For three of these problems, speedups never exceed 2 and they are usually below 1. For the other two problems, speedups seem to peak at 8 or 16 threads. Average speedup rate is about 0.8 compared to the parallel implementation running with a single thread, 0.7 compared to the sequential implementation, and 0.3 compared to hpf. For only `nlpkkt` and `rgg_strip` examples in which our algorithm with 20 threads can outperform the sequential `hpf` algorithm, these numbers become 5.6, 5.6, and 3.1 respectively.

We measured operation counts of basic routines in our algorithm and compared them to the numbers reported by other algorithms. These operation counts turned out to be in accordance with execution times in general as expected. The difference between the operation counts of our sequential and parallel implementations are usually insignificant. In some cases, our parallel version has lower operation counts than the sequential version. This is due to changes in the operation order of active nodes in the parallel version which can occasionally lead to lower operation counts compared to the strict FIFO order.

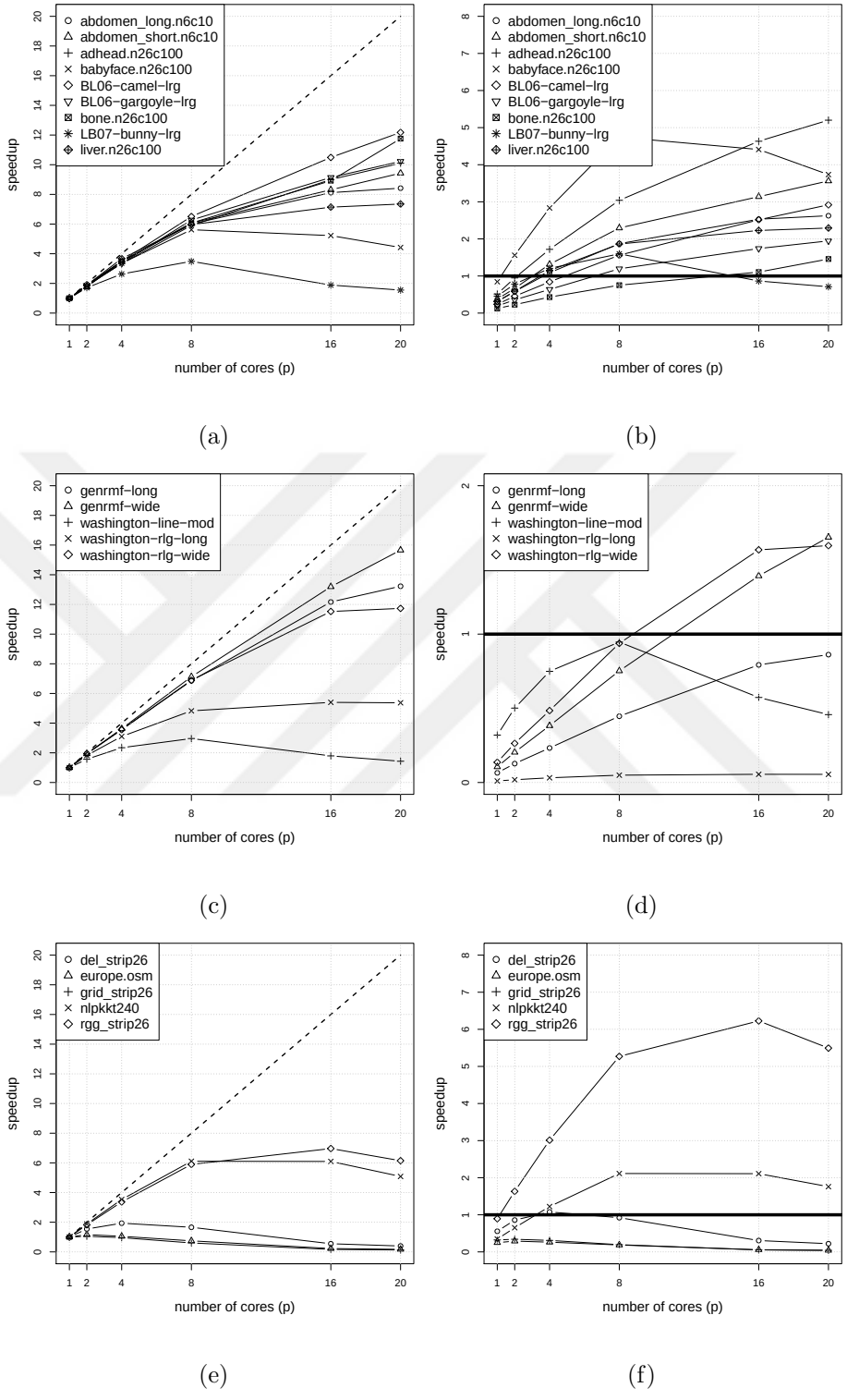


Figure 3.7. Speedups of cppr for vision problems compared to cppr ($p = 1$) (a) and hpf (b), for DIMACS problems compared to cppr ($p = 1$) (c) and hpf (d), and for KaHIP problems compared to cppr ($p = 1$) (e) and hpf (f).

Operation counts for vision problems are presented in Table 3.13, Table 3.14, and Table 3.15 respectively. For this set of problems, `fprf` usually performs fewer operations than the rest. Our algorithm performs similar numbers of operations as `hpf` on average. `hipr` usually performs the most number of operations for vision problems.

Table 3.13. Push counts for vision problems.

Problem	**cppr (seq)**	**cppr (par)**	**fprf**	**hipr**	**hpf**
abdomen_long .n6c10	465M	487M	277M	394M	269M
abdomen_short .n6c10	222M	224M	211M	308M	234M
adhead.n26c100	74M	74M	69M	112M	145M
babyface .n26c100	113M	112M	94M	134M	289M
BL06-camel-lrg	426M	426M	438M	869M	410M
BL06-gargoyle-lrg	552M	559M	515M	624M	454M
bone.n26c100	43M	43M	40M	40M	26M
LB07-bunny-lrg	664M	690M	485M	871M	111M
liver.n26c100	63M	64M	50M	58M	84M

Table 3.14. Relabel counts for vision problems.

Problem	**cppr (seq)**	**cppr (par)**	**fprf**	**hipr**	**hpf**
abdomen_long .n6c10	294M	295M	174M	297M	423M
abdomen_short .n6c10	149M	150M	144M	253M	275M
adhead.n26c100	40M	40M	37M	78M	61M
babyface .n26c100	49M	49M	40M	75M	103M
BL06-camel-lrg	151M	151M	151M	421M	211M
BL06-gargoyle-lrg	164M	167M	137M	309M	231M
bone.n26c100	25M	25M	23M	31M	11M
LB07-bunny-lrg	218M	218M	198M	511M	293M
liver.n26c100	36M	35M	29M	39M	31M

Table 3.15. Global relabel counts for vision problems (not applicable for hpf).

Problem	**cppr (seq)**	**cppr (par)**	**fprf**	**hipr**	**hpf**
abdomen_long .n6c10	4	4	2	3	n/a
abdomen_short .n6c10	3	3	2	2	n/a
adhead.n26c100	5	5	4	7	n/a
babyface .n26c100	11	11	9	15	n/a
BL06-camel-lrg	10	10	9	22	n/a
BL06-gargoyle-lrg	11	11	9	18	n/a
bone.n26c100	5	5	4	5	n/a
LB07-bunny-lrg	6	6	5	11	n/a
liver.n26c100	10	10	8	10	n/a

Operation counts for DIMACS problems are presented in Table 3.16, Table 3.17, and Table 3.18 respectively. In almost all cases, our algorithm has higher operation counts compared to others. `fprf` performs fewer operations compared to our algorithm but it still performs much more operations compared to others. `hipr` performs similar number of operations as `hpf` on average for this set of problems.

Table 3.16. Push counts for DIMACS problems.

Problem	**cppr (seq)**	**cppr (par)**	**fprf**	**hipr**	**hpf**
genrmf-long	469729M	453878M	352569M	849M	537M
genrmf-wide	15281M	15266M	10005M	9177M	52276M
washington-line-mod	5M	6M	5M	2M	2M
washington-rlg-long	9152M	9778M	5287M	124M	116M
washington-rlg-wide	1474M	1481M	1452M	1473M	445M

Table 3.17. Relabel counts for DIMACS problems.

Problem	**cppr (seq)**	**cppr (par)**	**fprf**	**hipr**	**hpf**
genrmf-long	28137M	27386M	29209M	436M	1117M
genrmf-wide	9009M	8863M	6504M	5257M	1327M
washington-line-mod	1M	1M	1M	1M	2M
washington-rlg-long	235M	234M	106M	40M	161M
washington-rlg-wide	503M	505M	503M	595M	331M

Table 3.18. Global relabel counts for DIMACS problems (not applicable for hpf).

Problem	**cppr (seq)**	**cppr (par)**	**fprf**	**hipr**	**hpf**
genrmf-long	106	103	109	2	n/a
genrmf-wide	43	43	32	26	n/a
washington-line-mod	3	3	2	2	n/a
washington-rlg-long	9	9	4	2	n/a
washington-rlg-wide	17	17	16	19	n/a

Operation counts for KaHIP problems are presented in Table 3.19, Table 3.20, and Table 3.21 respectively. Our algorithm has similar operation counts to `fprf`. `hpf` usually performs fewer push operations but more relabel operations compared to our algorithm and `fprf`. `hipr` performs the most number of operations by a large margin for this set of problems.

Table 3.19. Push counts for KaHIP problems.

Problem	**cppr (seq)**	**cppr (par)**	**fprf**	**hipr**	**hpf**
del_strip26	1151M	1148M	1092M	9656M	903M
europe.osm	285M	319M	235M	3652M	42M
grid_strip26	1857M	1964M	2511M	86515M	518M
nlpkkt240	351M	351M	343M	784M	217M
rgg_strip26	816M	720M	870M	6804M	527M

Table 3.20. Relabel counts for KaHIP problems (asterisk (*) denotes negative value is reported due to overflow).

