A Multithreaded Algorithm for the Maximum Flow Problem

by

Md. Mostafizur Rahman

A dissertation submitted in partial fulfillment of the
requirements for the degree of

Master of Science

Department of Computer Science
Faculty of Graduate Studies
University of Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION

A MULTITHREADED ALGORITHM FOR THE MAXIMUM FLOW PROBLEM

BY

Md. Mostafizur Rahman

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

Of

MASTER OF SCIENCE

Abstract

Finding a maximum flow through a transport network is a central problem in operations research, computer science and engineering. This research has been triggered by the diverse areas of applications of maximum flow problem as well as the algorithmic challenges it has offered to computer scientists and engineers alike. The focus of this work is on designing and developing a multithreaded algorithm for the maximum flow problem and implementing the algorithm on OpenMP.

Maximum flow problems that arise in applications are usually large, and hence computationally intensive. Solving these problems often stretches the capabilities of conventional uniprocessor architectures. Therefore, parallel computing becomes vitally important.

There are many parallel maximum flow algorithms for shared and distributed memory model on a traditional von Neumann computer architecture. These parallel algorithms have communication and synchronization latency problems inherent in this architecture. Multithreading overlaps computation with communication and thereby improves performance. Among many sequential algorithms, Goldberg and Tarjan's maximum flow algorithm has been extensively studied for parallelization. However, the only multithreaded implementation of this algorithm by Lie does not give good performance.

The irregularity and asynchronicity posed by the algorithm of Malhotra, Paramodh Kumar and Maheswari (MPM) makes it a very good candidate for multithreading. Moreover, the sequential version of the MPM algorithm has the same complexity as Goldberg and Tarjan's algorithm. We have designed and implemented a multithreaded algorithm for MPM approach. We have achieved a maximum speedup of 6.33 on 8 processors for our implementation which is better, to our knowledge, than the maximum speedup achieved in any shared memory implementation of Goldberg and Tarjan's maximum flow algorithm.

Acknowledgements

First, I would like to express my deepest gratitude to my thesis supervisors Dr. Parimala Thulasiraman and Dr. Ruppa K. Thulasiram for their guidance, encouragement, patience and inspiration. Dr. Parimala Thulasiraman introduced me to the area of research in maximum flow problem.

I would like to thank Faculty of Science for supporting me with Faculty of Science Scholarship, Faculty of Graduate Studies for supporting me with University of Manitoba Graduate Fellowship, Dr. Michel Toulouse and the department of Computer Science for supporting me with TA.

I am also thankful to the thesis committee member, Dr. Ekram Hossain, for being in my thesis committee.

Finally I express my heartfelt gratitude to my parents, my friends Mohammad Rashedur Rahman, Sajib Barua, Mohammad Mamunur Rashid and Rajesh Palit who encouraged me to come here and gave me support during my research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we introduce the maximum (max) flow problem, its applications and computing issues that arise in parallelizing the maximum flow algorithms.

1.1 Maximum Flow Problem and Its Applications

A transport network can be represented as a connected directed graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. As an example, we can represent a road map as a transport network where cities are vertices and roads between pairs of cities are edges. The transport network can be used to model the transportation of a material such as current through electrical circuits or a commodity from a production center to the market through transportation routes. Each edge (u, v) of a transport network is associated with two weights called *flow* $f(u, v)$ and *capacity* $c(u, v)$. The *capacity* $c(u, v)$ of an edge (u, v) represents the maximum rate at which a commodity can be transported along that edge and the *flow* $f(u, v)$ in an edge is the rate at which

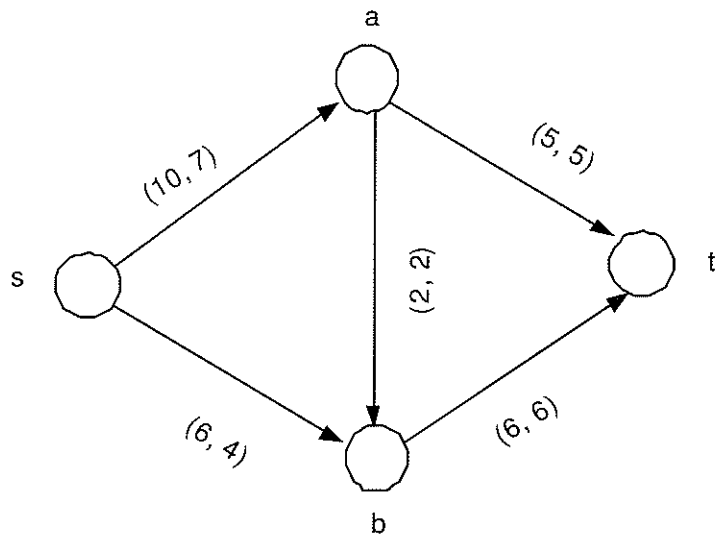

Figure 1.1: A transport network.

the commodity moves along that edge. Vertices are junctions of a transport network and for each vertex, except the source, s and the destination, t the rate at which a material enters a vertex must equal the rate at which it leaves the vertex. In other words, the incoming flow is equal to the outgoing flow for each vertex, except the source, s and the destination, t. This property is called flow conservation constraint. A flow f in a transport network $G = (V, E)$ is an assignment of a nonnegative real number $f(u, v)$ to each edge (u, v) in such a way that the following two conditions are satisfied:

Capacity Constraint: $f(u, v) \leq c(u, v)$ for every edge $(u, v) \in E$

Flow Conservation Constraint: $\sum_{\forall v} f(u, v) = \sum_{\forall v} f(v, u)$ for all $u \in E - \{s, t\}$

The source produces the material at some steady rate, and the destination consumes the material at the same rate. The value of a flow f is defined as

$| f | = \sum_{\forall u} f(s, u) = \sum_{\forall u} f(u, t)$

In Figure 1.1 we have shown a transport network with four vertices—s(source), a, b and t(sink). There are five edges in this network—edge (s, a) with capacity 10 and flow 7, edge (s, b) with capacity 6 and flow 4, edge (a, b) with capacity 2 and flow 2, edge (a, t) with capacity 5 and flow 5, edge (b, t) with capacity 6 and flow 6. For vertex a incoming flow is 7 and outgoing flow is $5 + 2 = 7$. Similarly for vertex b incoming flow is $2 + 4 = 6$ and outgoing flow is 6. For source s the outgoing flow is $7 + 4 = 11$ and for sink t the incoming flow is $5 + 6 = 11$. The flow in the transport network of Figure 1.1 is 11.

The maximum flow problem (see for example, [42]) determines the greatest rate at which the material can be pushed from the source, s, to the sink (destination), t, without violating the capacity constraints of the edges in the network. In other words, a flow f^* in a transport network is said to be maximum flow if there is no such flow f in the network such that $| f | > | f^* |$.

In [2, 42, 49] the maximum flow problem and its applications are discussed extensively. The maximum flow problem arises in a wide variety of situations and in several forms. In some cases, the maximum flow problem occurs as a subproblem in the solution of more difficult network optimization problems, such as the minimum cost flow problem [2], transhipment problem [12] or the generalized flow problem [55] applied in telecommunications [2]. The problem also arises in a number of combinatorial optimization problems [42], such as network connectivity, matchings and covers, which on the surface might not appear to involve maximum flow. Other applications that apply the

maximum flow directly are in task scheduling [37], load assignment problems [43] and tanker scheduling [13]. One of the recent applications of maximum flow problem is in identification of web communities [17, 18, 30]. A web community is a set of web pages having a common topic. The web is constructed from some seed nodes having information on the topic. Those nodes are added in the graph as successors of seed nodes which have links from the seed nodes. Again some nodes are added to the successors of seed nodes. A virtual source, s, is added to the seed nodes with capacity ∞ and a virtual sink, t, is added to the nodes that are at distance 2 from the seed nodes and the capacity of the incoming edges for the sink is set as 1. Then the maximum flow algorithm is applied to this graph. After application of the maximum flow on the graph those nodes that are reachable from source are added as seed nodes. A new graph is constructed and maximum flow algorithm is repeatedly applied up to some number of iterations. After the iterative application of maximum flow algorithm those nodes that are reachable from the source comprise the web community. Another application of maximum flow algorithm is in the improved depth estimation in stereo correspondence problem of stereo images [44, 48]. In [44, 48], the authors use maximum flow approach instead of dynamic programming approach to solve the stereo correspondence problem.

1.2 Parallel Computing Bottlenecks

Maximum flow problems that arise in applications are characterized by their large data sets and solving these problems are often beyond the capabilities of conventional unipro-

cessor architectures. Therefore, parallel computing becomes vitally important. Many parallel algorithms [1, 3, 24, 50, 54] exist in the literature to solve the maximum flow problem on shared and distributed memory architectures. However, the considerable asynchronicity introduced during the computation of the maximum flow problem produces very slight improvement in performance due to the latency problems of remote memory access and synchronization inherent in conventional parallel computers [29]. Problems that exhibit high asynchronicity, variable changes in data movements and distribution, chaotic communication pattern during runtime are characterized as *irregular* problems. The pattern of data distribution in this problem is nonuniform (irregular) and therefore requires high-level data structures such as graphs, trees or unstructured meshes. The density of the data points in most cases is sparse (note that sparsity in applications is an area of research [47] that involves challenging issues in parallel computing). An effective parallel solution of *irregular* computations poses greater challenges because of the need for facilities for dynamic creation of work and dynamic load balancing.

In parallel systems, two types of latencies are incurred [4, 28]: *communication* latency due to remote accesses and *synchronization* latency due to data dependencies. Conventional message passing Massively Parallel Processing systems (MPPs) [7] that follow a von Neumann model of computing or Single Program Multiple Data (SPMD) model do not yield high performance if such latencies are frequent in the parallel solutions employed. Several techniques at the hardware level (such as superscalar, superpipelined, VLIW, prefetching) [26] have been used to hide or tolerate both communication and

synchronization latencies. From the software perspective, multithreading is the general technique.

Multithreading hides the communication and synchronization latencies of parallel computers by overlapping computation with communication and moves away from the traditional SPMD programming model. Traditional parallel maximum flow algorithms do not provide very efficient performance results. Multithreaded architectures have been promoted as potential processing nodes for future parallel systems due to their ability to overlap computation with communication.

However, very little work has been reported in the literature on the design of multithreaded algorithms for the maximum flow problem. This lack motivates our present research and we consider multithreaded algorithm as an alternative method for this thesis. We have developed a multithreaded algorithm for the maximum flow problem and implemented it in a shared memory environment using OpenMP.

1.3 Thesis Organization

This thesis addresses the design, development and implementation of a multithreaded algorithm for the maximum flow problem. In this chapter, we have discussed about the maximum flow problem, its applications and the reason behind developing a multithreaded maximum flow algorithm. The rest of the thesis is organized as follows. We describe the related work in the second chapter. In the third chapter, we discuss about multithreading and briefly describe some multithreaded platforms. We explain the se-

quential algorithm of Malhotra, Pramodh Kumar and Maheswari (MPM) [36] and our multithreaded algorithm in the fourth chapter. In Chapter five, we show our theoretical analysis. We give our experimental results in the sixth chapter. We present our conclusion and future work in chapter seven.