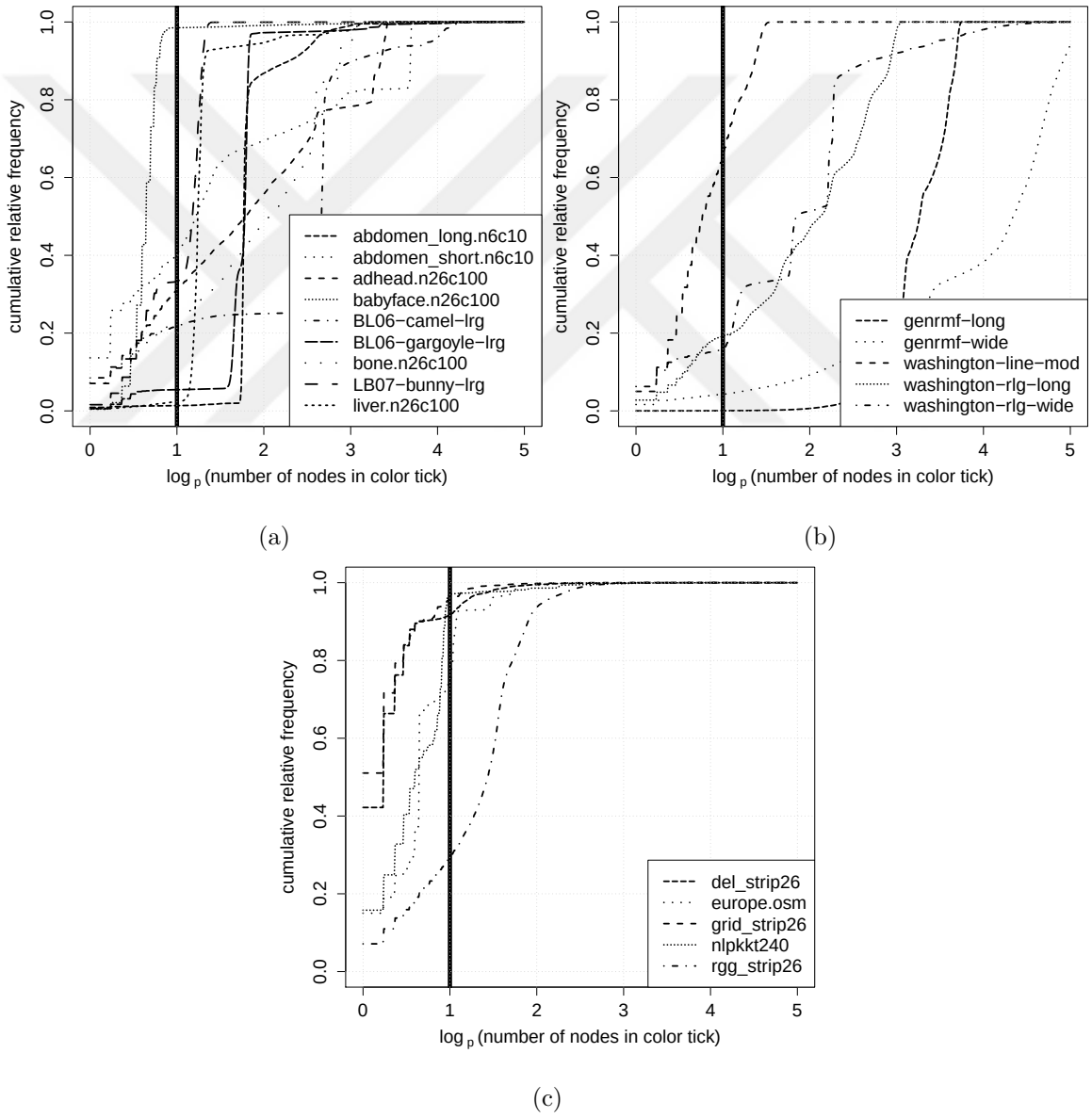
Problem	**cppr (seq)**	**cppr (par)**	**fprf**	**hipr**	**hpf**
del_strip26	845M	846M	793M	7373M	1561M
europe.osm	147M	165M	122M	1917M	93M
grid_strip26	912M	966M	1234M	43274M	1484M
nlpkkt240	134M	135M	134M	417M	187M
rgg_strip26	644M	564M	684M	6175M	-2015M*

Table 3.21. Global relabel counts for KaHIP problems (not applicable for hpf).

Problem	**cppr (seq)**	**cppr (par)**	**fprf**	**hipr**	**hpf**
del_strip26	23	23	20	185	n/a
europe.osm	11	12	9	127	n/a
grid_strip26	19	20	24	807	n/a
nlpkkt240	18	18	17	51	n/a
rgg_strip26	18	16	18	155	n/a

For some problems, parallelization is limited by the number of available active nodes of the same color at the start of each iteration for processing. If the number of active nodes of the current color is less than the number of threads then some of the threads stay idle until the next iteration. Similarly if the number of nodes is barely over the number of threads then the load distribution may suffer. For these cases, much of the work is spent for picking the next color, merging queues, and distributing the load which are all sequential. One of the possible reasons of this issue is the high number of colors used for graph coloring which is the case for dense graphs. Other than that, the problem instance may have few active nodes that is independent of colors which is the case for long graphs. Figure 3.8 show the relative cumulative distribution of number of active nodes of the same color recorded at each iteration. These are shown as log plots

using the number of threads as the base, so anything below 1 is a case in which some threads stay idle during the iteration. One such example is `washington-line-mod` instance which has about 70% of iterations below 1. This is a relatively dense graph which is colored with 55 colors. Indeed, our algorithm performs poorly on this example showing a slowdown in the parallel implementation as expected. However, also note that iterations with a lower number of active nodes have a lesser load to process, thus these plots are inherently pessimistic in nature.



(a) (b)

(c)

Figure 3.8. Distribution of number of nodes in color ticks in vision problems (a), DIMACS problems (b), and KaHIP problems (c) ($p = 20$).

3.5. Conclusions

We presented a parallel push-relabel algorithm. The experiments show that our algorithm is competitive for wide graphs with low diameters. Wide graphs usually have more active nodes for parallelization during the execution. On the other hand, having low diameter is important to have good performance when using FIFO queues for active nodes. We also show that our algorithm performs poorly for dense graphs in which the number of active nodes for parallelization is limited by colors and/or graph structure. For these examples, the pseudoflow algorithm is more suitable as it usually performs better even with a single thread. We note that there are many real world graphs that are wide and sparse and have low diameters. One can find such graphs for instance in social networks, artificial intelligence, bioinformatics, and auction problems. Hence our algorithm should be useful in practice.

We have a proof of correctness of our algorithm that is easy to follow and a corresponding implementation that is robust in practice. Our implementation is the only parallel implementation that is able to run successfully for all examples in our experiments. All other parallel implementation suffer from memory problems such as leaks or other logical errors such as races or distance label violations. Memory problems either cause crashes or high memory usage during the execution. Logical errors make them unreliable to use in an application where correctness is desirable. We note that fixing these errors may be difficult if not impossible and they can lead to losses in performance.

Performance of our sequential minimum cut computations seem to be a little better than our maximum flow computations compared to other algorithms. As we mention before, this is due to the use of a more efficient second phase implementation in Goldberg's algorithms to get the maximum flow from the minimum cut. This method can also be implemented to our sequential implementation to increase the performance further, though it is not easy to parallelize. The applicability of this method for our parallel implementation is still an open problem.

Our sequential minimum cut algorithms performs differently than `fprf`. This is most likely due to some of the implementation details we choose differently. Goldberg's algorithm uses a current arc pointer whereas our implementation starts each iteration from the beginning during discharging. This is a trade-off since the latter allows the minimum distance to be saved during the iteration and avoid an extra iteration in case a node is relabeled afterwards. In our experiments, we did not find a clear advantage among these two methods. In relation to that, we also noted that the order of arcs in the input file can also have a significant effect on the execution time.

In the future, experiments on bigger machines may reveal if the scaling trend continues as the number of threads increases. Our speedup rates usually show positive numbers in scaling for most problems. We have only identified the number of active vertices issue for scalability. As the number of threads increase, different limiting factors may come up. One particular possibility is the sequential greedy vertex coloring algorithm we used. This step mostly takes only a fraction of the total execution time except for a few examples. It may be worthwhile to use a different algorithm or parallelize the existing one to deal with Amdahl's law [78].

We carried out our experiments on a locally available computer that had 20 cores and 128 GB memory. Our algorithm may offer further advantages on NUMA shared memory based supercomputer systems that offer hundreds of cores and very large memories. On such systems, we may be able to increase network sizes dramatically. On very large problems, sequential algorithms may run into processing as well as limited cache bottlenecks whereas parallel algorithms can utilize hundreds of cores as well as the caches available on such systems.

4. MINIMUM COST FLOW PROBLEM

In this chapter, we present a new method for the parallelization of pivoting operation in the network simplex method.

4.1. Introduction

Simplex algorithm is an important method for many different optimization problems. This algorithm has been commonly used to solve linear programming problems. It is known that the minimum cost flow problem can be converted to an equivalent linear programming problem. Therefore, simplex algorithm can be used to solve the minimum cost flow problem. However, the resulting algorithm is not competitive in practice compared to other algorithms used to solve this problem.

Network simplex algorithm is an adaptation of the simplex algorithm to be used for network optimization problems. This algorithm uses special techniques to benefit from the special structure of the network to accelerate the regular simplex algorithm. Operations used in this algorithm closely resemble their counterparts in the simplex algorithm.

Finding entering arc operation is usually the bottleneck of this algorithm. Better strategies can significantly decrease the number of iterations performed in this algorithm. On the other hand, these strategies can be too expensive to provide a computational performance advantage. *Block searching* is a common strategy to have an acceptable number of iterations and to avoid taking too much time to find an entering arc. In this chapter, we propose parallelization of block searching operation to accelerate the network simplex algorithm.

We propose the following strategies for parallelization:

(i) Finding the arc with the most violation of optimality in a block can be searched in parallel with multiple threads using OpenMP.

(ii) Within each thread, this reduction operation can be vectorized using AVX instructions.

(iii) We can increase the block size to increase the work done for finding an appropriate arc but also heuristically decrease the number of iterations to come up with an optimum solution. In return, this helps us to trade some of the sequential portion of the program with parallel portion which is to be calculated with the additional computation power due to parallelization (i) and vectorization (ii). This is somewhat similar to Gustafson's law [79] which was proposed to remedy pessimistic result of Amdahl's law [78] when modeling speedup behavior of parallel applications. As we increase the total parallel work with increasing number of threads, the total sequential portion is decreased at the same time along with the number of iterations.

The rest of the chapter is organized as follows. Section 4.2 provides background for the minimum cost flow problem and the network simplex algorithm. Section 4.3 discusses our parallel block searching method. Section 4.4 provides implementation details of our solver. Section 4.5 presents our experimental results. Section 4.6 concludes the chapter and discusses future work directions.

4.2. Background

In this section we only provide a high level overview of the network simplex algorithm and proofs are left out for brevity. Seminal book by Ahuja et al. [1] is a slightly outdated but still a good introductory material on the topic which you can refer for more information.

4.2.1. Network Simplex Algorithm

Simplex algorithm is one of the most famous algorithms devised to solve optimization problems [80]. It has been originally invented to solve linear programming problems and then found many uses in various fields. Its success is largely due to providing exceptional efficiency in practice and being useful theoretically in sensitivity analysis and duality theorems.

Simplex algorithm can be used to solve network optimization problems, though it does not provide a competitive performance compared to other algorithms used for these problems. Network simplex algorithm is a specialized version of the simplex algorithm to exploit the network structure of the underlying optimization problem to have an efficient performance [81, 82]. For the minimum cost flow problem, network simplex algorithm is one of the fastest known algorithms, if not the best. This algorithm can also be seen as an efficient variant of *cycle cancelling algorithm*. The idea behind cycle cancelling algorithm is to start with a feasible flow and then converge to an optimum solution by repeatedly identifying cycles with negative costs at each iteration and then cancelling the flow on the cycle to get a better solution. Network simplex algorithm provides a fast method to identify such cycles in the graph. There is also another common variant of the algorithm called *dual network simplex algorithm* which starts with an optimum flow and converges to feasibility. In this chapter, we only consider the former *primal network simplex algorithm* which we simply refer as the network simplex algorithm.

Following concepts are defined to introduce the network simplex algorithm:

Free arc is an arc (i, j) in a feasible flow if $0 < x_{ij} < u_{ij}$.

Restricted arc is an arc (i, j) in a feasible flow if $x_{ij} = 0$ or $x_{ij} = u_{ij}$.

Spanning tree solution is a feasible flow and an associated tree structure in which every non-tree arc is a restricted arc. On the other hand, tree arcs can be free or restricted. When all tree arcs are free arcs, the tree is called *non-degenerate*,

otherwise it is called *degenerate*. To represent a spanning tree structure, we divide the arc set into three subsets (T, L, U) to denote tree arcs, non-tree arcs at lower bounds, and non-tree arcs at upper bounds respectively.

It can be shown that if a given minimum cost flow problem is feasible, then it always has an optimum spanning tree solution. We say that a spanning tree (T, L, U) is optimum if it is feasible and for some choice of node potentials π, reduced cost $c_{ij}^{\pi} = c_{ij} + \pi_i - \pi_j$ for each arc satisfies

$$\begin{aligned} c_{ij}^{\pi} &= 0 && \forall (i,j) \in T \\ c_{ij}^{\pi} &\geq 0 && \forall (i,j) \in L \\ c_{ij}^{\pi} &\leq 0 && \forall (i,j) \in U. \end{aligned}$$

Network simplex algorithm is based on the idea of maintaining a spanning tree solution. This spanning tree solution corresponds to a basic feasible solution in the simplex algorithm. At each iteration, the algorithm moves from a basic feasible solution to another by replacing an arc on the tree with another arc not on the tree. This operation is called pivoting as in the simplex algorithm. Pivoting is repeated until all arcs satisfy the optimality conditions given above. High level overview of the network simplex algorithm is given in Figure 4.1.

Data: Graph, node balances, arc costs and arc limits
Result: Optimum spanning tree solution
Find an initial feasible spanning tree solution;
Calculate arc flows and node potentials;
while *there is a non-tree arc violating optimality* **do**
    Select an entering arc violating the optimality;
    Add the entering arc to the tree;
    Identify the new cycle on the tree;
    Determine the leaving arc on the cycle;
    Update tree, arc flows and node potentials;
end

Figure 4.1. Network Simplex Algorithm.

4.2.2. Pivoting Operation

Pivoting is the central operation in the network simplex algorithm. It can be divided into multiple steps as follows:

Finding entering arc is the first step to find a non-tree arc violating the optimality. If there is no such arc, then the current solution is optimum. Hence, this step also serves as the termination condition.

Finding joining node is to identify the new cycle on the spanning tree formed by the addition of the entering arc. The entering arc is always on this cycle. A joining node is determined by travelling upwards in the tree starting from both ends of the entering arc. The cycle hangs from the joining node.

Finding leaving arc is to identify an arc with the minimum residual capacity that would first block sending additional flow along the cycle. There can be multiple such arcs, in which case, the leaving arc can be determined according to a leaving arc selection rule. Entering and leaving arcs can be the same arc, in which case, the arc moves between L and U. Residual capacity of the leaving arc can be

zero on degenerate trees which is referred as *stalling*. In this case, the spanning tree is changed but the solution is not improved. It is possible to loop forever in the algorithm by always picking the same arcs in consecutive iterations which is referred as *cycling*. However, there are known selection rules to avoid cycling by maintaining a *strongly feasible spanning tree*, in which case, the algorithm always terminate in a finite number of iterations [83, 84].

Updating tree is to update the flow values of arcs and the data structure holding the current spanning tree solution. This step is rather involved and depends on the choice of underlying data structures.

Updating potentials is the last step to update node potentials according to the new spanning tree. Removal of the leaving arc splits the old spanning tree into two subtrees and only one of the subtrees requires potential updates.

There is some flexibility in choosing the entering arc in the first step of pivoting. The other steps are more straightforward but still require clever data structures and strategies for efficient implementations. Determining an entering arc is performed using a pivoting rule. There are a number of pivoting rules proposed in the literature. Some of the well known ones are as follows:

Best eligible always selects an eligible arc of maximum violation, which is also known as Dantzig rule.

First eligible examines the arcs cyclically and selects the first eligible arc [85].

Block search cyclically examines a fixed number of arcs (blocks) and selects the best eligible candidate among these arcs [86].

Candidate list examines the arcs cyclically to build a list which is then used in a fixed number of subsequent iterations to select an arc of maximum violation in the list. This rule is first proposed by Srinivasan and Thompson [87] and later improved by Bradley et al. [88] and Mulvey [89].

Altering candidate list is similar to the candidate list rule except that it attempts to extend and filter the list at each iteration [90].

Violation of an non-tree arc depends on its reduced cost and the arc being in either L or U. We define a convenience variable ϕ_a where a is the index of an arc as

$$\phi_a = \begin{cases} 0, & a \in T \\ 1, & a \in L \\ -1, & a \in U. \end{cases}$$

Any arc with a positive violation can be selected as the entering arc in pivoting. The choice depends on the pivoting rule and there is no need to consider every arc at each iteration. Block pivoting technique which we consider in this study is given in Figure 4.2 and Figure 4.3. The search is performed block by block until an eligible arc is found or all arcs are examined. The current arc index is saved as *curr* at the end of each iteration to start the next iteration from where it is left last time in a cyclical manner. The number of arcs in a block is an adjustable parameter represented as *size*. At the end of each block search, the arc with the most violation in the block is saved as *enter* with its arc index as *enter.ind* and its violation as *enter.val*. If there are no eligible entering arcs, then *enter* is left zero as initialized at the beginning of each iteration.

4.3. Parallel Block Searching

The choice of pivoting rule has a significant effect on the number of iterations. Also, it can take the majority of the running time in the pivoting operation depending on the strategy. First eligible rule is the fastest to finish but results in a high number of iterations. On the other hand, best eligible rule is slow but finishes in fewer iterations. These two strategies are unlikely to be effective in practice. Experiments show that block searching as a hybrid strategy is faster than both strategies. Candidate list strategies are often on par with block searching and perform slightly better or worse depending on the input. We consider block pivoting in this study for being simple and efficient. It is also considered more robust than others so it is used as the default strategy in `lemon-ns` [90].

```
   Data: m, curr, size
   Result: curr and enter are updated
 1 Procedure find_entering_arc is
 2     begin ← curr;
 3     while begin < m and enter.val = 0 do
 4         end ← min{begin + size, m};
 5         block_search(begin, end);
 6         begin ← end;
 7     end
 8     if enter.val = 0 then
 9         begin ← 0;
10         while begin < curr and enter.val = 0 do
11             end ← min{begin + size, curr};
12             block_search(begin, end);
13             begin ← end;
14         end
15     end
16     curr ← end;
17 end
```

Figure 4.2. Block Searching Algorithm.

```
Data: c, φ, π
Result: enter is updated
1  Procedure block_search(begin, end) is
2      for a ← begin to end do
3          (i, j) ← a;
4          c ← φ_a × (c_a + π_i − π_j);
5          if c < enter.val then
6              enter.val ← c;
7              enter.ind ← a;
8          end
9      end
10 end
```

Figure 4.3. Sequential Block Searching Algorithm

Finding eligible arc is the only step that is easily parallelizeable in pivoting. Other steps involve either traversing or updating the spanning tree on the graph which makes them difficult to parallelize, if not impossible. Instead, we focused on minimizing the time spent in these steps by adjusting the pivoting rule parameters to decrease the number of iterations.

Calculations of arc violations are independent of each other. Properties of arcs are stored in continuous arrays and they can be accessed with the arc index. These properties include sources, targets and costs of arcs to calculate the reduced costs and ϕ values to denote the current spanning tree. Similarly, node potentials are stored in a contiguous array and they can be accessed with a node index. Given these values, we can simply calculate the reduced cost c_a^π of an arc using the formula given before. The reduced cost is then multiplied by ϕ_a to get the violation of the arc. Finally, all arc violations are reduced to a single value with the maximum violation having the minimum negative value. There are no eligible arcs if there are no negative values. The

reduction operation can be performed in parallel. An example entering arc selection is shown in Figure 4.4.

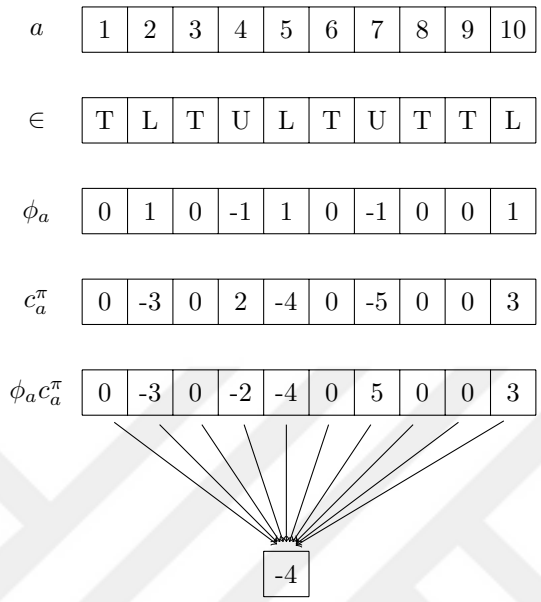


Figure 4.4. An example entering arc selection is shown. Violation calculations are independent of each other. Bottom row shows the violation of arcs. Violations are reduced to a single value with the most violation having the minimum negative value.

4.3.1. Shared Memory Parallelism

In a single iteration, the number of blocks to search might be one or many. Since we do not know the number of blocks beforehand, we used OpenMP to parallelize the search within a block. Small instances are unlikely to benefit from such parallelism, however we see increased speeds on bigger instances. OpenMP supports reduction clauses to provide race-free reduction operations without any synchronization overhead.

4.3.2. Vectorization

AVX (Advanced Vector Extensions) are extensions to the x86 instruction set to add SIMD (Single Instruction Multiple Data) parallelism. These extensions are successors to MMX and SSE (Streaming SIMD Extensions) introduced in previous generations. AVX and AVX2 instructions can use 256 bit registers to operate on

multiple elements. AVX512 instructions extend this further to 512 bit registers. These registers are divided according to the width of the underlying data type used in the calculation.

We have combined AVX instructions with OpenMP to further accelerate the search within a block. As the calculations are memory intensive, AVX instructions are only able to speed things up slightly. Also, as the number of threads increase, vectorization starts to compete with parallelization to get enough arcs to operate on. Hence, we see the speedup gains of vectorization disappear with increasing number of threads. However, vectorization is still beneficial to increase the utilizations of the cores by scaling up to the maximum possible speedups using fewer threads.

Our parallel and vectorized block searching is given in Figure 4.5 and Figure 4.6. Array notation is used to emphasize memory access patterns. AVX instructions use 512 bit vectors operating as 32 bit integers. Common prefixes `mm512` and suffixes `si512` and `epi32` are erased from the instruction names for brevity. Most of the work is performed in the first parallel loop. A custom minimum function for vectors named `vminf` is used in both the subsequent iterations of the loop and the final reduction of thread local vectors to a single shared vector. This final `mm512` vector is then reduced to a single `i32` integer in the second loop. Finally, leftover arcs are checked in the third loop when the number of arcs in the range is not a multiple of 16.

```
Data: tails array as arc source nodes, heads array as arc target nodes,
      states array as φ, costs array as c, pots array as π,
Result: enter is updated
1  Procedure block_search(begin, end) is
2      vmin ← {set1(0), set1(0)};
3      parallel for a ← begin to end by 16
4      reduction vminf : vminlastprivate a do
5          vind ← setr(a, a + 1, ..., a + 15);
6          vtail ← load(&tails[a]);
7          vhead ← load(&heads[a]);
8          vstate ← load(&states[a]);
9          vcost ← load(&costs[a]);
10         vpoti ← i32gather(vtail, &pots[0], 4);
11         vpotj ← i32gather(vhead, &pots[0], 4);
12         vcost ← add(vcost, vpoti);
13         vcost ← sub(vcost, vpotj);
14         vcost ← mullo(vcost, vstate);
15         vminf(vmin, velem{vind, vcost});
16     end
17     min ← {0, 0};
18     for i ← 0 to 16 do                                  // mm512 to i32
19         if vmin.val[i] < min.val then
20             min.val ← vmin.val[i];
21             min.ind ← vmin.ind[i];
22         end
23     end
24 ...
```

Figure 4.5. Parallel Block Searching Algorithm.

```
23 ...
24     for a ← a to end do                                  // leftover arcs
25         i ← tails[a];
26         j ← heads[a];
27         c ← cost[a] + pots[i] − pots[j];
28         if c < min.val then
29             min.val ← c;
30             min.ind ← a;
31         end
32     end
33     enter ← min;
34 end
35 Procedure vminf(out, in) is
36     vmask ← cmpgt_mask(out.val, in.val);
37     out.ind ← mask_blend(vmask, out.ind, in.ind);
38     out.val ← min(in.val, out.val);
39 end
```

Figure 4.6. Parallel Block Searching Algorithm (cont.).