Chapter 2

Related Work

The maximum flow problem was born from applications [13] in the 1940's and 1950's. Since then it has developed into a strong theoretical topic with many practical applications and numerous algorithmic issues. In 1950's pseudo-polynomial time algorithms were developed (pseudo-polynomial time algorithm means that the complexity of the algorithm is a polynomial function of input size n and some largest number K). The first polynomial time algorithms for the maximum flow problem were developed in the 1970's and since then constant progress has resulted in faster algorithms. Goldberg [22] gives a very extensive survey on maximum flow algorithms. In Table 2.1 concise information about some remarkable sequential maximum flow algorithms for a transport network with m edges, n vertices and U as the highest capacity of the edges is given. The most notable sequential and parallel maximum flow algorithms are discussed in sections 2.1 and 2.2 respectively.

Authors	Year	Complexity
Ford & Fulkerson	1956	$O(mnU)$
Edmonds & Karp	1972	$O(m^2n)$
Dinic	1970	$O(mn^2)$
Malhotra *et al.*	1978	$O(n^3)$
Goldberg & Tarjan	1985	$O(mn^2)$, $O(n^3)$, $O(mn\log(\frac{n^2}{m}))$
Cheriyan *et al.*	1989	$O(n^3/\log n)$
King *et al.*	1994	$O(mn + n^{2+\epsilon})$
Goldberg & Rao	1998	$O(min(n^{2/3}, m^{1/2})m\log(n^2/m)\log U)$

Table 2.1: Notable sequential maximum flow algorithms

2.1 Sequential Maximum Flow Algorithms

In 1956, Ford and Fulkerson [19] solved the maximum flow problem by developing a labeling algorithm and established the celebrated max-flow min-cut theorem. Ford and Fulkerson offered an extensive treatment of their labeling algorithm in [20]. They introduced the concept of an augmenting path. An augmenting path is a path from the source to the destination into which more flow can be pushed. The algorithm starts by labeling the source. A labeled vertex can assign a label to any of its neighboring unlabeled vertices if more flow can be pushed to this unlabeled vertex. In this process, if the destination can be labeled then there exists an augmenting path from the source to the destination. The algorithm terminates when an augmenting path cannot be found.

In this algorithm, there may be several alternative augmenting paths and performance depends on which of these augmenting paths is chosen. The labeling algorithm may require pseudo-polynomial time for bad choices of augmenting paths. Moreover, Ford and Fulkerson showed that for networks with arbitrary irrational edge capacities, the algorithm can perform an infinite sequence of augmentations and might converge to a flow value different from the maximum flow value. In the worst case, the complexity of Ford and Fulkerson labeling algorithm is $O(mnU)$ for a graph with m edges, n vertices and U as the highest capacity of the edges.

To avoid the slower convergence problem of the labeling algorithm, Edmonds and Karp [16] suggested a refinement to the labeling algorithm of Ford and Fulkerson. They showed that the flow should be augmented along a shortest path at each step where the shortest path is a path having the smallest number of edges. They showed that this refinement guarantees the number of computational steps for implementing the algorithm is independent of the capacities of the edges. Edmonds and Karp's algorithm has complexity $O(m^2n)$.

It is interesting to note that Dinic [15] worked independently and developed his layered network algorithm while Edmonds and Karp developed their algorithm. Dinic introduced the concept of layered network which is a very different approach from the algorithm of Edmonds and Karp. At each step of Dinic's algorithm, a layered network with a shortest length augmenting path from the source to the destination with respect to a particular flow f is created. The layered network is represented as a directed acyclic graph. Then

the flow is augmented from the source to the destination in the layered network and the flow is updated in the original network. This procedure is repeated until a flow f^* is found such that no layered network can be created with an augmenting path from the source to the destination. Dinic's algorithm achieves a complexity of $O(mn^2)$.

In 1978, Malhotra, Pramodh Kumar and Maheshwari (MPM) [36] developed a very simple, elegant and ingenious algorithm called MPM algorithm based on Dinic's layered network concept. In MPM algorithm, the flow is augmented not along a single path but along several paths simultaneously. Therefore, the MPM algorithm can be easily designed into a parallel and multithreaded implementation. This algorithm has $O(n^3)$ complexity. The various steps of MPM algorithm will be discussed in section 4.1.

Another algorithm called preflow push-relabel algorithm, introduced by Goldberg and Tarjan [24], follows a different approach from the other contemporary algorithms in the literature, since it does not use the concept of layered networks or augmenting paths. Also, in all the previous algorithms, one important constraint that is maintained while pushing flow is that the amount of flow into a vertex is equal to the amount flowing out. This constraint is relaxed in Goldberg and Tarjan's algorithm and leads easily to a parallel implementation, though not to a multithreaded implementation. In [24] Goldberg and Tarjan describe in detail their algorithm. The complexity of their algorithm is $O(mn^2)$ but can be made $O(n^3)$ if the vertices are considered in a certain order in the algorithm. They show that use of dynamic tree decreases the complexity of the algorithm to $O(mn \log(\frac{n^2}{m}))$. However, use of dynamic tree does not improve the performance of

maximum flow algorirhm in practical cases [3].

Other notable sequential maximum flow algorithms are Cheriyan *et al.* [11], King *et al.* [33] and Goldberg and Rao [23]. Cheriyan *et al.* [11] is a randomized algorithm of $O(n^3/\log n)$ time complexity. King *et al.* gives $O(mn + n^{2+\varepsilon})$ deterministic version of Cheriyan *et al.* [11]. Goldberg and Rao introduce a new algorithm based on assigning zero or unit arc lengths which depends on the residual arc flow value and the residual arc capacities. The algorithm has a time complexity of $O(min(n^{2/3}, m^{1/2})m \log(n^2/m) \log U)$.

2.2 Parallel Maximum Flow Algorithms

The parallel version of the maximum flow algorithm of the sequential preflow push-relabel algorithm by Goldberg and Tarjan [24] has been widely discussed in the literature. For parallel implementation, the computing models of Goldberg and Tarjan fit very well with exclusive-read, exclusive-write parallel random-access machine (PRAM) and distributed random-access machine (DRAM) computing models. For both models the total number of operations is $O(n^3)$ and the running time is $O(n^2 \log n)$ with $O(m)$ space for n processors.

Anderson and Setubal [3] implemented Goldberg and Tarjan's maximum flow algorithm [24] on Sequent Symmetry S81, a shared-memory parallel machine. In this implementation Anderson and Setubal made two contributions — concurrent relabeling of vertices and use of an adaptive data structure for maintaining the workpile. These contributions, however, resulted in slight performance improvement only. Anderson and

Setubal used sparse graph with 2^{14} vertices and dense graph with 500 vertices. Their relative speedup was 8.8 on sixteen processors.

Agarwal and Ng [1] implemented an ingenious version of Goldberg and Tarjan's algorithm [24] on a distributed shared memory parallel machine. This implementation radically differs from Anderson and Setubal's approach in the sense that the processors are designed in a pipelined manner and the transport network graph is partitioned among these pipeline of processors. Agarwal and Ng [1] used Multiprocessor Architecture for Rapid Simulations (MARS) for the implementation of their version of algorithm. MARS has fifteen identical processors. However, they used only six processors as their algorithm requires six different tasks at a time for execution. They used sparse graphs with 1000, 5000, 10000, 20000 and 30000 vertices for their experiment and achieved a speedup of 4.8 to 5.9 on six processors.

Träff [54] implemented a maximum flow algorithm on a distributed memory parallel machine using a layered network technique. The experiments were conducted on a transputer consisting of sixteen T800 processors. Both dense graphs (100, 200 vertices) and sparse graphs (500, 1000 and 2000 vertices) were used. The algorithm achieved a speedup of two to three on eight processors.

Nagy and Akl [38] introduced a dynamic algorithm for the maximum flow problem on a Reconfigurable Multiple Bus Machine (RMBM) architecture to study the dynamic allocation of modules to two processors subject to real time constraints. Computation and communication requirements of modules change over time in the module allocation

problem. This dynamic allocation problem inspired the authors in coining their dynamic maximum flow algorithm on the RMBM. In the dynamic maximum flow problem the number of vertices, the number of edges and the capacity of edges in the transport network change over time.

Recently, there has been a report [35] on the design and implementation of the preflow push-relabel algorithm of Goldberg and Tarjan [24] in OpenMP on a shared memory multiprocessor. This report considers various parallel programming interfaces and studies the programming and performance evaluation of these interfaces on one particular algorithm, the maximum flow algorithm. The platform used is the SGI Origin 2000. The parallel maximum flow algorithm implementation on this architecture gives a speedup of 1.5x on 6 processors for a sparse graph with 5000 vertices.

Note that in the literature, the preflow push-relabel algorithm [24] has been considered for parallel implementations. Thus far, this sequential algorithm has been regarded as the "best" sequential maximum flow algorithm [23]. However, from the multithreading perspective, the preflow algorithm is not an efficient algorithm as can be seen from the report by Lie [35], where the performance result is only 1.5x on 6 processors. Therefore, I have considered the MPM algorithm which lends easily to the multithreaded environment. In the MPM algorithm, the size of graph dynamically reduces during the execution of the algorithm. This decrease in the size of the graph creates immense load balancing problems in a traditional SPMD model. Hence, the MPM algorithm, has not been opted for parallel implementation in the literature. The irregularity and the asynchronicity

posed by this algorithm makes it a suitable candidate for the multithreaded approach.

Also, note that theoretically the MPM [36] and preflow push-relabel [24] algorithms have the same complexity of $O(n^3)$. Therefore, MPM algorithm can also be considered as good as push-relabel algorithms theoretically.

Chapter 3

Multithreading

A thread is a small set of instructions which are executed in sequence. In the traditional sequential computing model (also known as the von Neumann computing model), there is only one thread or flow of control. Traditional parallel computers that follow the von Neumann computing model suffer from communication and synchronization latencies that are caused by remote memory access in a distributed shared memory system and ordering among the execution of instructions in different processors. Multithreading is a programming paradigm in which a single program is broken into multiple threads of control which interact to solve a single program. Multithreading tries to overlap computation with communication to overcome communication and synchronization latencies.

There are several multithreaded environments such as Pthreads [34], Tera [8], Java [21], EARTH [27], Charm++ [32], Cilk [45], and OpenMP [9]. Some of these multithreaded systems are software-based while others require software as well as hardware support for

multithreading. These multithreaded environments support either coarse-grained threads (number of instructions per thread is large and the context switching time among the threads is also large) or fine-grained threads (number of instructions per thread is small and the context switching time among the threads is also small). Some of the multithreaded systems follow control flow mechanism (the order of execution is determined explicitly by the code of the program) while others follow the dataflow mechanism (the order of execution of instructions is determined by the order of the availability of data). Here we will discuss these multithreaded systems briefly.

3.1 Pthreads

The POSIX threads or Pthreads [34] are operating system (OS) threads. POSIX threads is a standard for operating system threads while there are several implementations of POSIX threads by different vendors. OS threads exploit parallelism at a coarser grain level and thus must execute a higher number of instructions between thread switching. They are also called library based systems since the required functions for managing threads are implemented as library. The library is implemented as user-level library or kernel-level library. The library provides a set of multithreading primitives to manage the threads created by programmer on top of OS threads. In this approach the management of threads requires a few system calls such as creation of threads, start and end of threads which are costly in terms of execution cycles. Therefore, we have not considered Pthreads in this study.

3.2 Tera

Tera is a multithreaded architecture [8] that requires hardware as well as software support for multithreading. In Tera, each processor can support 128 threads. Tera does not have any data cache. Tera is a shared memory machine with nonuniform access time. In Tera multithreaded system the latency incurred in remote memory access is hidden by extremely fast switching of threads. Tera provides good throughput for a highly multithreaded fine-grained program by use of thread parallelism. However, we have not considered Tera since Tera is not available at University of Manitoba and used mainly in national research labs such as NASA Ames [39] and the San Diego Supercomputer Center where the machine is installed.

3.3 Java

Java [21] is a multithreaded language for single and multiprocessor computers. Java is an interpreter-based language and interpreted Java code can be executed in any system supporting a Java virtual machine. Though Java can be used for parallel programming, the main focus of Java is on internet programming. Java supports coarse-grained threads and context switching between Java threads require significant amount of time. Therefore, we have not considered Java. Also, Java does not support fine-grained threading.

3.4 EARTH

Efficient Architecture for Running THreads (EARTH) [27] is a multithreaded dataflow architecture that does not follow the traditional von Neumann model of computing in traditional parallel computers or the Tera multithreaded architecture. EARTH threads are very versatile and EARTH supports both fine-grained and coarse-grained threads. In EARTH, threads are scheduled depending on control and data dependencies, which circumvent synchronization and communication latencies quite significantly. Though there is a simulator version of the EARTH and it is possible to access that version remotely, remote access is always very difficult. Moreover, the language used by this architecture is called Threaded-C [46] which is not a user-friendly language. In addition, the model of programming is totally different from Cilk or OpenMP and the comparison of my algorithm on EARTH will not be feasible, if we were to use EARTH.

3.5 Charm++

Charm++ [32] is an object-oriented language that supports multithreading. One of the important differences between Charm++ and the other models is that Charm++ is a message driven model. That is, in Charm++ computations (or threads) are activated upon receiving a message.

A Charm++ program consists of chares, Charm++ concurrent objects encapsulating medium-grained units of work. These chares are distributed among the available pro-

cessors. The chares have some public entry methods, private and public methods and data. Chares can be created dynamically and they can send messages to one another to invoke methods asynchronously. On the other hand, the entry methods of chare can be executed in separate user-level threads provided by Charm++.

This is a good programming model. However, we have not been able to install it in the machines of our department.

3.6 Cilk

Cilk [45] is an ANSI C-based language for multithreaded parallel programming. Cilk is effective for exploiting dynamic, highly asynchronous parallelism, which might be difficult to write in data-parallel or message-passing style. Cilk system can be configured to gather various runtime statistics for a whole program and parts of a program. The Cilk runtime was built on its theoretical foundation. Cilk gives a way to analyze and get experimental statistics such as parallelism, total work, critical-path length, wall-clock running time for a program or part of a program. The critical-path length of a program can be defined as the maximum time required to execute the longest path threads created by the Cilk program. Critical-path length and total work are two parameters that can be used to predict the performance of a program in Cilk [31]. That is, measuring the execution time and critical-path length one can theoretically analyze the performance of the algorithm. The total work of a multithreaded computation is the total time to execute all the operations in the computation serially on one processor. In Cilk, a thread spawns one

or more threads, thereby creating a parent-child relationship among the threads. The idea behind Cilk is that a programmer should concentrate on structuring the program to expose parallelism. The runtime scheduler of Cilk takes the responsibility of mapping the dynamically unfolded computations onto available processors to execute the program efficiently. Cilk insulates the programmer from many low-level implementation details such as load balancing, paging and communication protocols. Unlike other multithreaded languages Cilk is algorithmic in the sense that the runtime system guarantees efficient and predictable performance [6].

Cilk is a very elegant language and the MPM algorithm maps not only very directly onto the Cilk programming paradigm but also gives a very efficient multithreaded program design. Also, among the multithreaded environments, Cilk is easy to learn and it is installed in the SMP machine at the Department of Computer Science. We have implemented the multithreaded maximum flow algorithm in Cilk. Though our multithreaded algorithm maps very well in Cilk paradigm we had to switch to OpenMP as we have faced some problems with Cilk installed on our system. We obtained the performance of the algorithm by using the Cilk statistics and running the algorithm on different number of processors for transport networks with different number of vertices and edges. The performance result of the algorithm is not satisfactory. We did not achieve good speedup. However, after checking the example programs provided by the Cilk-developers we also found some anomalous behavior for some of their programs (For running example programs provided with Cilk we have found for an example program speedup of 1.5 for 8

processors while in the Cilk manual [45] the speedup for the same program is mentioned as 7.8 for 8 processors). Though we checked the machine configuration we could not find the reason for this anomalous behavior of Cilk. We contacted the Cilk-developers, and since we did not receive reply for three months we could not progress.

3.7 OpenMP

OpenMP [9, 40] stands for Open specifications for Multi Processing. It is developed as an Application Programming Interface (API) for multithreaded environment for shared memory systems through collaboration among researchers from the hardware and software industry, government and academia. The OpenMP API supports C/C++ and Fortran on Linux, Unix and Windows NT architecture by providing a set of compiler directives and library routines. As most of the constructs of OpenMP are compiler directives, an OpenMP program can be compiled either on a sequential or parallel computer without any modifications to any parallel code. The compiler automatically disregards the compiler directives provided in the language while compiling on a sequential computer. It is an easy language to learn.

OpenMP runs on a shared memory multiprocessor. One notable feature of OpenMP is that it supports loop-level data parallelism as well as SPMD (Single Program Multiple Data) model as in MPI. In loop-level parallelism such as a *for* loop, each iteration of the loop can be forked as a thread, thereby supporting fine-grained threading. The threads are divided among the processors automatically. On the other hand, to employ the SPMD

model of programming, OpenMP allows the programmer to divide the data into chunks where each chunk is a thread. Each thread is distributed to a processor for execution. OpenMP uses a concept called parallel regions (a region is a sequence of instructions or thread) and these parallel regions are executed independently by different processors. OpenMP, therefore, can support coarse-grained threading in SPMD model.

OpenMP provides several automatic load balancing techniques such as static, dynamic and guided scheduling of work [9]. In static load balancing technique fixed amount of work is distributed among threads in a round-robin fashion. On the other hand, in dynamic and guided scheduling the size of work distributed among threads change with time.

I have implemented the maximum flow algorithm in OpenMP.

Chapter 4

A Multithreaded Maximum Flow Algorithm

We have developed our multithreaded maximum flow algorithm using the sequential algorithm of Malhotra, Pramodh Kumar and Maheshwari (MPM) [36]. In sections 4.1 and 4.2 we have discussed MPM algorithm and our multithreaded algorithm respectively.

4.1 Sequential MPM Algorithm

In 1978, Malhotra, Pramodh Kumar and Maheshwari [36] developed MPM algorithm. This very simple, elegant and ingenious algorithm is based on Dinic's layered network concept [15]. In this approach the original network is first labeled using Breadth First Search (BFS). The labeling starts from the source, s, which is labeled as having distance 0 from the source. An unlabeled vertex is labeled from a labeled vertex in BFS manner.

An unlabeled vertex v is labeled from a labeled vertex u if (u, v) is an edge and $f(u, v) < c(u, v)$ or (v, u) is an edge and $f(v, u) > 0$. In other words, the labeling is done from vertex u to vertex v if there is a residual edge from vertex u to vertex v. That is, for each edge (u, v), if the capacity exceeds flow then more flow can be pushed from u to v. On the other hand, for each edge (v, u), if there is a positive flow then this flow can be pushed back from u to v. For both of these cases we say that there is a residual edge (u, v) in the original network. For the first case, the residual capacity is the amount by which the capacity of the edge exceeds the flow in the edge. In the second case, the residual capacity is the amount of flow in the edge. If edge (u, v) is saturated (flow is equal to capacity) then this edge is not considered for labeling v from u. On the other hand, if there is no flow in the edge (v, u) then v is not labeled from u. After labeling, an auxiliary layered network is created where the vertices at distance one away from the source are in layer one; the vertices at distance two away from the source are in layer two; in general, the vertices that are at a distance i from the source are in layer i. If we let L to be the maximum number of layers, then the source, s, is in layer zero, all the other vertices other than the sink, t, is in one of the layers from 1 to $L - 2$ and t is in layer $L - 1$. The resulting network is a smaller network called the auxiliary layered network. The edges between two layers are then connected in this layered network. Note that the algorithm connects the edges in this network in such a way that each edge is considered as a forward edge. That is, there is no edge connecting two vertices from layer i to $i - 1$, only from layer i to layer $i + 1$. The auxiliary layered network thus created is a directed

acyclic graph (DAG). In this directed acyclic graph, there is one or more paths from the source to the sink through which flow can be augmented. That is, there is one or more augmenting paths from the source to the sink in the auxiliary layered network.

The next step of the MPM algorithm is to select a vertex say v_i in the layered network. The criterion for selecting v_i is as follows. The in-potential and out-potential of each vertex is calculated. The in-potential is the maximum amount of flow that can be pushed to the vertex and the out-potential is the maximum amount of flow that can be pushed away from the vertex. In-potential and out-potential of every vertex is calculated from the auxiliary capacity of edges incident on the vertex. The auxiliary capacity of an edge in an auxiliary layered network is the difference between capacity and flow in that edge. The in-potential of a vertex is the summation of auxiliary capacity of the incoming edges of that vertex. Similarly, the out-potential of a vertex is the summation of the auxiliary capacity of the outgoing edges of that vertex. The source, s, has no incoming edges and the sink, t, has no outgoing edges. For the source, s, the potential is its out-potential and for the sink, t, the potential is its in-potential. For any other vertex the potential is the minimum of its in-potential and out-potential. The $potential = min(in - potential, out - potential)$ gives the maximum flow that a vertex can handle. After obtaining the potential for all vertices, the vertex which has the least potential is selected as the target vertex (v_i) from which flow can be pushed and pulled. Vertex (v_i) is called the minimum potential vertex and its potential is called min-potential. The selection of the vertex in such a manner guarantees that there is

never excess flow being pushed at a vertex and therefore the flow is always positive.

The flow is then pushed from v_i to the destination through several paths simultaneously. In the same way, an equal amount of flow is pulled back from v_i to the source through several paths simultaneously. Since the graph is a DAG, the flow can be pushed and pulled safely without worrying about any cycles. Also, since the layers are independent of one another, we can safely apply parallelization to this network.

After these push and pull-back operations, some edges may get saturated and are therefore deleted. The deletion of edges may leave some vertices isolated and so, they are deleted too. The selection of vertices, pushing and pulling of flow and deletion of vertices and edges are repeated until the layered network becomes disconnected.

The last step is to update the flow in the original network. The next iteration starts by constructing an auxiliary layered network again and the push-pull and deletion operations are repeated. The algorithm terminates when no layered network can be created with augmenting paths from the source to the destination, *i.e.*, the flow f^* in the transport network is maximum if there is no augmenting path from the source to the destination. The flow in the original transport network is now the maximum flow.

MPM lagorithm is explained with examples in [5, 42]. Now we will explain how MPM algorithm works for the transport network of Figure 4.1. For every edge (u, v) two weights are associated—capacity and flow. For example, in Figure 4.1 the capacity and flow of edge (s, a) is 11 and 0 respectively.