4.3.3. Block Sizes

Block selection works with a parameter to determine the block sizes. An efficient value is to use $\sqrt{m}$ as the block size [90]. We see that this value is usually optimum for most instances in our dataset. However, this optimum changes naturally when we introduce parallelism. Searching for an entering arc finishes much faster and the portion of time spent decreases with increasing number of threads. Theoretically, block size should be increased by a factor of p, the number of threads, to keep the same distribution of time for each step if we assume perfectly linear speedups for block searching. In practice, speedup values are lower since the computation is short

and memory intensive. Therefore, we use a different factor k for block sizes (i.e. $size = k \times \sqrt{m}$).

On the other hand, we also expect a decrease in the number of iterations in return with increasing block sizes, otherwise the extra work is wasted. We see various behaviors across different instances in our dataset. Number of iterations decrease dramatically for some instances but not so much for others. Therefore, it is usually required beforehand to determine whether or not block size increments are beneficial for the input instance at hand.

Amdahl's law assumes that the fraction of the inherently sequential portion of an application is fixed which in turn implies a pessimistic result that limits the maximum theoretical speedup to be bounded by the reciprocal of the sequential portion of the program [78]. Our discussion is similar to Gustafson's law, which states that the problem size scales with the number of processors in practice, but we also decrease the total sequential work performed at the same time [79]. By increasing block sizes, we expect to perform better pivots and decrease the number of iterations. Increasing block sizes is equivalent to increasing the portion of the program which we can parallelize, and decreasing the number of iterations means there is less sequential work. In return, we trade the sequential portion of the execution with more parallel portion to achieve better scaling. Note, this conversion is often not free and there is some overhead involved in such parallelism.

4.4. Implementation

We have implemented the algorithm in C++ language in both sequential and parallel versions. Parallel versions are implemented using OpenMP with and without AVX2 and AVX512 instructions. Intrinsic functions are used for vector instructions. AVX2 is added mostly as a fallback since AVX512 is relatively new and not widely supported yet. AVX2 and AVX512 have equivalent instructions for 256 and 512 bit operations respectively, though there are slight differences. We have not used older

vectorization instructions such as AVX and SSE since gather instructions are missing in them. Our implementation in the form of a standalone DIMACS file format solver named `pns` is made publicly available online [4].

Our implementation mostly follow the text book definition of the algorithm [1]. However, some implementation details are often not provided in most text books. Kelly and ONeill [91] present more details than most other materials on the topic. We have also used the `lemon-ns` implementation extensively as a reference as it has efficient implementations for most procedures [90]. In particular, the data structure holding the spanning tree as an improved version of the extended threaded index first proposed by Barr et al. [92] along with the tree update operation has many optimizations to minimize the amount of work performed. We directly adapted the same tree code in our implementation. Similarly, we use the same artificial initialization approach for the initial feasible solution. However, the two implementations are not identical in general as `lemon-ns` aims to be more flexible as a library. We have used `lemon-ns` as reference in our experiments since there are differences in performance between the two implementations. `lemon-ns` is compared to other implementations by Kovács so our implementation can also be transitively compared to others [42].

4.4.1. Data Alignment

Vector instructions have separate instructions for aligned and unaligned memory accesses. On some processors, aligned memory access is expected to be faster. Data needs to be aligned accordingly to be able to use aligned memory access instructions. For AVX2 and AVX512 instructions, the alignment requirements are 256 and 512 bits respectively. Similarly, processors typically have 64 byte (512 bit) aligned cache lines. For this reason, we aligned our arrays with 512 bit alignment.

We used structure of arrays as it is often more appropriate for vectorization compared to array of structures [93]. Data is held as 32 bit integers in our arrays. With 512 bit capacity, a single cache line can hold 16 elements at once. So we used 16

element blocks for OpenMP scheduler to avoid false sharing. Block size is also rounded up to a multiple of 16 when possible. However, the effect of this change is hardly any noticeable as our calculations are mostly bottlenecked by indirect memory accesses rather than direct memory accesses.

4.4.2. Determinism

Deterministic algorithms are sometimes preferred over non-deterministic ones to perform consistently. This can become a challenge for parallel algorithms. There has been various weak and strong definitions of determinism in the literature [94, 95]. The network simplex algorithm has a possible non-determinism scenario due to multiple arcs having the same maximum violation. In a sequential algorithm, this non-determinism is avoided by always selecting the first or last arc in such cases. As a result, subsequent invocations of the algorithm always execute the same number of iterations.

Our regular sequential implementation is also deterministic. Vectorized implementations are deterministic in the sense that subsequent invocations always result in the same number of iterations. However, this number may differ slightly from the normal sequential implementation due to variations in vector reductions. It is trivial to avoid this difference by picking the arc with the minimum index among equals during vector reductions. Our parallel implementations are similarly non-deterministic due to variations in reductions. This can be avoided if the underlying OpenMP implementation support deteministic reductions (e.g. Intel OpenMP).

Deterministic reductions can be important to always finish with the same result when the calculations involve floating point arithmetic. In our case, we only have integer arithmetic so there is no such difference in the final result. The only difference is the number of iterations performed to get to the final result. For all instances we used in our experiments, we did not observe a significant difference in the number of iterations. Therefore, we have not made any effort to make our executions deterministic.

4.5. Experiments

In this section, we provide timing results along with various other aspects to provide insight about cases in which our implementation is competitive. Such insight is necessary to decide whether or not to use our method in accordance with the input instance at hand as our algorithm does not perform well for all cases. Our general consensus is that parallelization is worthy when there are enough arcs in the input instance and block size increments are often beneficial than not.

We only show results for block size factor 1, 4, and 16 to avoid over-optimizing according to the instances in our dataset. These values can be considered small, medium, and large block sizes respectively. Further improvements might be possible using a finer grained parameter set with respect to the input instance.

Timings and speedups are only shown with the vectorized and parallel version of the program for brevity. However, we also include separate results to show the effect of vectorization. AVX2 and AVX512 show similar improvements so we only show the results for AVX512 and kept AVX2 version of the code only for compatibility.

We included three algorithms from LEMON library as references, network simplex (i.e. `lemon-ns`), cost scaling (i.e. `lemon-cos`), and capacity scaling (i.e. `lemon-cas`). These are referred as the fastest algorithms for the minimum cost flow problem by the authors [42]. We used these algorithms with their default parameters including block pivoting strategy for `lemon-ns`. We used a time limit of 4 hours for each run which only became an issue with `lemon-cas` on some instances.

4.5.1. Specifications

We run the experiments on a system with 4 x Intel Xeon Gold 6238 CPUs each having 22 physical cores and 44 threads totaling up to 176 threads. The system had 1TB memory attached. It was running Ubuntu 20.04.02 operating system with Linux

5.8.0-36 kernel. All programs were compiled using GCC 9.3.0 with -O3 optimizations. We used Boost 1.71.0 for memory alignment. LEMON implementations use LEMON version 1.3.1.

The machine is a shared multi-user system but it was mostly idle except for a few other occasional light jobs throughout our experiments. We used OpenMP dynamic scheduling with a block size of 16. Threads were mapped to the same socket and processor binding option was enabled to prevent thread migrations. Each run is only executed once as executions are long.

We have only used up to 16 cores to avoid the effects of non-uniform memory accesses. We observed further increments result in a performance loss on our test machine as the computations are often memory bound. Experiments for 32 and 64 threads are not included in here for brevity but they can be seen in our code repository [4].

4.5.2. Dataset

We used a dataset similar to the one used by Kovács [42]. Most of the graphs are generated with programs from the first DIMACS implementation challenge [75]. For each graph families, we only present a single big instance instead of multiple increasing sizes for brevity. For graphs generated from real world data, `road` and `vision`, we used the biggest instances available in the LEMON website [96]. For other synthetic graphs, we tried to generate the biggest instances possible with reasonable size and performance requirements. We used the same random seed used to generate the first instance of each graph family in the LEMON study (i.e. graph names with `a` suffixes in the website). For `netgen` generator, `netgen_8` is a sparse graph with a degree 8, `netgen_sr` is a denser graph with a degree $\sqrt{n}$, and `netgen_lo` are variants with low supplies. In the original LEMON study, `netgen_deg` is a family with a fixed number of nodes and increasing degree up to n. In our study, we only used a single big instance, so `netgen_deg` corresponds to the densest variant with a degree of n. For `gridgen` generator, suffixes have similar meaning as `netgen`. For `grid` graphs, we used

the biggest instances available in LEMON website. We excluded `goto` instances from our dataset. We observed that these instances rely on the initial pivoting heuristic as mentioned by Király and Kovács [90] to have competitive performance. These instances are dramatically slower in our implementation at the moment. However, we see negligible difference of this heuristic in other instance types. We simply skipped this heuristic for this study and excluded this instance type from experiments. Note, there are no restrictions for adding this heuristic to our implementation in the future.

Dimensions of instances we used in our experiments are given in Table 4.1. Our biggest instances have up to a billion arcs. To our knowledge, this is the first study to experiment with graphs of this size for the minimum cost flow problem.

4.5.3. Distributions

Figure 4.7 shows the distributions of execution times for each step of the network simplex algorithm for each instance in our dataset. These results are from a sequential execution with a block size factor of 1. Find join node and find leaving arc operations follow a similar procedure so they have similar execution times and these are never the dominant step within an iteration. Update tree operation is well optimized and often takes a minority of execution time except for one instance. Update potential operation is a significant step and takes the most amount of time in 6 of 16 instances. Finding entering arc being the only step we parallelized in our algorithm takes the majority of the time in the rest of the instances. Note, these distributions are meant to be changed with increasing number of threads and block sizes.

Table 4.1. Dimensions of instances used in the experiments.