When labeling algorithm is applied on the transport network of Figure 4.1 the source,

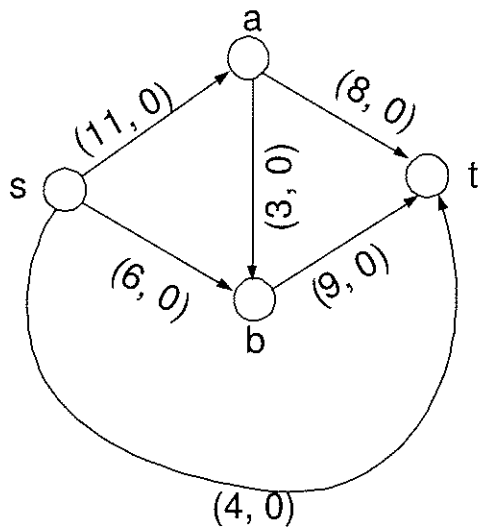

Figure 4.1: An example of a transport network with 0 flow

s, is labeled as 0. There is an edge from s to a with capacity 11 and flow 0. Hence, $11-0=11$ units of flow can be pushed from s to a. Similarly $4-0=4$ units of flow can be pushed from s to t and $6-0=6$ units of flow can be pushed from s to b. Hence, a, b and t are labeled as 1. As a result, the auxiliary layered network shown in Figure 4.2(a) is generated. This auxiliary layered network has two layers. The source, s, is in layer 0 and the other three vertices a, b and t are in layer 1. There is no edge between vertices a and b in the auxiliary layered network as they are in the same layer. Vertices a and b have no outgoing edges. These two vertices and the two edges incident on them are deleted and the auxiliary layered network shown in Figure 4.2(b) is generated. In this layered network both s and t have potential 4 and s is selected as the minimum potential vertex. $g_1 = 4$ units of flow is pushed from s to t. The edge (s,t) becomes saturated and the edge (s,t) is deleted fom the auxiliary network and the auxiliary layered network

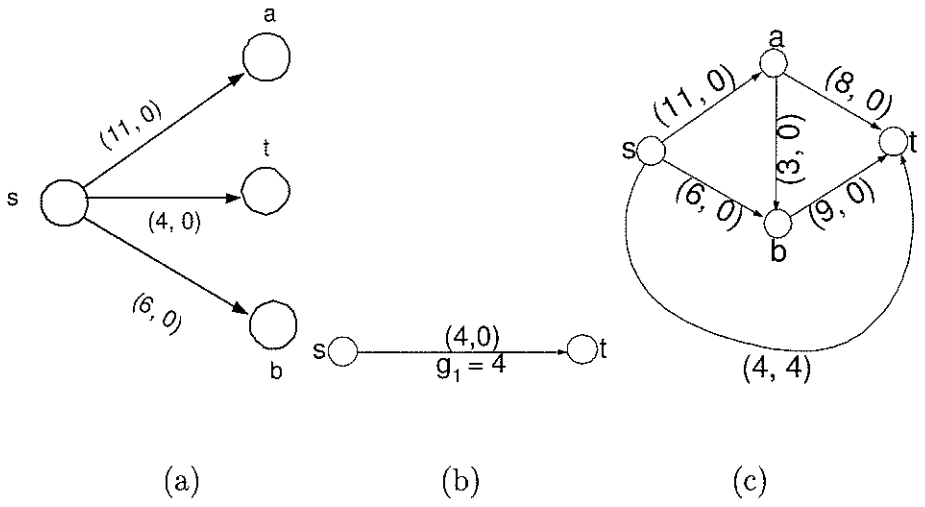

Figure 4.2: Flow in transport network and auxiliary layered network with two layers

becomes disconnected. The flow in edge (s, t) in the original network is updated as shown in Figure 4.2(c).

As the auxiliary layered network shown in Figure 4.2(b) is disconnected after push operation labeling is again applied on the transport network shown in Figure 4.2(c). In this labeling operation the source, s, is labeled as 0. $11 - 0 = 11$ units of flow can be be pushed from s to a and $6 - 0 = 6$ units of flow can be pushed from s to b. Hence, vertices a and b are labeled as 1. $8 - 0 = 8$ units of flow can be pushed from a to t. Therefore, t is labeled as 2. After this labeling operation the auxiliary layered network with three layers is generated as shown in Figure 4.3(a). In this layered network the source, s, is in layer 0, vertices a and b are in layer 1 and the sink, t, is in layer 2. The source, s, has potential $11 + 6 = 17$, vertex a has potential $min(11, 8) = 8$, vertex b has potential $min(6, 9) = 6$ and the sink, t, has potential $8 + 9 = 17$. Vertex b has the minimum potential and it

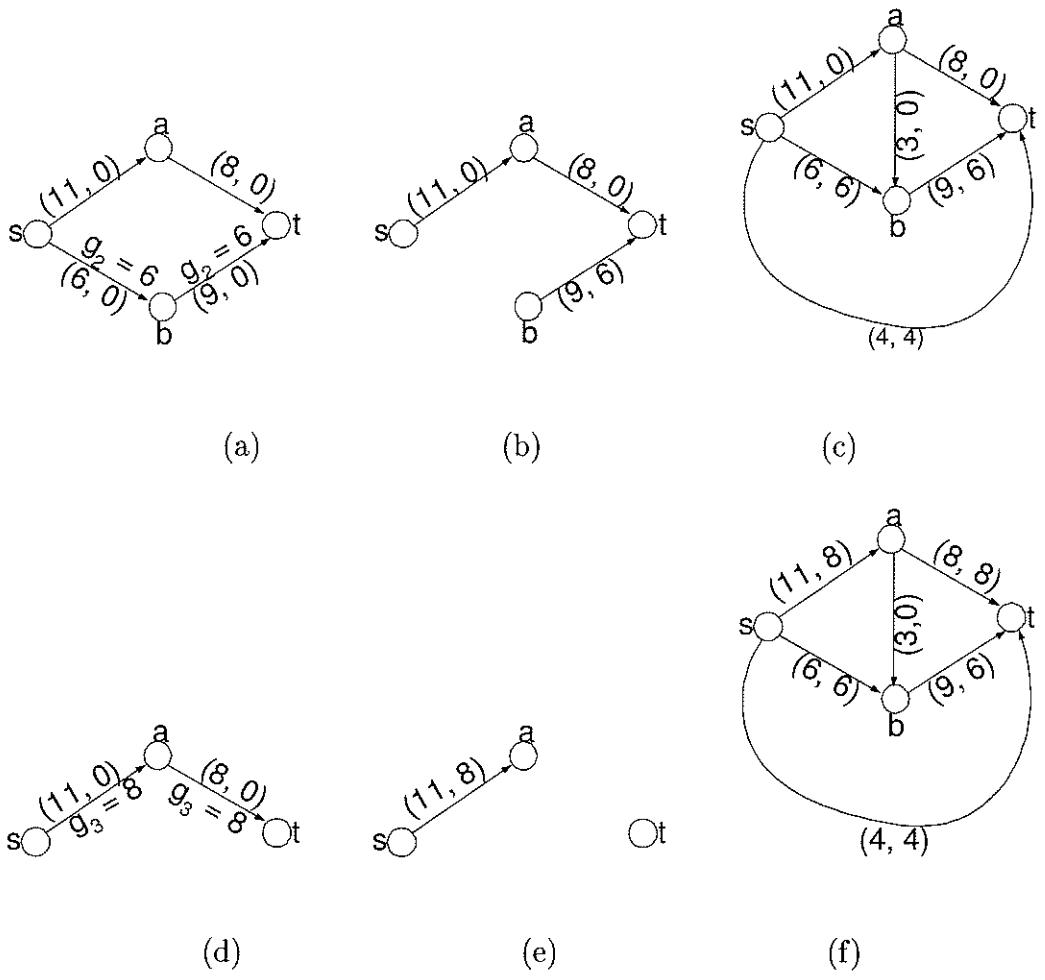

Figure 4.3: Flow in transport network and auxiliary layered network with three layers

is selected as the minimum potential vertex. Now $g_2 = 6$ units of flow is pushed from b to t and $g_2 = 6$ units of flow is pulled back from s to b. As a result of push-pull operation edge (s, b) is saturated and is deleted from the auxiliary layered network as shown in Figure 4.3(b). The flow in the original transport network is updated as shown in Figure 4.3(c). Now in the auxiliary network of Figure 4.3(b), as vertex b has no incoming edges its outgoing edge (b, t) and vertex b are deleted and the auxiliary layered

network of Figure 4.3(d) is generated. In the auxiliary layered network of Figure 4.3(d), the source, s, has potential 11, vertex a has potential $min(11, 8) = 8$ and the sink, t, has potential 8. The minimum potential vertex is a and minimum potential is 8. Now $g_3 = 8$ units of flow is pushed from vertex a to the sink, t, and $g_3 = 8$ units of flow is pulled back from the source, s, to vertex a. As a result of push-pull operations edge (a, t) becomes saturated and it is deleted from the auxiliary layered network (Figure 4.3(e)). The flow in the original transport network is also updated which is shown in Figure 4.3(f). In Figure 4.3(e) vertex a has no outgoing edges. Hence, the incoming edge (s, a) of vertex a is deleted and vertex a is also deleted. As a result, the auxiliary layered network becomes disconnected.

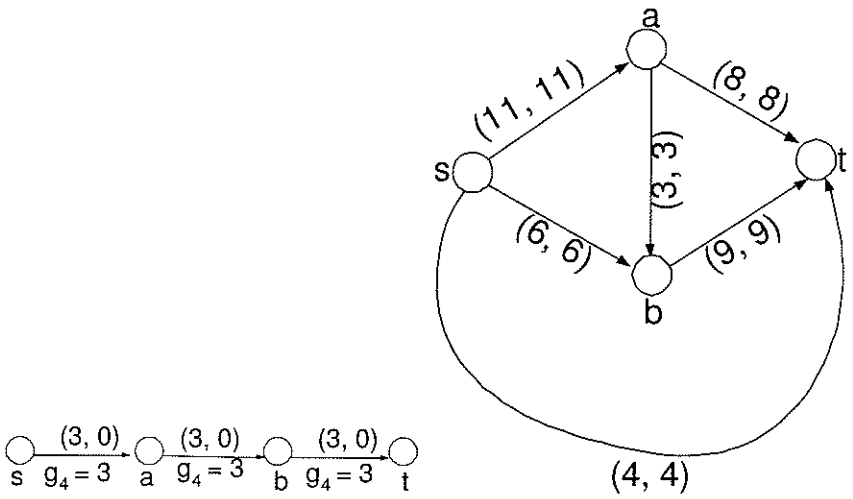

Figure 4.4: Flow in transport network and auxiliary layered network with four layers

The labeling operation of vertices is applied on the transport network of Figure 4.3(f).

The source, s, is labeled as 0. $11-8=3$ units of flow can be pushed from s to a. Hence, vertex a is labeled as 1. $3-0=3$ units of flow can be pushed from a to b. Therefore, vertex b is labeled as 2. $9-6=3$ units of flow can be pushed from b to the sink, t. Hence, t is labeled as 3. After this labeling operation the auxiliary layered network with four layers is generated (Figure 4.4(a)). In this auxiliary layered network the source, s, is in layer 0, vertex a is in layer 1, vertex b is in layer 2 and the sink, t, is in layer 3. The source, s, has potential 3, vertex a has potential $min(3,3)=3$, vertex b has potential $min(3,3)=3$ and the sink, t, has potential 3. The source, s, is selected as the minimum potential vertex and the minimum potential is 3 units. Now, $g_4=3$ units of flow is pushed from the source, s, to vertex a. Vertex a pushes $g_4=3$ units of flow to vertex b. Vertex b then pushes $g_4=3$ units of flow to the sink, t. As a result of these push operations, edges (s,a), (a,b) and (b,t) become saturated and these edges are deleted. As vertices a and b have no incoming or outgoing edges, they are also deleted and the auxiliary layered network becomes disconnected. The flow is also updated in the original transport network (Figure 4.4(b)).

Labeling operation is again applied on the transport network of Figure 4.4(b). The source, s, is labeled as 0. The three outgoing edges from the source, s, are (s,a), (s,b) and (s,t) which are saturated and cannot be reached from the source, s. Hence, no vertices can be labeled from the source, s. Therefore, no auxiliary network can be created. As a result, the algorithm terminates. The summation of flow in the outgoing edges from the source, s, is $11+6+4=21$ and the summation of flow in the incoming edges to the sink,

t, is $8 + 9 + 4 = 21$. Hence, the flow of 21 units is the maximum flow for the transport network of Figure 4.1.

4.2 A Multithreaded Maximum Flow Algorithm

The MPM algorithm [36] proceeds in several phases and each phase consists of several steps. Each phase is dependent on previous phases. Hence, there is some synchronicity. However, there is asynchronicity within the phases. This asynchronicity is where we will apply multithreading. This asynchronicity is one of the main reasons that motivated this research towards multithreaded paradigm.

We apply multithreading to the labeling of vertices, determining minimum potential vertices and push-pull operations. All other steps are same with sequential MPM algorithm. We will discuss about the parallelization in the following sections.

4.2.1 Multithreading in Labeling Vertices

In each phase of the MPM algorithm, a layered network with a shortest augmenting path from the source, s, to the sink, t, is created. The creation of the layered network can be parallelized since this is nothing but applying labeling of vertices in parallel. Labeling of vertices can be done by applying a shortest path algorithm. For labeling vertices of a graph the distance from vertex u to vertex v is defined as 1 if any flow can be pushed from u to v. On the other hand, the distance from vertex u to vertex v is defined as ∞ if no flow can be pushed from u to v. Then the shortest path algorithm is

applied on the graph after assigning distance to each pair of vertices in the graph. Many sequential shortest path algorithms exist, but the one that is amenable to parallelization is the Dijkstra's algorithm [14]. The parallel version of Dijkstra's algorithm for disributed memory machine is explained in [25]. The beauty of this distributed memory algorithm is that it can be easily implemented for shared memory machine. We have used the parallel version of Dijkstra's algorithm [14] for labeling of the vertices. Note that there are several parallel shortest path algorithms and their implementations such as Chandy and Mishra [10], Paige and Kruskal [41], Thulasiraman *et al.* [53] and Thulashiraman and Khokhar [52].

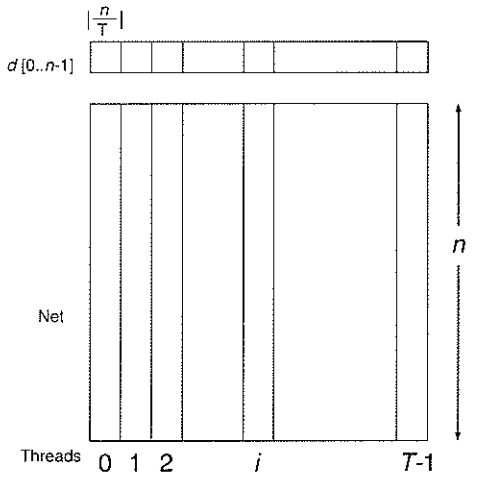

Figure 4.5: The partitioning of labeling array d and the adjacency matrix *Net* among T threads

We represent the original transport network using an adjacency matrix *Net* and the labels of the vertices by a one dimensional array d. The adjacency matrix *Net* of the original transport network and the labeling vector d is partitioned among threads using

1-D block mapping (In 1-D block mapping matrices are partitioned either row-wise or column-wise) as shown in Figure 4.5. Here, we are assuming that there are n vertices from vertex 0 to vertex $(n-1)$ and T threads from thread 0 to thread $T-1$. The source vertex, s, is vertex 0 and the sink vertex, t, is vertex $n-1$. Each of the T threads is assigned $\frac{n}{T}$ consecutive columns of the adjacency matrix. Thread i initializes from vertex $(i * \frac{n}{T})$ to vertex $((i+1) * \frac{n}{T} - 1)$. At first the labeling vector d is initialized using the Initialize algorithm of Figure 4.6.

```
Procedure Initialize(d, n, T, ThreadNumber, Net)
begin
    if(ThreadNumber = 0)
        d[0] ← 0;
        for(i ← 1 to n/T - 1)
            Mark vertex i as not used for labeling;
            if(Net[0][i] contains an edge)
                d[i] ← 1;
            else
                d[i] ← ∞;
            endif
        endfor
        Mark vertex 0 as used for labeling;
    else
        for(i ← ThreadNumber*n/T to (ThreadNumber+1)*n/T - 1)
            Mark vertex i as not used for labeling;
            if(Net[0][i] contains an edge)
                d[i] ← 1;
            else
                d[i] ← ∞;
            endif
        endfor
    endif
end
```

Figure 4.6: Initialize algorithm

After applying *Initialize* procedure the next task is to find a candidate vertex which can be used for labeling more vertices. In the next step each thread finds the vertex which is not used for labeling. Each thread executes the MinVert algorithm of Figure 4.7.

```
Procedure MinVert(d, n, T, ThreadNumber, Net)
begin
    find the vertex x in the range (ThreadNumber*n/T..(ThreadNumber+1)*n/T-1)
    which is not used for labeling and which is not labeled as ∞;
    if no such x can be found
        return n;
    else
        return x;
    endif
end
```

Figure 4.7: MinVert algorithm

After execution of the above mentioned code by each thread the master thread (whose ThreadNumber is 0) finds the minimum from the calculated minimum of all the minimums calculated by the threads. If the minimum u is in the range $(1..n-2)$ the Label procedure of Figure 4.8 is called.

```
Procedure Label(d, n, T, Net, u)
begin
    for(i ← 1 to n/T-1)
        if(Net[u][i] or Net[i][u] contains an edge and flow can
        be pushed from u to i and d[i] > d[u]+1)
            d[i] ← d[u]+1;
        endif
    endfor
    Mark vertex u as used for labeling;
end
```

Figure 4.8: Label algorithm

After calling *Label*, *MinVert* is called. if *MinVert* does not return a value in the range $[1..n-2]$ the whole labeling procedure ends.

The labeling of the vertices in the original transport network consists of initializing the vertices and then repeatedly applying probable candidate generation, synchronization and then selection of the candidate vertex for labeling, synchronization and testing for the termination of the labeling operations. All these works are done by LabelAll algorithm. The pseudo-code of LabelAll algorithm for labeling vertices is given below in Figure 4.9. This algorithm uses synchronization to synchronize work among different threads. LabelAll algorithm terminates when calling MinVert algorithm does not generate any candidate vertex for labeling other unlabeled vertices.

```
Procedure LabelAll(d, n, T, ThreadNumber, Net)
begin
    Call Initialize(d, n, T, ThreadNumber, Net) by all the threads;
    synchonize;
    flag ←true;
    while(flag = true)
        Call MinVert(d, n, T, ThreadNumber, Net) by all the threads;
        synchronize;
        Use the Master Thread to combine the data returned from
        MinVert and store the value in u;
        synchronize;
        if(u is not in the range from 1 to n-2)
            flag ←false;
        endif
    endwhile
end
```

Figure 4.9: LabelALL algorithm

4.2.2 Multithreading in Determining the Minimum Potential Vertex

After labeling of the vertices is done we check whether the sink, t, has been labeled or not. If the sink, t, is not labeled the algorithm terminates. Otherwise a layered network is created with L layers where the sink is labeled as $L-1$. Now this L layers are distributed among T threads. Each thread contains $\frac{L}{T}$ layers and each thread calculates probable minimum potential vertex from its data. The master thread combines all the data to get the minimum potential vertex. In Figure 4.10 we have shown how the layers of an auxiliary layered network are distributed among the available T threads. In this figure it is assumed that every thread computes the minimum of the potential of vertices of two layers.

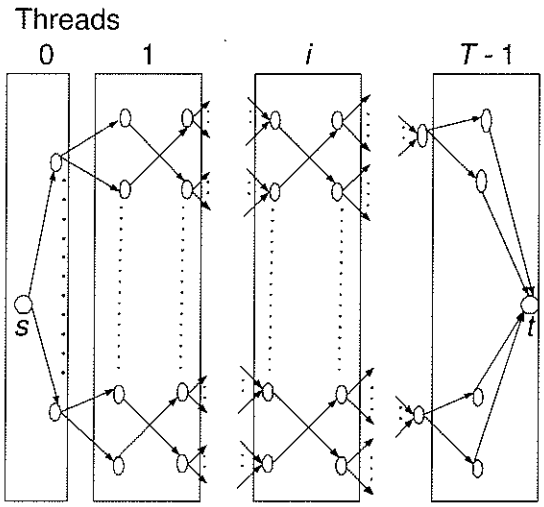

Figure 4.10: Distribution of layers in an auxiliary layered network for computation of the minimum potential vertex

4.2.3 Multithreading in Push-Pull

Push-pull operations are also done in parallel. If the minimum potential vertex is in layer i then at first flow is pushed to layer $i+1$ vertices and flow is pulled from layer $i-1$ vertices. The threads take the vertices of layers $i+1$ and $i-1$ to which flow is pushed or pulled. The procedure for push operation is given in Figure 4.11.

```
Procedure Push(u, potential)
begin
    RemainingPotential ← potential;
    lock the vertex u
    while(RemainingPotential > 0)
        v ← first vertex in the successor list of u;
        lock the vertex v;
        auxiliary_capacity ← difference of capacity and flow in (u,v);
        f ← min(RemainingPotential, auxiliary_capacity);
        RemainingPotential ←RemainingPotential - f;
        increase flow of(u,v) by f and increase flow in the corresponding
        edge in the original transport network by f;
        if(u,v) is saturated
            delete the edge from the layered network;
        endif
        if(v ≠ t)
            Enqueue (v,f) in the queue Q;
        endif
        unlock the vertex v;
    endwhile
    unlock the vertex u;
    while (Q ≠ ∅)
        Dequeue (v,f) pair from Q and Push(v,f);
    endwhile
end
```

Figure 4.11: Push algorithm

In the above mentioned *Push* procedure we lock vertices u and v before flow is pushed

from vertex u to vertex v so that two or more threads cannot access the same vertex at the same time. A thread unlocks a vertex when the pushing of flow through that vertex is completed. The vertices are locked to ensure the atomicity of push operation. We have showed in Figure 4.12 how push and pull operations start from the layer i of auxiliary network.