Instance	n	m
gridgen_8_20	1,048,577	8,388,616
gridgen_deg_15	32,762	1,073,545,216
gridgen_sr_18	262,145	134,218,240
grid_long_20	1,048,578	2,031,632
grid_square_20	1,048,578	2,097,152
grid_wide_20	1,048,578	2,162,672
netgen_8_20	1,048,576	8,388,608
netgen_deg_15	32,768	1,061,879,982
netgen_lo_8_20	1,048,576	8,388,608
netgen_lo_sr_18	262,144	134,217,728
netgen_sr_18	262,144	134,217,728
road_flow_07	2,073,870	5,156,088
road_paths_07	2,073,870	5,156,088
vision_inv_05	3,899,394	23,091,149
vision_prop_05	3,899,394	23,091,149
vision_rnd_05	3,899,394	23,091,149

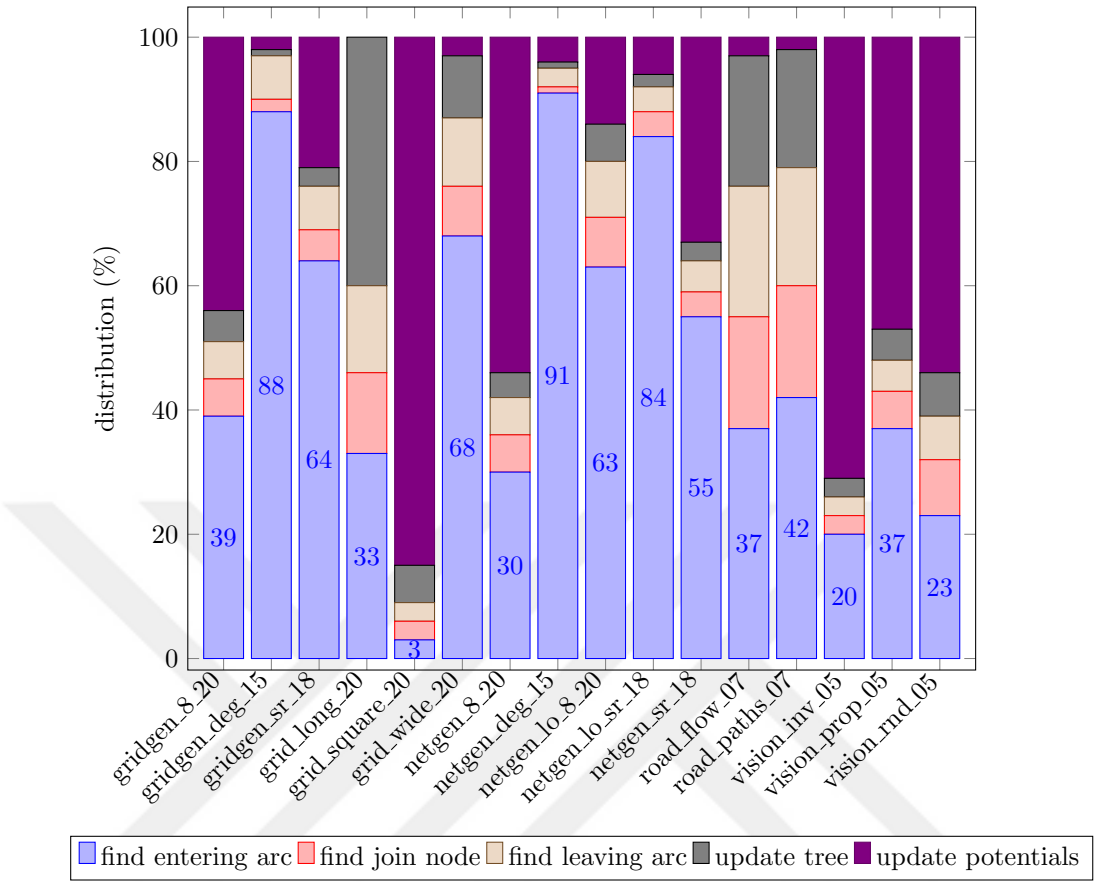


Figure 4.7. Distributions of execution times for each step of the network simplex algorithm for each instance in our dataset ($p = 1$ and $k = 1$). Finding entering arc is the only parallel step in this chapter shown with percentage numbers in the figure.

4.5.4. Iterations

Figure 4.8 shows relative numbers of iterations as percentages with increasing block size factors. In `lemon-ns` implementation, there is also a block size factor but its value is set to 1 by default which is often optimum for a sequential implementation. We see dramatic decreases in `gridgen_deg` and `netgen_deg` instances which are the biggest and the densest instances in our dataset. Three other instances, `gridgen_sr`, `netgen_lo_sr` and `netgen_sr` instances which are also big and dense graphs show a little more than half a decrease. The one exception is `grid_square` which is a small and sparse instance but shows a good amount of decrease. These instances are appropriate for parallelization by increasing block sizes. Increasing block sizes in other instances are more likely to waste computation.

4.5.5. Timings

Table 4.2 shows timings of all instances in our dataset. For most sparse instances, `lemon-cos` and `lemon-cas` performs better than others. In 2 `road` instances, `lemon-cas` is the best. In 3 `vision` instances, `lemon-cos` performs better than others. For `grid_long`, `lemon-cas` is the best by a long margin. For `grid_square`, `lemon-cos` is better than other algorithms. For `grid_wide`, network simplex algorithm performs better than others but our implementation is not able to catch `lemon-ns`. All `grid` instances are rather small compared to other instances in terms of the number of arcs. In the 2 biggest instances, `gridgen_deg` and `netgen_deg`, and the other 3 big instances, `gridgen_sr`, `netgen_sr` and `netgen_lo_sr`, increasing block sizes and number of threads seem to be beneficial as expected. These examples are also denser compared to other instances. For all instances with more than 25 million arcs, our method shows some positive improvements.

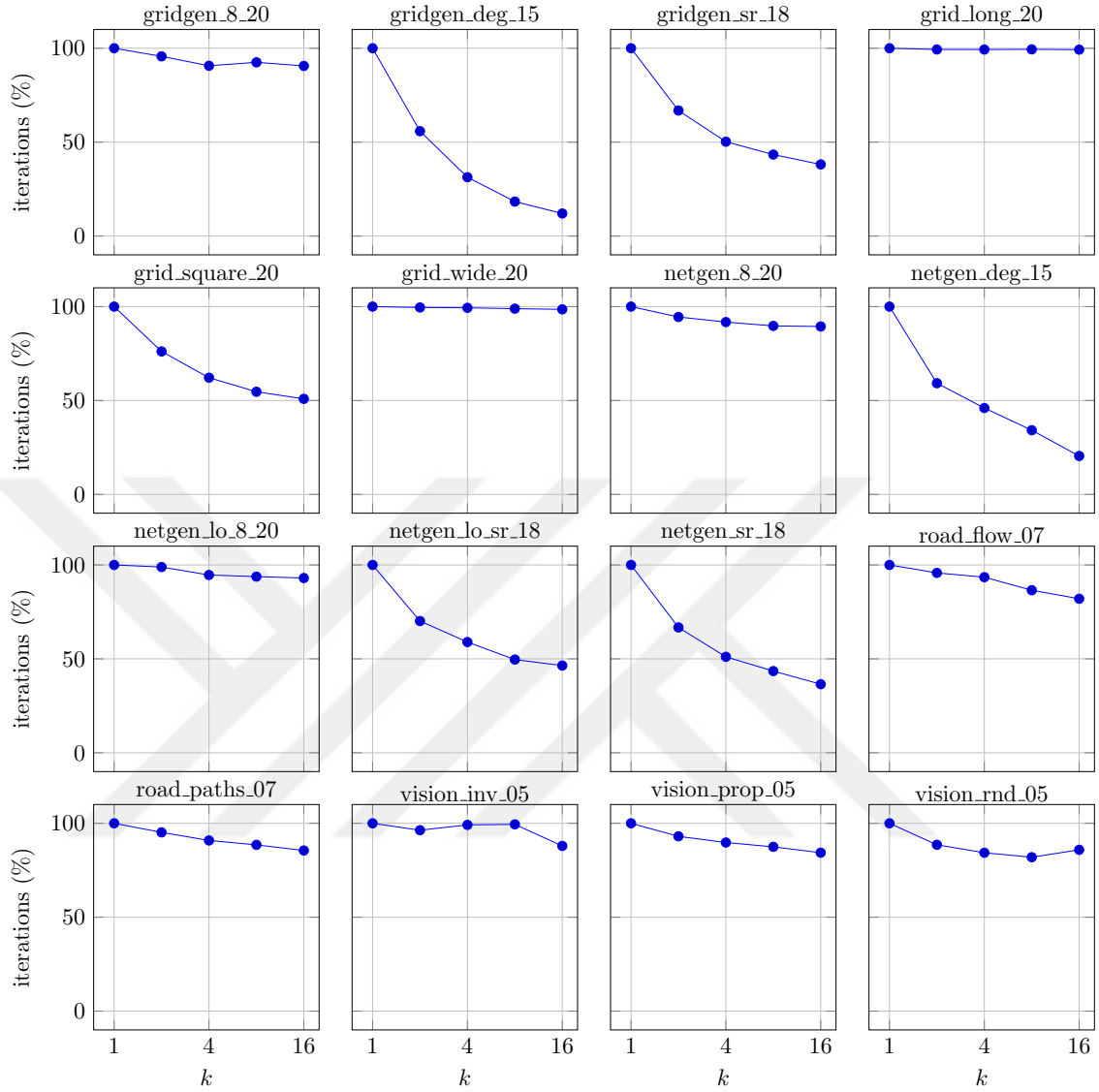


Figure 4.8. Relative numbers of iterations as percentages with increasing block size factors. The numbers are taken from our sequential implementation and they are normalized according to the first case (i.e. $k = 1$).

Table 4.2. Timings of all instances in our dataset (seconds). The left columns shows LEMON implementations for reference, and other columns show our `pns` solver with different block size factors (k) and numbers of threads (p) (dash (-) denotes timeout).

Instance	`lemon-ns`	`lemon-cos`	`lemon-cas`	$k1p1$	$k1p4$	$k1p16$	$k4p1$	$k4p4$	$k4p16$	$k16p1$	$k16p4$	$k16p16$
gridgen_8_20	118	28	3267	127	137	186	230	154	173	737	276	202
gridgen_deg_15	146	1748	-	113	94	95	107	67	43	143	68	37
gridgen_sr_18	214	233	-	237	197	174	310	127	90	837	244	98
grid_long_20	459	11	1	561	1082	2637	551	564	985	548	455	562
grid_square_20	761	107	581	1733	1525	1924	804	796	937	685	594	678
grid_wide_20	8	262	11382	12	25	70	27	27	71	92	38	70
netgen_8_20	186	30	-	335	212	246	341	261	252	951	386	286
netgen_deg_15	142	4128	-	113	77	89	104	59	53	158	73	47
netgen_lo_8_20	51	21	143	89	51	73	154	71	74	537	146	91
netgen_lo_sr_18	94	213	251	110	54	49	164	61	38	474	138	53
netgen_sr_18	352	235	-	416	340	324	406	199	137	994	324	139
road_flow_07	146	114	28	304	288	608	327	262	418	502	271	312
road_paths_07	125	74	6	226	244	562	273	220	358	454	224	268
vision_inv_05	2805	578	3394	1774	1184	1798	2807	2579	2600	3931	2564	2512
vision_prop_05	2638	623	10464	1201	941	1586	2167	1915	2076	4549	3151	3091
vision_rnd_05	5457	366	8809	2031	1428	1861	2401	2014	2197	4578	2985	2945

4.5.6. Speedups

Figure 4.9 shows speedups of our implementation with increasing number of threads calculated according to `lemon-ns` implementation. On multiple occasions, increased block sizes start slow with fewer threads but pass others with increasing number of threads in scaling instances. This is in accordance with our rationale for increasing block sizes for better parallelization. In `vision` instances, 16 threads are too many and the optimum number of threads is between 1 and 16. For these instances, block size increments help with the scaling but not enough to be beneficial. In one of our biggest instances `netgen_deg`, we see a flat curve going from 8 to 16 threads suggesting that there might be room for improvement with further block size increments.

4.5.7. Vectorization

Figure 4.10 shows speedups of our vectorized implementation with increasing number of threads calculated according to our non-vectorized implementation. These are shown with increased block sizes to emphasize the differences. For most instances in which finding entering arc takes a majority of execution time, we see an improvement with vectorization. The general trend seems to be that the difference starts noticeable with fewer threads but disappears with increasing number of threads. This trend suggests that vectorization competes with threads to claim enough number of arcs in a block for parallelization. In this case, vectorization can still be useful to have better utilization of cores with fewer threads. It might also be possible to further improve speedups with increasing block sizes for instances in which block size increments keep decreasing the number of iterations.