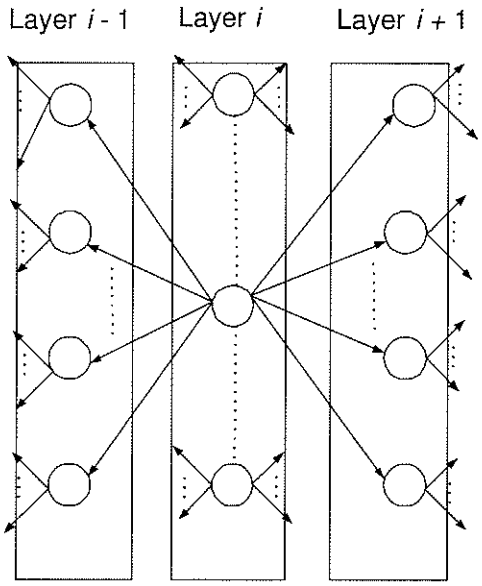

Figure 4.12: Push and pull operations from Layer i

The pull operation is quite similar to push operation. Vertices are also locked during pull operation. The only difference is that pull terminates when flow reaches the source, s, while push terminates when the flow reaches the sink, t.

Chapter 5

Theoretical Analysis

We have applied coarse-grained multithreading to labeling of vertices, calculation of minimum potential vertex and push-pull operations. We use SPMD model for our computation in OpenMP. There are no standard models to analyze computations in multithreading. For example, the theoretical analysis in [51] is for a specific data-flow architecture EARTH. However, we present some of our analytical results for application of multithreading to labeling of vertices and determining minimum potential vertex. We also explain why it is difficult to analyze push-pull of flow.

5.1 Analysis of Labeling of Vertices

In this approach, given n vertices and T threads, we allocate $\lceil \frac{n}{T} \rceil$ or $\lfloor \frac{n}{T} \rfloor$ vertices to each thread (Here, $\lceil a \rceil$ means the lowest integer $\geq a$ and $\lfloor a \rfloor$ means the greatest integer $\leq a$). Each thread executes the same code of labeling vertices. Then there is a synchronization

barrier. After this barrier each thread executes the same code for calculation of probable candidate for the next labeling phase. A synchronization barrier is executed again. The master thread then calculates the next candidate for labeling using the computation of all the threads. Another synchronization barrier is executed and the above mentioned steps are continued until no candidate for labeling operation is found. In the worst case, these steps will be executed n times.

Let us assume that 1 unit time is required to label a vertex from a candidate vertex and each comparison operation for finding candidate vertex also requires 1 unit of time. Then the time spent in one iteration for labeling by a thread is $(\lceil \frac{n}{T} \rceil + t_{barrier})$ unit. In the worst case, there will be n iterations. Hence, the total time spent in labeling vertices by a thread is $T_{Lab} = \lceil \frac{n}{T} \rceil * n + n * t_{barrier}$. Similarly for a candidate vertex generation, time is spent in finding probable candidate by each thread and then choosing the candidate vertex from probable candidates by the master thread. These operations require synchronization operations after probable candidate generation by each thread and then determining the candidate by the master thread from the probable candidate vertices. Therefore, one iteration for candidate vertex generation for labeling involves $\lceil \frac{n}{T} \rceil$ time for candidate vertex generation by each thread, $t_{barrier}$ time for synchronization after probable candidate vertex generation by each thread, T time for the master thread to compute the candidate vertex from probable T candidate vertices and finally $t_{barrier}$ time for synchronization before starting other operations. Hence, one iteration of candidate vertex generation requires $(\lceil \frac{n}{T} \rceil + T + 2 * t_{barrier})$ unit time. In the worst case, all these

steps are executed n times. Therefore, the time to find candidate vertex for labeling is

$T_{Can} = (\lceil \frac{n}{T} \rceil + T) * n + 2 * n * t_{barrier}$

Therefore, if we consider the case when the number of threads T is equal to the number of processors p the total coarse-grained parallelism time for labeling is

$$\begin{aligned} Total_{CGP} &= T_{Lab} + T_{Can} \\ &= \lceil \frac{n}{T} \rceil * n + n * t_{barrier} + (\lceil \frac{n}{T} \rceil + T) * n + 2 * n * t_{barrier} \\ &= (2 * \lceil \frac{n}{T} \rceil + T) * n + 3 * n * t_{barrier} \end{aligned}$$

If the computation of labeling of vertices were done by sequential operations then no synchronization barrier would be required. In that case, in each iteration of labeling by a candidate vertex involve n operations and each iteration of candidate vertex generation involves also n operations. Therefore, in the worst case for n iterations, the total time spent in labeling operation is $Total_{SEQ} = 2 * n^2$. Hence, if we consider the case when the number of threads T equals the number of processors p the speedup achieved by our multithreaded labeling method is

$$\begin{aligned} speedup &= \frac{Total_{SEQ}}{Total_{CGP}} \\ &= \frac{2 * n^2}{(2 * \lceil \frac{n}{T} \rceil + T) * n + 3 * n * t_{barrier}} \\ &= \frac{T}{\frac{\lceil \frac{n}{T} \rceil}{\frac{T}{n}} + \frac{1}{2} + \frac{3 * t_{barrier}}{2 * n * T}} \end{aligned}$$

Now, if we consider the situation when $n >> T$ and $n >> t_{barrier}$, the speedup approaches T or p. Hence, if the number of vertices in the transport network is large, the theoretical speedup is good.

5.2 Analysis of Determining Minimum Potential Vertex

The method to determine the minimum potential vertex is applied on the layered network which is implemented as an adjacency list. If there are T number of threads and L number of layers in a layered network with flow f and n_f vertices then each thread gets $\lceil \frac{L}{T} \rceil$ or $\lfloor \frac{L}{T} \rfloor$ number of layers. However, the number of vertices in a layer is not fixed. Hence, the number of vertices a thread handles during the computation of a probable minimum potential vertex is not fixed. Let us assume that one unit of time is required for each comparison and update operations for computation of the minimum potential vertex. Therefore, if a thread handles maximum n_{max} number of vertices then it takes n_{max} unit time and the time spent by all the threads will be dominated by this time. If the number of threads is equal to the number of processors then the time spent in parallel portion of the method is n_{max} unit. After parallel computation of probable min potential vertices there is a synchronization barrier. After this barrier the master thread computes the minimum potential vertex from T probable minimum potential vertices in T operations. If the number of threads is equal to the number of processors then the total time spent in parallelization of determining minimum potential vertex requires time $TotalMinPot_{CGP} = n_{max} + T + t_{barrier}$. On the other hand, if the computation of minimum potential vertices is done by sequential computation then time spent is $TotalMinPot_{SEQ} = n_f$. Therefore, if the number of threads T is equal to the number of processors p the speedup achieved

by our multithreaded method for determining minimum potential vertex is

$$\begin{aligned} speedup &= \frac{TotalMinPot_{SEQ}}{TotalMinPot_{CGP}} \\ &= \frac{n_f}{n_{max} + T + t_{barrier}} \end{aligned}$$

Now if all the threads have equal number of vertices, *i.e.*, $n_{max} = \frac{n_f}{T}$ the above equation of parallelism changes to

$$\begin{aligned} speedup &= \frac{n_f}{\frac{n_f}{T} + T + t_{barrier}} \\ &= \frac{T}{1 + \frac{T^2}{n_f} + \frac{t_{barrier} * T}{n_f}} \end{aligned}$$

Now, if we consider the situation when $n_f >> T$ and $n_f >> t_{barrier}$, the speedup approaches T or p. Hence, if the number of vertices n_f in the layered network is large and the vertices are divided among threads evenly the theoretical speedup for the computation of the minimum potential vertex is good.

5.3 Analysis of Push-Pull

We use OpenMP loop level (*for* loop) parallelism for push-pull operations. If the minimum potential vertex v_{min} is in layer i, then flow is pushed to vertices in layer $i+1$. Similarly flow is pulled from vertices in layer $i-1$. The vertices from which flow is pulled and to which flow is pushed are stored in an array *pPushPullInf*. Let us assume that there are n_p vertices in the array *pPushPullInf*. Let us also assume that there are T number of threads.

In loop level parallelism, OpenMP scheduler distributes the work load among the processors trying to evenly balance the load. However, the path length from v_{min} to the source, s, might be different from the path length from v_{min} to the sink, t. Hence, the work for push and pull operations might be different. Moreover, for two vertices v_x and v_y of layer $i+1$, to which flow is pushed from the minimum potential vertex v_{min}, the time to complete the push flow might be different as the amount of flow pushed from v_x and v_y might be different and the number of paths might be different. Similarly the time taken by two vertices v_a and v_b of layer $i-1$ might take different amount of time to pull flow from the source, s.

Another problem in analyzing push-pull operation is that vertices are locked during push-pull operations. When a thread pushes flow from a vertex v_x to another vertex v_y, the thread locks both vertices so that no other threads pushes flow from or to vertices v_x and v_y. Similarly a thread locks vertices v_a and v_b when it tries to pull flow to v_a from v_b. When two or more threads try to lock a vertex v, locking contention occurs and one or more vertices have to wait to gain the lock of the vertex.

If there were no locking contentions and the load were equally distributed among the threads, the time taken by a thread would have been $T_{thread} \geq \frac{T_{SeqPushPull}}{T}$ where $T_{SeqPushPull}$ is the time taken by sequential computation of push-pull operations. However, we cannot say anything about the time taken by a thread for locking contention and uneven load distribution for push-pull operations.

Chapter 6

Performance Results

6.1 Experimental Methodology

The experiments were conducted on an 8-processor i686 (Pentium III) Symmetric Multiprocessor (SMP) machine. The processor speed is 700.011 MHZ with a cache size of 1024 KB and a total memory space of 6 GB. The program was written in C using OpenMP library.

We tested the implementation of our algorithm on square mesh type graphs. Square mesh graphs are square grids of vertices. Every vertex in a row has 4 edges to randomly chosen vertices in the next row. The source and sink are external to the grid. The source has edges to all vertices in the top row, and all vertices in the bottom row have edges to the sink. Edge capacities are positive integers drawn randomly and uniformly from the range [1..15]. The experiment was conducted on graphs with 5186, 6402, 7746, 9218,

10818, 12546 and 14402 vertices.

The methodologies followed for the experiment are given below:

- For every instance of graphs three runs were conducted. The runtime are average of these three runs.
- Runs on the same set of inputs were done with number of processors ranging from 1 to 8.
- The experiment was conducted for both static and dynamic scheduling of OpenMP.
- Reported runtime does not include input time.

6.2 Performance Results and Analysis

In this section we describe the performance of the implementation of our multithreaded algorithm. Tables 6.1 and 6.2 give the data about the execution time (second) for 1 to 8 processors with different number of vertices for static and dynamic scheduling respectively. Figures 6.1 and 6.2 show execution time for different number of vertices with 1 to 8 processors for static and dynamic scheduling respectively. The data of Tables 6.3 and 6.4 about the speedup for dynamic and static scheduling is computed from Tables 6.1 and 6.2 respectively. Figures 6.3 and 6.4 show speedup for different number of vertices with 1 to 8 processors for static and dynamic scheduling respectively.