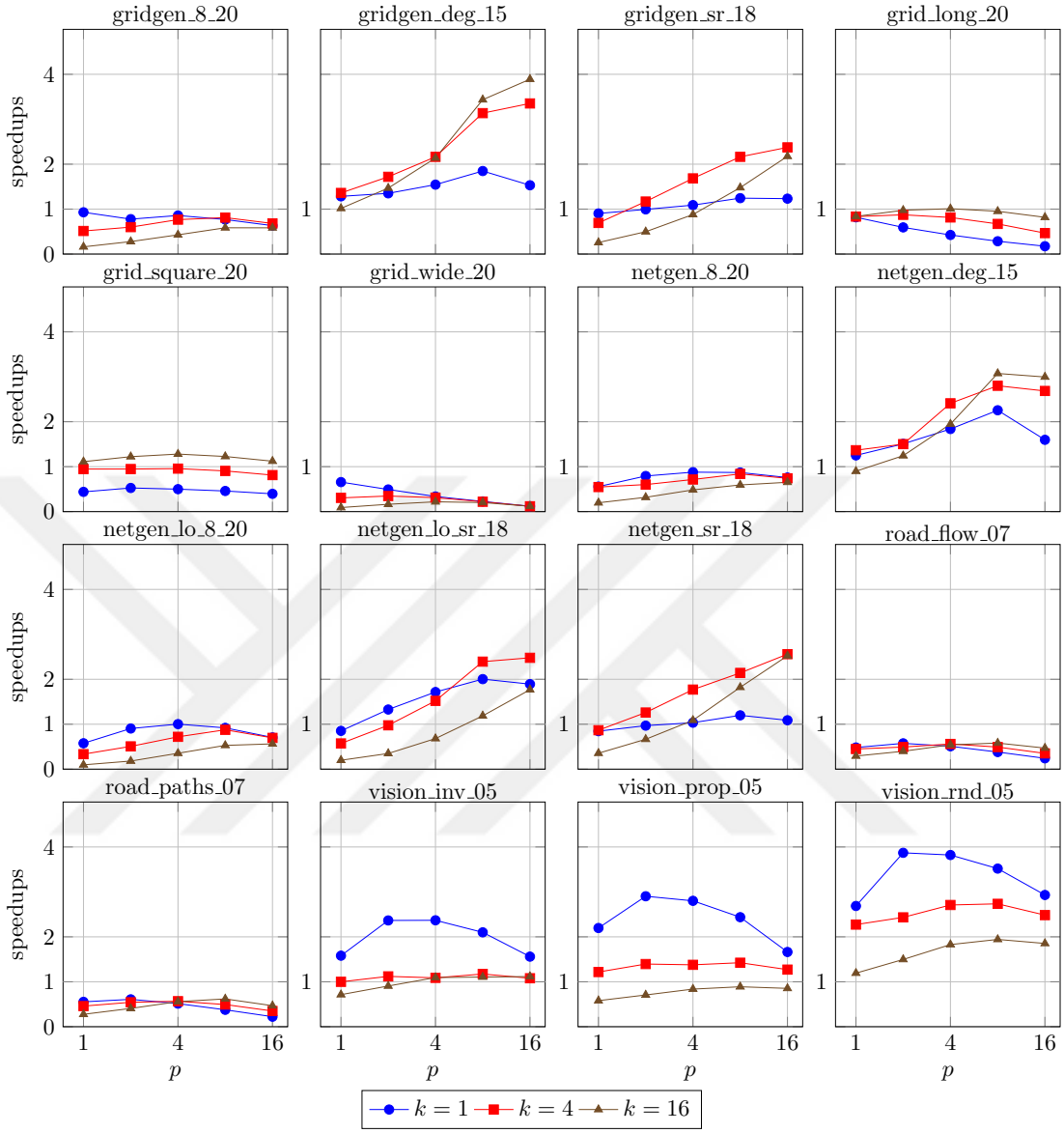


Figure 4.9. Speedups of our implementation with increasing number of threads calculated according to `lemon-ns` implementation. Our algorithm uses both OpenMP and AVX and `lemon-ns` is used with default parameters (i.e. $k = 1$).

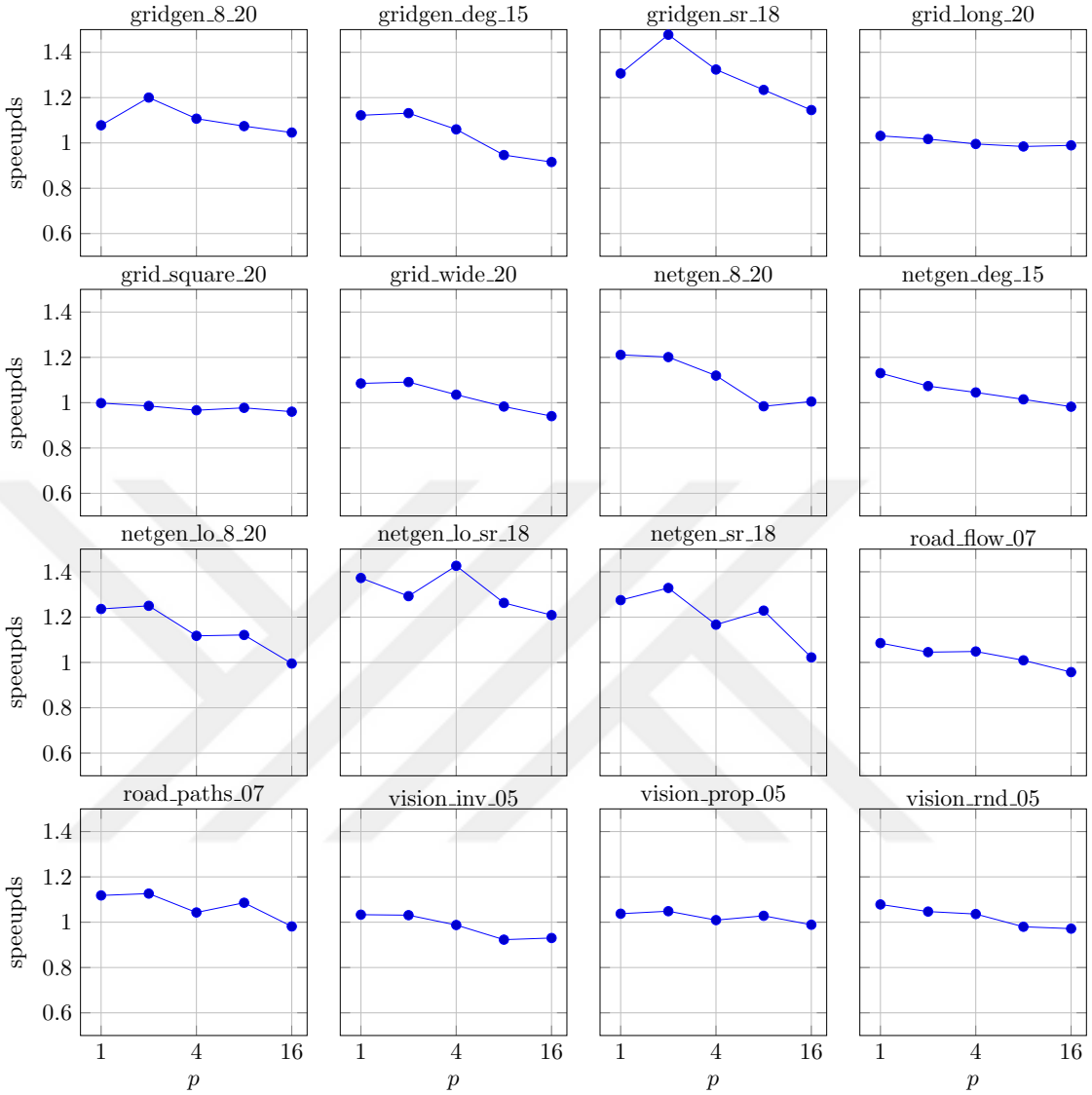


Figure 4.10. Speedups of our vectorized implementation with increasing number of threads calculated according to our non-vectorized implementation ($k = 16$).

4.6. Conclusions

We proposed a simple approach for the parallelization of the network simplex algorithm and demonstrated the benefits with an experimental study. Our results show this approach can be useful for instances in which we can decrease the number of iterations with the additional parallel computation power. OpenMP threads can scale as long as there are enough arcs in the scans to be parallelized. Vectorization seems to be useful to have better utilization of cores using fewer threads. Results suggest parameter space optimizations can have a significant effect on execution times especially when parallelization is involved.

We included big graphs having up to a billion arcs in our dataset to better demonstrate the scalability of our approach. For synthetic graph instances, we tried to generate instances as big as possible. For natural graph instances, we used the biggest instances available which already have long execution times. Our results show good scalability with bigger graphs in which there is enough room for parallelization. For other instances, we also expect improvements once bigger instances become solvable in a reasonable time on better hardware in the future.

Our approach is unique in that it shows improvements with block searching which is among the best performing pivoting rules available. We experimented with best eligible pivoting rule but failed to get any speedups over block searching even though there is a greater possibility for parallelization. This suggests that increasing block sizes for parallelization has an upper limit to be useful, though this limit may be beyond what is presented in this chapter.

We also tried various GPU approaches for parallelization but failed to get any improvements over the CPU version. On GPUs, arc scans can easily be processed in parallel but data dependencies can slow things down. Other operations involve tree walking so they are harder to parallelize as discussed in this chapter. Specifically, we tried to use the Euler tour technique introduced by Tarjan and Vishkin [97] to linearize

the spanning tree and then used parallel list traversal for tree update operations. However, existing improvements for tree operations are not directly adaptable with this approach. On top of these difficulties, iterations in the network simplex algorithm are typically in the sub-millisecond range which makes data transfers a big challenge. Therefore, efficient GPU implementations may instead require investigation of running multiple iterations in parallel.

As for future work, performance differences of our implementation with others can be further investigated. The network simplex implementation has been studied extensively for a long time and there are many implementation details to improve performance. Initial pivoting heuristic used in `lemon-ns` can be added to our implementation as well. Also, we expect candidate list pivoting rules to be similarly appropriate for parallelization. Such pivoting rules can be as efficient as block pivoting rule in the sequential case so any speedup gains would be beneficial. Lastly, different variants of the network simplex can be of interest in regards to parallelization. These variants include dual and scaling versions of the algorithm, and other specialized variants for the maximum flow and the shortest path problems.

5. CONCLUSIONS

We proposed two parallel algorithms for the maximum flow and the minimum cost flow problems. Experiments show that parallel implementations can be useful depending on the characteristics of the input graph. However, there are also cases where the use of our parallel algorithms is not beneficial. Therefore, the decision of making use of parallel algorithms for the problem at hand is left to the user. We have tried to include as many different input graphs as possible to our experiments to help this decision. Our general consensus is that our parallel algorithms can be useful for wide and sparse graphs for the maximum flow problem and dense graphs for the minimum cost flow problem. In both problems, input graph needs to be sufficiently big to benefit from parallelization.

We have used shared memory parallelism in both of our algorithms. Our biggest graph inputs have more than a billion arcs for both problems. Our test machines had sufficient memory to hold these graphs in memory. Bigger graphs may require distributed memory parallel algorithms to share the data over the memories of multiple machines. However, distributed memory algorithms can face a challenge to avoid the communication bottleneck. Our shared memory algorithms are a good compromise between performance and sufficiency. Nevertheless, distributed memory algorithms for network flow problems can still be an interesting topic as a future work.