We have achieved speedup of 3.56 to 6.33 on 8 processors. This speedup is comparable to the speedup achieved by Anderson and Setubal [3] and better than the speedup

No. of Threads	No. of Vertices						
	5186	6402	7746	9218	10818	12546	14402
1	103.01	153.08	225.68	561.99	916.42	685.04	1473.36
2	64.02	84.54	119.81	275.47	488.95	402.17	886.60
3	48.09	59.54	94.01	201.46	342.99	292.01	674.74
4	38.05	48.74	66.55	172.91	241.15	231.13	520.90
5	32.58	43.34	60.47	139.84	190.33	184.15	439.88
6	29.76	39.79	54.49	113.45	164.27	146.22	357.74
7	28.42	37.60	51.72	101.50	147.92	130.92	328.48
8	28.91	38.03	50.13	95.53	144.83	126.41	320.24

Table 6.1: Execution time for static scheduling

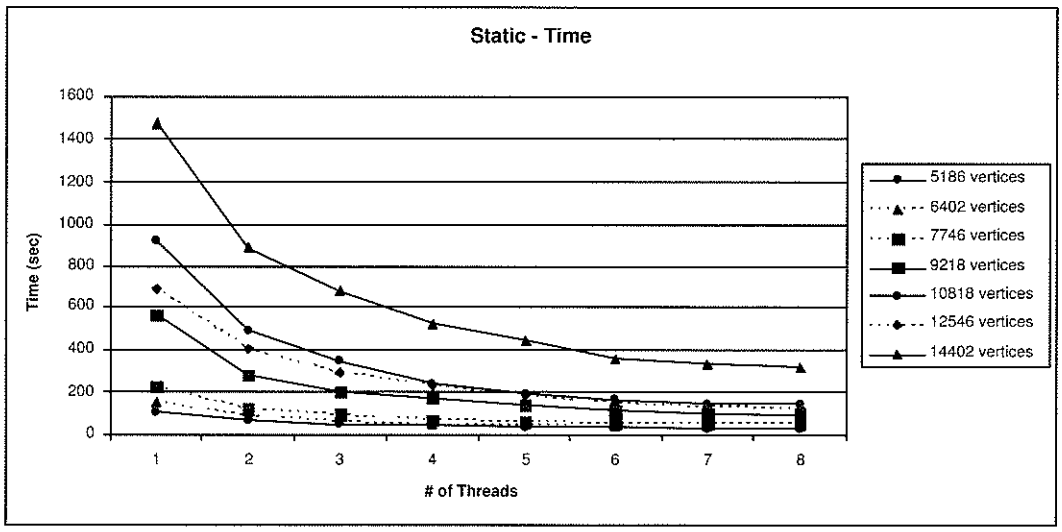

Figure 6.1: Execution time with different number of threads in static scheduling

No. of Threads	No. of Vertices						
	5186	6402	7746	9218	10818	12546	14402
1	102.11	150.61	231.13	554.74	923.01	668.86	1464.72
2	64.32	84.31	124.77	269.17	497.67	407.73	859.29
3	50.85	60.16	93.96	202.55	344.80	282.41	664.85
4	37.72	49.04	72.25	170.04	233.51	231.09	495.40
5	33.45	43.10	60.03	137.42	194.59	176.86	446.19
6	29.70	39.84	53.29	113.30	165.39	145.84	371.32
7	28.00	37.98	50.75	103.97	154.00	129.26	307.69
8	27.55	36.48	48.89	95.53	152.53	123.18	307.45

Table 6.2: Execution time for dynamic scheduling

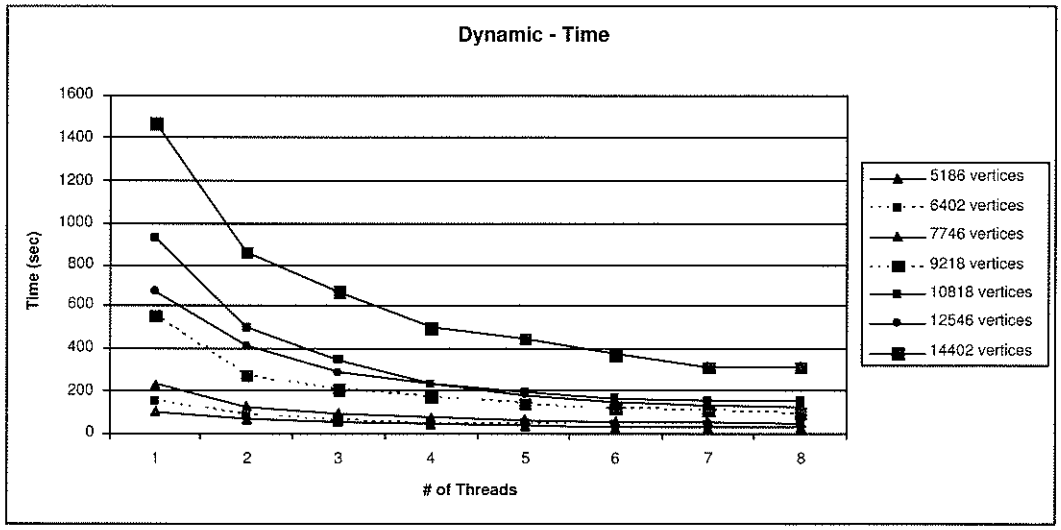

Figure 6.2: Execution time with different number of threads in dynamic scheduling

No. of Threads	No. of Vertices						
	5186	6402	7746	9218	10818	12546	14402
2	1.61	1.81	1.88	2.04	1.87	1.70	1.66
3	2.14	2.57	2.40	2.79	2.67	2.35	2.18
4	2.71	3.14	3.39	3.25	3.80	2.96	2.83
5	3.16	3.53	3.73	4.02	4.81	3.72	3.35
6	3.46	3.85	4.14	4.95	5.58	4.68	4.12
7	3.62	4.07	4.36	5.54	6.20	5.23	4.49
8	3.56	4.03	4.50	5.88	6.33	5.42	4.60

Table 6.3: Speedup for static scheduling

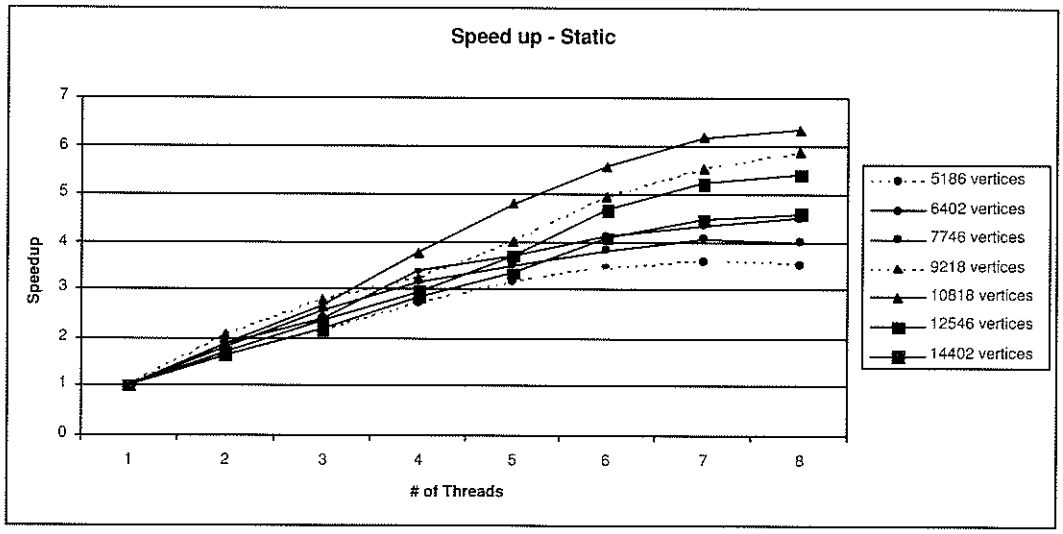

Figure 6.3: Speedup in static scheduling

No. of Threads	No. of Vertices						
	5186	6402	7746	9218	10818	12546	14402
2	1.59	1.79	1.85	2.06	1.85	1.64	1.70
3	2.01	2.50	2.46	2.74	2.68	2.37	2.20
4	2.71	3.07	3.20	3.262	3.95	2.89	2.96
5	3.05	3.49	3.85	4.04	4.74	3.78	3.28
6	3.44	3.78	4.34	4.90	5.58	4.59	3.94
7	3.65	3.97	4.55	5.34	5.99	5.17	4.76
8	3.71	4.13	4.73	5.81	6.05	5.43	4.76

Table 6.4: Speedup for dynamic scheduling

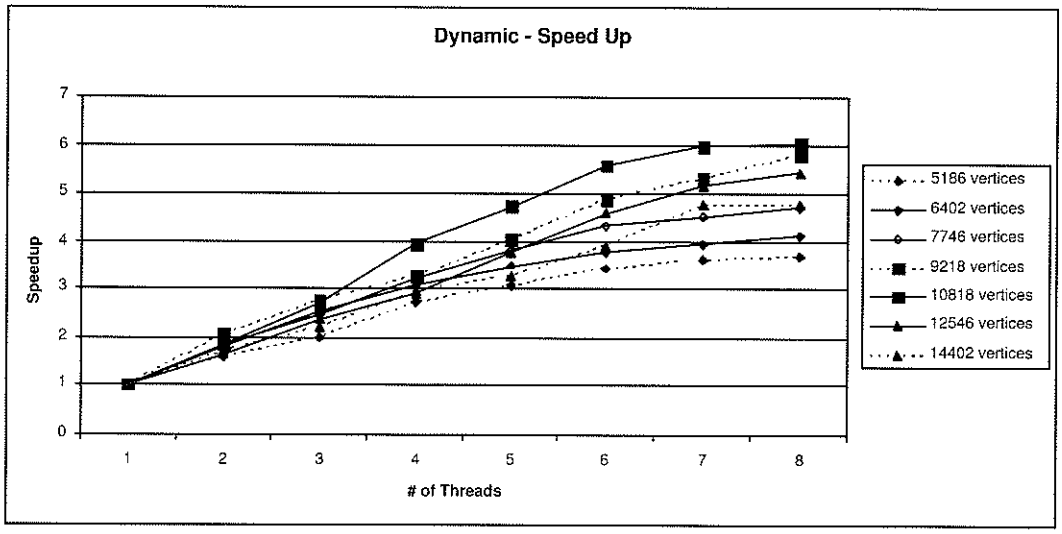

Figure 6.4: Speedup in dynamic scheduling

achieved by Lie [35]. Anderson and Setubal obtained a speedup of 5 to 6 on 8 processors whereas Lie obtained a speedup of 1.5 on 6 processors.

It can be found from Figures 6.3 and 6.4 that the speedup is not linear. Lack of parallelism, synchronization, hardware effects and lock contention are the major factors behind the nonlinear speedup. Here, we will discuss these factors concisely.

Lack of parallelism occurs when the total number of vertices to which flow can be pushed from or pulled to the least potential vertex is less than the number of processors. In our algorithm a layered network is created in every phase. Then push-pull and deletion operations are applied on the layered network. The number of vertices and edges decreases with increasing number of iterations within a phase which results in a lack of parallelism.

Synchronization overhead happens during synchronization of different threads in labeling of the vertices and calculation of the minimum potential vertex and after every push-pull operations.

Hardware overheads are due to bus contention, cache flushes due to false sharing, and memory hierarchy effects common to a shared-memory machine. Bus contention results from the attempt of gaining the access to the same bus by two or three processors while cache flushes due to false sharing occurs for sharing a cache line by two or more processors.