We have also considered using GPUs for parallel processing during our research. However, our limited efforts have not yielded satisfactory results. The most important challenge with GPU algorithms were to avoid the data transfer bottlenecks between the main and GPU memories. This becomes an issue especially for iterative algorithms if the sequential portion of the algorithm has a significant amount of computation. In such cases, neither keeping the data in GPU memory nor transferring it between main and GPU memories provides a sufficient performance improvement. All aside, we still think GPU algorithms can be useful for the network flow research in the future.

At the same time, GPUs have been influencial for the development of new CPU features such as vector extensions found in modern CPUs. These extensions provide a limited form of parallelism similar to GPUs, though the performance difference is quite high. On the flip side, data transfer bottlenecks are not an issue with these extensions. Our use of vector extensions provided some improvements but not as much as we expected. The most important reason is that our computations are generally memory bound rather than compute bound. Therefore a good strategy involving vectorization should focus on improving memory access performance.

On the other hand, cache memory hierarchies have been getting deeper over time generally in favor of regular access patterns for good performance. Graph algorithms are often known to perform irregular data accesses due to having sparse matrix semantics for data representations. Our results also shows a similar trend for network flow algorithms. Previous research shows that data structures are an important factor to achieve good performance for most network flow algorithms. The design of better data structures for parallel network flow problems to utilize caches of multiple cores can provide better or even super-linear speedups in the future.

Network flow algorithms often require much effort to implement in a correct manner. Parallelization of these algorithms adds an extra layer of complexity on top of this. Working on the design and implementation of our algorithms and experimenting with other existing implementations showed us that it is easy to make an error which is often non-deterministic in nature. For this reason, we tried to formally prove the correctness of our push-relabel algorithm. For the network simplex algorithm, we only used a parallel reduction operation but it still added some benign non-determinism to the implementation as discussed before. Our consensus is that formal proofs are essential in parallel network flow algorithms for any non-trivial changes to the original algorithm.

REFERENCES

1. Ahuja, R., T. Magnanti and J. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice Hall, Feb. 1993.

2. Dagum, L. and R. Menon, "OpenMP: An Industry Standard API for Shared-Memory Programming", *IEEE Computational Science and Engineering*, Vol. 5, No. 1, pp. 46–55, Jan. 1998.

3. Kara, G. and C. Özturan, *Algorithm 1002: Graph Coloring Based Parallel Push-Relabel Algorithm for the Maximum Flow Problem*, ACM Collected Algorithms (CALGO), Dec. 2019, `https://calgo.acm.org/1002.zip`, accessed in October 2021.

4. Kara, G. and C. Özturan, *PNS: Parallel Network Simplex Algorithm for the Minimum Cost Flow Problem*, Zenodo, Sep. 2021, `https://zenodo.org/record/5502052`, accessed in October 2021.

5. Kara, G. and C. Özturan, "Algorithm 1002: Graph Coloring Based Parallel Push-relabel Algorithm for the Maximum Flow Problem", *ACM Transactions on Mathematical Software (TOMS)*, Vol. 45, No. 4, pp. 1–28, 2019.

6. Kara, G. and C. Özturan, "Parallel Network Simplex Algorithm for the Minimum Cost Flow Problem", *Concurrency and Computation: Practice and Experience*, Vol. 34, No. 4, p. e6659, 2022.

7. Goldberg, A. V., "Recent Developments in Maximum Flow Algorithms", *Scandinavian Workshop on Algorithm Theory*, pp. 1–10, Springer, 1998.

8. Ford Jr, L. R., D. R. Fulkerson and A. Ziffer, *Flows in Networks*, AIP, 1963.

9. Edmonds, J. and R. M. Karp, "Theoretical Improvements in Algorithmic Efficiency

for Network Flow Problems", *Journal of the ACM*, Vol. 19, No. 2, pp. 248–264, Apr. 1972.

10. Dinitz, Y., "Dinitz' Algorithm: The Original Version and Even's Version", *Theoretical Computer Science*, pp. 218–240, Springer, Berlin, Heidelberg, 2006.

11. Ahuja, R. K. and J. B. Orlin, "Distance-Directed Augmenting Path Algorithms for Maximum Flow and Parametric Maximum Flow Problems", *Naval Research Logistics*, pp. 413–430, 1991.

12. Boykov, Y. and V. Kolmogorov, "An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 26, No. 9, pp. 1124–1137, Sep. 2004.

13. Goldberg, A. V., S. Hed, H. Kaplan, R. E. Tarjan and R. F. Werneck, "Maximum Flows by Incremental Breadth-First Search", *Algorithms – ESA 2011*, pp. 457–468, Springer, Berlin, Heidelberg, Sep. 2011.

14. Goldberg, A. V., *Efficient Graph Algorithms for Sequential and Parallel Computers*, Ph.D. Thesis, Massachusetts Institute of Technology, 1987.

15. Goldberg, A. V. and R. E. Tarjan, "A New Approach to the Maximum-Flow Problem", *Journal of the ACM*, Vol. 35, No. 4, pp. 921–940, Oct. 1988.

16. Cherkassky, B. V. and A. V. Goldberg, "On Implementing the Push—Relabel Method for the Maximum Flow Problem", *Algorithmica*, Vol. 19, No. 4, pp. 390–410, Dec. 1997.

17. Cerulli, R., M. Gentili and A. Iossa, "Efficient Preflow Push Algorithms", *Computers & Operations Research*, Vol. 35, No. 8, pp. 2694–2708, Aug. 2008.

18. Goldberg, A. V., "The Partial Augment–Relabel Algorithm for the Maximum Flow Problem", *Algorithms - ESA 2008*, pp. 466–477, Springer, Berlin, Heidelberg, Sep.

2008.

19. Goldberg, A. V., "Two-Level Push-Relabel Algorithm for the Maximum Flow Problem", *Algorithmic Aspects in Information and Management*, pp. 212–225, Springer, Berlin, Heidelberg, Jun. 2009.

20. Hochbaum, D. S., "The Pseudoflow Algorithm and the Pseudoflow-Based Simplex for the Maximum Flow Problem", *Integer Programming and Combinatorial Optimization*, pp. 325–337, Springer, Berlin, Heidelberg, Jun. 1998.

21. Hochbaum, D. S., "The Pseudoflow Algorithm: A New Algorithm for the Maximum-Flow Problem", *Operations Research*, Vol. 56, No. 4, pp. 992–1009, Aug. 2008.

22. Hochbaum, D. S. and J. B. Orlin, "Simplifications and Speedups of the Pseudoflow Algorithm", *Networks*, Vol. 61, No. 1, pp. 40–57, Jan. 2013.

23. Chandran, B. G. and D. S. Hochbaum, "A Computational Study of the Pseudoflow and Push-Relabel Algorithms for the Maximum Flow Problem", *Operations Research*, Vol. 57, No. 2, pp. 358–376, Jan. 2009.

24. Fishbain, B., D. S. Hochbaum and S. Mueller, "A Competitive Study of the Pseudoflow Algorithm for the Minimum S–T Cut Problem in Vision Applications", *Journal of Real-Time Image Processing*, Vol. 11, No. 3, pp. 589–609, Mar. 2016.

25. Anderson, R. J. and J. C. Setubal, "On the Parallel Implementation of Goldberg's Maximum Flow Algorithm", *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, pp. 168–177, ACM, New York, NY, USA, 1992.

26. Anderson, R. and J. C. Setubal, "A Parallel Implementation of the Push-Relabel Algorithm for the Maximum Flow Problem", *Journal of Parallel and Distributed Computing*, Vol. 29, No. 1, pp. 17–26, Aug. 1995.

27. Bader, D. A. and V. Sachdeva, *A Cache-Aware Parallel Implementation of the Push-Relabel Network Flow Algorithm and Experimental Evaluation of the Gap Relabeling Heuristic*, Technical Report, Georgia Institute of Technology, Feb. 2006.

28. Hong, B., "A Lock-Free Multi-Threaded Algorithm for the Maximum Flow Problem", *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, 2008.

29. Hong, B. and Z. He, "An Asynchronous Multithreaded Algorithm for the Maximum Network Flow Problem With Nonblocking Global Relabeling Heuristic", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 22, No. 6, pp. 1025–1033, Jun. 2011.

30. He, Z. and B. Hong, "Dynamically Tuned Push-Relabel Algorithm for the Maximum Flow Problem on CPU-GPU-Hybrid Platforms", *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pp. 1–10, Apr. 2010.

31. Baumstark, N., G. Blelloch and J. Shun, "Efficient Implementation of a Synchronous Parallel Push-Relabel Algorithm", *Algorithms - ESA 2015*, pp. 106–117, Springer, Berlin, Heidelberg, 2015.

32. Halim, F., R. H. C. Yap and Y. Wu, "A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs", *2011 31st International Conference on Distributed Computing Systems*, pp. 192–202, Jun. 2011.

33. Caragea, G. C. and U. Vishkin, "Brief Announcement: Better Speedups for Parallel Max-Flow", *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pp. 131–134, ACM, New York, NY, USA, 2011.

34. Liu, J. and J. Sun, "Parallel Graph-Cuts by Adaptive Bottom-Up Merging", *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*,

pp. 2181–2188, Jun. 2010.

35. Delong, A. and Y. Boykov, "A Scalable Graph-Cut Algorithm for N-D Grids", *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1–8, Jun. 2008.

36. Strandmark, P. and F. Kahl, "Parallel and Distributed Graph Cuts by Dual Decomposition", *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 2085–2092, Jun. 2010.

37. Shekhovtsov, A. and V. Hlaváč, "A Distributed Mincut/Maxflow Algorithm Combining Path Augmentation and Push-Relabel", *International Journal of Computer Vision*, Vol. 104, No. 3, pp. 315–342, Sep. 2013.

38. Dixit, N., R. Keriven and N. Paragios, *GPU-Cuts: Combinatorial Optimisation, Graphic Processing Units and Adaptive Object Extraction*, Tech. rep., École des Ponts ParisTech, 2005.

39. Hussein, M., A. Varshney and L. Davis, "On Implementing Graph Cuts on CUDA", *First Workshop on General Purpose Processing on Graphics Processing Units*, Vol. 2007, 2007.

40. Vineet, V. and P. J. Narayanan, "CUDA Cuts: Fast Graph Cuts on the GPU", *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pp. 1–8, Jun. 2008.

41. Sifaleras, A., "Minimum Cost Network Flows: Problems, Algorithms, and Software", *Yugoslav Journal of Operations Research*, Vol. 23, No. 1, pp. 3–17, 2013.

42. Kovács, P., "Minimum-Cost Flow Algorithms: An Experimental Evaluation", *Optimization Methods and Software*, Vol. 30, No. 1, pp. 94–127, Jan. 2015.

43. Dezső, B., A. Jüttner and P. Kovács, "LEMON – An Open Source C++ Graph

Template Library", *Electronic Notes in Theoretical Computer Science*, Vol. 264, No. 5, pp. 23–45, Jul. 2011.

44. Vieira, C. L. D. S., M. M. M. Luna and J. M. Azevedo, "Minimum-Cost Flow Algorithms: A Performance Evaluation Using the Brazilian Road Network", *World Review of Intermodal Transportation Research*, Vol. 8, No. 1, pp. 3–21, Jan. 2019.

45. Dong, Y., Y. Gao, R. Peng, I. Razenshteyn and S. Sawlani, "A Study of Performance of Optimal Transport", *arXiv: 2005.01182*, May 2020.