Vertices are locked during push-pull operations so that two different vertices from layer i cannot push flow to the same vertex in layer $i+1$ or two different vertices from

layer i cannot pull flow from the same vertex of layer $i-1$ at the same time. Locking contention occurs when two or more threads try to lock the same vertex.

It can be noticed from Tables 6.3 and 6.4 that the maximum speedup achieved for static scheduling is 6.33 while maximum speedup achieved for dynamic scheduling is 6.05. The reason behind this difference in speedup might be that in case of static scheduling the scheduler schedules during compilation of the program while in case of dynamic scheduling the scheduler schedules during runtime. The difference in the scheduling policy results in different speedup.

It can also be noticed from Tables 6.3 and 6.4 and Figures 6.3 and 6.4 that speedup increases with increase in number of vertices from 5186 to 10818 and and speedup decreases with increase in number of vertices from 10818 to 14402. The increase in speedup for increase in data size can be inferred from the fact that the ratio of time spent in execution to the time loss in overhead is bigger for larger data size compared to lower data size. However, with increase in data size speedup decreases after a certain point on data size for cache misses for larger data size.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The goals of this thesis work were to 1) design and implement a multithreaded maximum flow algorithm in OpenMP and 2) compare the performance of the algorithm with Lie's implementation of Goldberg and Tarjan's maximum flow algorithm in OpenMP.

We have designed and implemented a multithreaded maximum flow algorithm using Malhotra, Pramodh Kumar and Maheswari's maximum flow algorithm (MPM algorithm) [36]. We have done extensive experiments with the implementation of our algorithm for square mesh graphs.

In this thesis we have developed a multithreded approach for labeling vertices. We have devised a technique to push and pull flow along several paths simultaneously.

We have achieved speedup of 3.56 to 6.33 on 8 processors whereas Lie's implementa-

tion has a maximum speedup of 1.5 on 6 processors. We would like to mention that Lie's implementation of Goldberg and Tarjan's maximum flow algorithm in OpenMP is the only multithreaded implementation of a maximum flow algorithm on a shared memory machine before our work. Lie did not get a good speedup. However, our multithreaded implementation of MPM algorithm has achieved a good speedup.

One of our major achievements in this thesis is obtaining a maximum speedup of 6.33 in OpenMP on 8 processor shared memory machine which can be considered as a good speedup for an eight processor shared memory machine on OpenMP.

7.2 Future Work

One future direction of this work can be min cost flow algorithm which is a natural extension of maximum flow algorithm. Min cost flow problem considers edge capacities as well as cost of an edge. Min cost flow problem is a generalization of shortest path and maximum flow problem. Shortest path problem considers only cost of edges and maximum flow problem considers only edge capacities. Min cost flow problem can be solved using primal-dual approach. In this approach min cost flow problem is solved in an iterative manner. In primal-dual approach each iteration solves a shortest path problem with nonnegative arc lengths and a maximum flow problem. In this approach flow is augmented along all shortest paths simultaneously. In this thesis we have developed a multithreaded maximum flow algorithm and the calculation of shortest path is part of our algorithm. Hence, our work can be extended to the design and implementation of a

multithreaded algorithm for the min cost flow problem.

Bibliography

[1] P. Agarwal and A. Ng. Computing network flow on a multiple processor pipeline. *IEEE Transactions on Parallel and Distributed Systems*, 5(6):653–658, 1994.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, Princeton, New Jersey, 1993.

[3] R. J. Anderson and J. C. Setubal. On the parallel implementation of Goldberg's maximum flow algorithm. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 168–177, San Diego, CA, July 1992.

[4] Arvind and R. A. Iannucci. Two fundamental issues in multiprocessing. In *Proceedings of DFVLR Parallel Processing on Science and Engineering*, pages 61–88, West Germany, July 1987.

[5] V. K. Balakrishnan. *Network Optimization.* Chapman & Hall, 1995.

[6] R. D. Blumofe, C. E. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pages 207–216, Santa Barbara, CA, July 1995.

[7] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 704–709, Cancún, Mexico, April 1994.

[8] S. M. Brunett, J. Thornley, and M. Ellenbecker. An initial evaluation of the Tera multithreaded architecture and programming system using the C3I parallel benchmark suite. In *Supercomputing '98*, pages 1–19, Orlando, Florida, November 1998.

[9] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Academic Press, San Diego, CA, USA, 2001.

[10] K. M. Chandy and J. Misra. Distributed computation on graphs: Shortest path algorithm. *Communications of the ACM*, 25(11):833–837, 1982.

[11] J. Cheriyan, T. Hagerup, and K. Mehlhorn. An $o(n^3)$-time algorithm maximun-flow algorithm. *SIAM Journal on Computing*, 25:1144–1170, 1996.

[12] V. Chvatal. *Linear Programming*. Freeman Co., Maryland, USA, 1990.

[13] G. B. Dantzig and D. R. Fulkerson. Minimizing the number of tankers to meet a fixed schedule. *Naval Research Logistics Quarterly*, 1:217–222, 1954.

[14] E. Dijkstra. A note on two problems in connexion with graphs. *Numeriche Mathematics*, 1:269–271, 1959.

[15] E. A. Dinic. Algorithm for the solution of a problem of maximum flow in a network with power estimation. *Soviet Math. Dokl.*, 11:1277–1280, 1970.

[16] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

[17] G. W. Flake, S. Lawrence, and C. L. Giles. Efficient identification of web communities. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 150–160, Boston, Massachusetts, August 2000.

[18] G. W. Flake, S. Lawrence, C. L. Giles, and F. Coetzee. Self-organization of the web and identification of communities. *IEEE Computer*, 35(3):66–71, 2002.

[19] L. R. Ford and D. R. Fulkerson. Maximum flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[20] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University, Princeton, New Jersey, 1962.

[21] V. Getov and M. Philippsen. Java communications for large-scale parallel computing. In J. Waśniewski S. Margenov and P. Yalamov, editors, *Proceedings of Third International Conference on Large-Scale Scientific Computations*, pages 33–45, Sozopol, Bulgaria, June 2001. Springer Verlag, Lecture Notes in Computer Science 2179.

[22] A. V. Goldberg. Recent developments in maximum flow algorithms. Technical Report 98-045, NEC Research Institute, April 1998.

[23] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, September 1998.

[24] A. V. Goldberg and R. E. Tarjan. A new approach to maximum flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[25] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing, Second Edition.* Addision-Wesley, Harlow, England, 2003.

[26] J. L. Hennesey and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Third Edition.* Morgan Kaufmann, San Francisco, CA, 2003.

[27] H. H. J. Hum, K. B. Theobald, and G. R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 288–294, Cancún, Mexico, April 1994.

[28] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, New York, NY, 1993.

[29] K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture and Programming.* McGraw Hill, 1998.

[30] N. Imafuji and M. Kitsuregawa. Effects of maximum flow algorithm on identifying web community. In *Proceedings of the Fourth International Workshop on Web Information and Data Management*, pages 43–48, McLean, Virginia, November 2002.

[31] C. Joerg and B. Kuszmaul. Massively parallel chess. In *Third DIMACS Parallel Implementation Challenge Workshop*, Rutgers University, October 1994.

[32] L. V. Kale and S. Krishnan. Charm++: Parallel programming with message-driven objects. In *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[33] V. King, S. Rao, and R. E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, 1994.

[34] B. Lewis and D. J. Berg. *Threads Primer - A Guide to Multithreaded Programming.* SunSoft Press, Prentice Hall, Upper Saddle River, NJ, USA, 1996.

[35] S. Lie. Parallel programming interfaces. Technical report, MIT, 2003.

[36] V. M. Malhotra, M. P. Kumar, and S. N. Maheshwari. An $O(\mid V \mid^3)$ algorithm for finding maximum flows in networks. *Information Processing Letters*, 7(6):277–278, October 1978.

[37] C. Martel. Preemptive scheduling with release times, deadlines, and due times. *Journal of the ACM*, 29(3):812–829, 1982.

[38] N. Nagy and S. G. Akl. The maximum flow problem: A real-time approach. *Parallel Computing*, 29(6):767–794, 2003.

[39] L. Oliker and R. Biswas. Parallelization of a dynamic unstructured application using three leading paradigms. *IEEE Transactions on Parallel and Distributed Computing*, 11(9):931–940, September 2000.

[40] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface*, March 2002. http://www.openmp.org/specs/mp-documents/cspec20.pdf.

[41] R. C. Paige and C. P. Kruskal. Parallel algorithms for shortest path problems. In *International Conference on Parallel Processing (ICPP '85)*, pages 14–20, University Park, PA, USA, August 1985.

[42] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications Inc., Mineola, New York, 1982.

[43] A. Pinar and B. Hendrickson. Exploiting flexibly assignable work to improve load balance. In *Proceedings of the Fourteeth Annual ACM Symposium on Parallel Algorithms and Achitectures*, pages 155–163, Winnipeg, MB, August 2002.

[44] S. Roy and I. J. Cox. A maximum-flow formulation of the n-camera stereo correspondence problem. In *Sixth International Conference on Computer Vision*, pages 492–502, January 1998.

[45] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.3.2 Reference Manual*, November 2001. url: http://supertech.lcs.mit.edu/cilk/manual-5.3.2.pdf.

[46] K. B. Theobald, J. N. Amaral, G. Herber, O. Maquelin, X. Tang, and G. R. Gao. Overview of the threaded-c language. Technical Memo CAPSL Technical Memo 19, Computer Architecture and Parellel Systems Laboratory (CAPSL), University of Delware, Newark, Delaware, March 1998.

[47] K. B. Theobald, R. Kumar, G. Agarwal, G. Heber, R. Thulasiram, and G. R. Gao. Implementation and evaluation of a communication intensive application on the EARTH multithreaded system. *Concurrency and Computation: Practice and Experience*, 14:183–201, March 2002.

[48] I. Thomo, S. Malassiotis, and M. G. Strintzis. Optimized block based disparity estimation in stereo systems using a maximum-flow approach. In *Proceedings of the XI SIBGRAPI'98 International Conference*, pages 410–417, Rio de Janeiro, RJ, Brazil, October 1998.

[49] K. Thulasiraman and M. N. S. Swamy. *Graphs: Theory and Algorithms.* John Wiley & Sons, Inc., New York, 1992.

[50] P. Thulasiraman. A distributed protocol for the network primal-dual method and simulation on a shared-memory multiprocessor. Master's thesis, Department of Computer Engineering, Concordia University, November 1991.

[51] P. Thulasiraman. *Irregular Computations on Fine-Grained Multithreaded Architectures.* PhD thesis, University of Delaware, August 2000.

[52] P. Thulasiraman and A. A. Khokhar. An asynchronous multithreaded algorithm for a class of linear programming problems. In *15th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS 2002)*, The Galt House, Louisville, KY, USA, September 2002.

[53] P. Thulasiraman, Xin-Min Tian, and G. R. Gao. Multithreading implementation of a distributed shortest path algorithm on earth multiprocessor. In *3rd International Conference on High Performance Computing*, pages 336–341, Trivandrum, India, December 1996.

[54] J. L. Träff. Distributed, synchronized implementation of an algorithm for the maximum flow problem. In *Proceedings of the 23rd International Conference on Parallel Processing (ICPP'94)*, pages 110–114, St. Charles, IL, Aug 1994.

[55] K. D. Wayne. *Generalized Maximum Flow Algorithms.* PhD thesis, Cornell University, 1999.