46. Xie, F. and R. Jia, "Nonlinear Fixed Charge Transportation Problem by Minimum Cost Flow-Based Genetic Algorithm", *Computers & Industrial Engineering*, Vol. 63, No. 4, pp. 763–778, Dec. 2012.

47. Rostami, R. and A. Ebrahimnejad, "An Approximation Algorithm for Discrete Minimum Cost Flows Over Time Problem", *International Journal of Operational Research*, Vol. 20, No. 2, pp. 226–239, Jan. 2014.

48. Ghasemishabankareh, B., M. Ozlen, X. Li and K. Deb, "A Genetic Algorithm With Local Search for Solving Single-Source Single-Sink Nonlinear Non-Convex Minimum Cost Flow Problems", *Soft Computing*, Vol. 24, No. 2, pp. 1153–1169, Jan. 2020.

49. Ghasemishabankareh, B., *Meta-Heuristics for Better Constraint Handling and Minimum Cost Flow Problems*, PhD Thesis, RMIT University, Australia, 2020.

50. Sedgewick, R., *Algorithms in Java, Part 5: Graph Algorithms*, Chap. 22. Network Flow, Addison-Wesley Professional, Jul. 2003.

51. Ahuja, R., T. Magnanti and J. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Chap. 9. Minimum Cost Flows: Basic Algorithms, Prentice Hall, Feb. 1993.

52. Nachtigall, K. and J. Opitz, "Solving Periodic Timetable Optimisation Problems by Modulo Simplex Calculations", *8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2008.

53. Goerigk, M. and A. Schöbel, "Improving the Modulo Simplex Algorithm for Large-Scale Periodic Timetabling", *Computers & Operations Research*, Vol. 40, No. 5, pp. 1363–1370, May 2013.

54. Goerigk, M. and C. Liebchen, "An Improved Algorithm for the Periodic Timetabling Problem", *17th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2017)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017.

55. Borndörfer, R., H. Hoppmann, M. Karbstein and F. Löbel, "The Modulo Network Simplex With Integrated Passenger Routing", *Operations Research Proceedings 2016*, Springer International Publishing, Hamburg, Germany, 2018.

56. Löbel, F., N. Lindner and R. Borndörfer, "The Restricted Modulo Network Simplex Method for Integrated Periodic Timetabling and Passenger Routing", *Operations Research Proceedings 2019*, Springer International Publishing, Dresden, Germany, 2020.

57. Holzhauser, M., S. O. Krumke and C. Thielen, "A Network Simplex Method for the Budget-Constrained Minimum Cost Flow Problem", *European Journal of Operational Research*, Vol. 259, No. 3, pp. 864–872, Jun. 2017.

58. Ryan, C. T., R. L. Smith and M. A. Epelman, "A Simplex Method for Uncapacitated Pure-supply Infinite Network Flow Problems", *SIAM Journal on Optimization*, Vol. 28, No. 3, pp. 2022–2048, Jan. 2018.

59. Beckenbach, I., "A Hypergraph Network Simplex Algorithm", *Operations Research*

Proceedings 2017, Springer International Publishing, Berlin, Germany, 2018.

60. Zheng, Q., J. Li, H. Tian, Z. Wang and S. Wang, "A 2-Layers Virtual Network Mapping Algorithm Based on Node Attribute and Network Simplex", *IEEE Access*, Vol. 6, pp. 77474–77484, 2018.

61. Nie, Z. and S. Wang, "A Sequential Simplex Algorithm for the Continuous Convex Piecewise Linear Network Flow Problem", *2019 IEEE 15th International Conference on Control and Automation (ICCA)*, pp. 1307–1313, Jul. 2019.

62. Lin, W., Z. He and M. Xiao, "Balanced Clustering: A Uniform Model and Fast Algorithm", *International Joint Conference on Artificial Intelligence*, pp. 2987–2993, 2019.

63. Disser, Y. and M. Skutella, "The Simplex Algorithm Is NP-Mighty", *ACM Transactions on Algorithms*, Vol. 15, No. 1, pp. 5:1–5:19, Nov. 2018.

64. Peters, J., "The Network Simplex Method on a Multiprocessor", *Networks*, Vol. 20, No. 7, pp. 845–859, 1990.

65. Miller, D. L., J. F. Pekny and G. L. Thompson, "Solution of Large Dense Transportation Problems Using a Parallel Primal Algorithm", *Operations Research Letters*, Vol. 9, No. 5, pp. 319–324, Sep. 1990.

66. Thulasiraman, K., R. P. Chalasani and M. A. Comeau, "Parallel Network Dual Simplex Method on a Shared Memory Multiprocessor", *Proceedings of 1993 5th IEEE Symposium on Parallel and Distributed Processing*, pp. 408–415, Dec. 1993.

67. Barr, R. S. and B. L. Hickman, "Parallel Simplex for Large Pure Network Problems: Computational Testing and Sources of Speedup", *Operations Research*, Vol. 42, No. 1, pp. 65–80, 1994.

68. Jiang, J., J. Chen and C. Wang, "Multi-Granularity Hybrid Parallel Network Sim-

plex Algorithm for Minimum-Cost Flow Problems", *The Journal of Supercomputing*, Vol. 76, No. 12, pp. 9800–9826, Dec. 2020.

69. Das, S., I. Finocchi and R. Petreschi, "Star-Coloring of Graphs for Conflict-Free Access to Parallel Memory Systems", *18th IEEE International Parallel and Distributed Processing Symposium, 2004*, p. 50, 2004.

70. Felzenszwalb, P. F. and D. P. Huttenlocher, "Efficient Belief Propagation for Early Vision", *International Journal of Computer Vision*, Vol. 70, No. 1, pp. 41–54, Oct. 2006.

71. Mahjourian, R., F. Chen, R. Tiwari, M. Thai, H. Zhai and Y. Fang, "An Approximation Algorithm for Conflict-Aware Broadcast Scheduling in Wireless Ad Hoc Networks", *Proceedings of the 9th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '08, pp. 331–340, ACM, New York, NY, USA, 2008.

72. Awais Hussein, S., P. Coussy, C. Chavet and E. Martin, "An Approach Based on Edge Coloring of Tripartite Graph for Designing Parallel LDPC Interleaver Architecture", *IEEE International Symposium on Circuits and Systems (ISCAS) 2011*, pp. 1720–1723, Rio de Janeiro, Brazil, May 2011.

73. Baumstark, N., *Speeding Up Maximum Flow Computations on Shared Memory Platforms*, Bachelor Thesis, Karlsruher Institut für Technologie (KIT), 2014.

74. Computer Vision Research Group, *Max-Flow Problem Instances in Vision*, University of Western Ontario, Oct. 2009, `https://vision.csd.uwo.ca/data/maxflow`, accessed in October 2021.

75. Johnson, D. S., C. C. McGeoch and others, *Network Flows and Matching: First DIMACS Implementation Challenge*, Vol. 12, American Mathematical Society, 1993.

76. Sanders, P. and C. Schulz, "Think Locally, Act Globally: Highly Balanced Graph

Partitioning", *International Symposium on Experimental Algorithms*, pp. 164–175, Springer, Berlin, Heidelberg, Jun. 2013.

77. Schulz, C., *KaHIP - Karlsruhe High Quality Partitioning*, GitHub, May 2013, `https://kahip.github.io/`, accessed in October 2021.

78. Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pp. 483–485, ACM, New York, NY, USA, Apr. 1967.

79. Gustafson, J. L., "Reevaluating Amdahl's Law", *Communications of the ACM*, Vol. 31, No. 5, pp. 532–533, May 1988.

80. Dantzig, G. B., "Maximization of a Linear Function of Variables Subject to Linear Inequalities", *Activity Analysis of Production and Allocation*, Vol. 13, pp. 339–347, 1951.

81. Dantzig, G. B., "Application of the Simplex Method to a Transportation Problem", *Activity Analysis and Production and Allocation*, 1951.

82. Dantzig, G. B., *Linear Programming and Extensions*, Princeton Univ. Press, Princeton, NJ, 1963.

83. Cunningham, W. H., "A Network Simplex Method", *Mathematical Programming*, Vol. 11, No. 1, pp. 105–116, Dec. 1976.

84. Barr, R. S., F. Glover and D. Klingman, "The Alternating Basis Algorithm for Assignment Problems", *Mathematical Programming*, Vol. 13, No. 1, pp. 1–13, Dec. 1977.

85. Dennis, J. B., "A High-Speed Computer Technique for the Transportation Problem", *Journal of the ACM (JACM)*, Vol. 5, No. 2, pp. 132–153, 1958.

86. Grigoriadis, M. D., "An Efficient Implementation of the Network Simplex Method", *Netflow at Pisa*, Mathematical Programming Studies, pp. 83–111, Springer, Berlin, Heidelberg, 1986.

87. Srinivasan, V. and G. L. Thompson, "Benefit-Cost Analysis of Coding Techniques for the Primal Transportation Algorithm", *Journal of the ACM*, Vol. 20, No. 2, pp. 194–213, Apr. 1973.

88. Bradley, G. H., G. G. Brown and G. W. Graves, "Design and Implementation of Large Scale Primal Transshipment Algorithms", *Management Science*, Vol. 24, No. 1, pp. 1–34, 1977.

89. Mulvey, J. M., "Pivot Strategies for Primal-Simplex Network Codes", *Journal of the ACM (JACM)*, Vol. 25, No. 2, pp. 266–270, Apr. 1978.

90. Király, Z. and P. Kovács, "Efficient Implementations of Minimum-Cost Flow Algorithms", *arXiv: 1207.6381*, Jul. 2012.

91. Kelly, D. J. and G. M. ONeill, *The Minimum Cost Flow Problem and the Network Simplex Solution Method*, PhD Thesis, University College Dublin, 1991.

92. Barr, R., F. Glover and D. Klingman, "Enhancements of Spanning Tree Labelling Procedures for Network Optimization", *INFOR: Information Systems and Operational Research*, Vol. 17, No. 1, pp. 16–34, Feb. 1979.

93. Abel, J. and K. Balasubramanian, "Applications Tuning for Streaming SIMD Extensions", *Intel Technology Journal*, Vol. Q2, 1999.

94. Dennis, J. B., G. R. Gao and V. Sarkar, "Determinacy and Repeatability of Parallel Program Schemata", *2012 Data-Flow Execution Models for Extreme Scale Computing*, pp. 1–9, Sep. 2012.

95. Blelloch, G. E., J. T. Fineman, P. B. Gibbons and J. Shun, "Internally Determin-

istic Parallel Algorithms Can Be Fast", *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pp. 181–192, ACM, New York, NY, USA, Feb. 2012.

96. Kovács, P., *Benchmark Data for the Minimum-Cost Flow Problem*, LEMON, Jan. 2015, `https://lemon.cs.elte.hu/trac/lemon/wiki/MinCostFlowData`, accessed in October 2021.

97. Tarjan, R. and U. Vishkin, "Finding Biconnected Componemts and Computing Tree Functions in Logarithmic Parallel Time", *25th Annual Symposium on Foundations of Computer Science*, pp. 12–20, Oct. 1984.

APPENDIX A: COPYRIGHT NOTICE

All visual content in this thesis book is produced by the author for his own publications [5, 6] and then reused in this thesis book. Visual content produced in the scope of this thesis study whose copyrights are transferred to a publisher are used in accordance with the publisher's current policies found in the publisher's web site as of this writing in regards to the reuse of text and visual content produced by the authors themselves in this thesis book.