

A Project Report on

“PARALLELIZING NETWORK FLOW ALGORITHM USING PUSH-RELABEL METHOD”

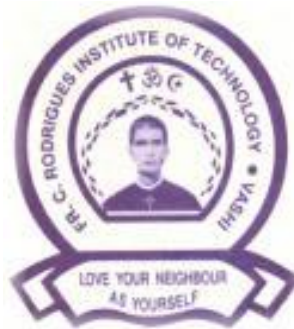
**Submitted in partial fulfillment of the requirement for
Degree in Bachelor of Engineering (Computer Engineering)**

By

**ABHIJIT BHANDARKAR
ADITYA GAYKAR
NIVEDITA SHARMA**

Guided by

Mr. Amroz S



**Department of Computer Engineering
Fr. Conceicao Rodrigues Institute of Technology
Sector 9A, Vashi, Navi Mumbai – 400703**

**University of Mumbai
2012-2013**

CERTIFICATE

This is to certify that the project entitled

PARALLELIZING NETWORK FLOW ALGORITHM USING PUSH-RELABEL METHOD

Submitted By

Abhijit G. Bhandarkar (100906)
Aditya Gaykar (100919)
Nivedita Sharma (100952)

In partial fulfillment of degree of **B.E (Sem- VIII)** in **Computer Engineering** for term work of the project is approved.

External Examiner

Internal Examiner

External Guide

Internal Guide

Head of the Department

Principal

Date: -

College Seal

ACKNOWLEDGEMENT

We are grateful to our principal **Dr. Rollin Fernandes** and to our Head of the Department **Prof. Harish Kumar Kaura** for giving us the time and resources for our project.

We would like to thank our internal guide **Mr. Amroz S** for giving us invaluable advice, inputs and time and providing endless moral support whenever we required his help.

We would also like to thank the entire Computer Department for their support and guidance.

Last but not the least, we acknowledge with a deep sense of gratitude all the researchers who have done pioneering work in the field of Network Flow and Parallelization of algorithms and all the authors whose papers have been taken as a reference and as a guidance material for our report.

ABSTRACT

Our project addresses the maximum-flow problem, where in, we compute the greatest rate at which we can ship material from the source to sink without violating any capacity constraints or flow constraints. It is among the common problems concerning flow networks.

Our project tries to solve this problem using efficient maximum-flow algorithm. These maximum flow algorithms are linear algorithms which can be implemented on a single machine. In our project, we plan to parallelize the network flow algorithm over a cluster consisting of 4 slave nodes and 1 master node, using Message Passing Interface (MPI) library. There are various algorithms such as Ford-Fulkerson algorithm, Edmonds-Karp algorithm, Goldberg's "generic" maximum flow algorithm. Our project implements Push Relabel method that runs in a time $O(V^3)$. Moreover, the entire algorithm is implemented in a parallel environment.

INDEX

Sr. No.	Topic	Page No.
1	Introduction	1
2	Literature Survey/Analysis	4
3	System Design	13
4	H/W and S/W requirements	19
5	Implementation 5.1 Implementation details 5.2 Implementation Issues 5.3 Screen Shots	21 25 26
6	Testing	27
7	Future Scope	29
8	Conclusion	30
9	References	31
10	Appendix	32

LIST OF FIGURES

Sr. No.	Name of the Figure	Page No.
1	Flow network diagram	1
2	Minimum cut in flow network	2
3	Conversion of maximum flow to minimum flow problem	11
4	Push Relabel Algorithm	13
5	Push Operation	14
6	Relabel operation	15
7	Sample network flow graph	17
8	Time line chart for processes	18
9	Input graph file	21
10	Output file format	22
11	Commands for compiling the parallel code	23
12	Command for executing parallel code	23
13	Cluster setup in R & D lab	26
14	Serial vs. parallel execution time	28
15	Serial vs. parallel CPU utilization	28

LIST OF TABLES

Sr. No.	Name of the Table	Page No.
1	Serial and parallel code execution time comparison	27
2	Serial and parallel code CPU utilization comparison	27

1. INTRODUCTION

1.1 Maximum flow Problem

In the maximum flow problem, we are given a directed or undirected graph, most commonly directed in real world applications, where one vertex is considered a source and another is the destination commonly referred to as the sink. Some object then flows along the edges of the graph from the source to the sink. Each edge along the path is given a maximum capacity that can be transported along that route. The maximum capacity can vary from edge to edge in which case the remainder must either flow along another edge towards the sink or remain at the current vertex for the edge to clear or be reduced. The goal of the maximum flow problem is to determine the maximum amount of throughput in the graph from the source to sink. In real world applications determining the maximum throughput allows the source to know exactly how much of something to produce and send along the path without creating waste.

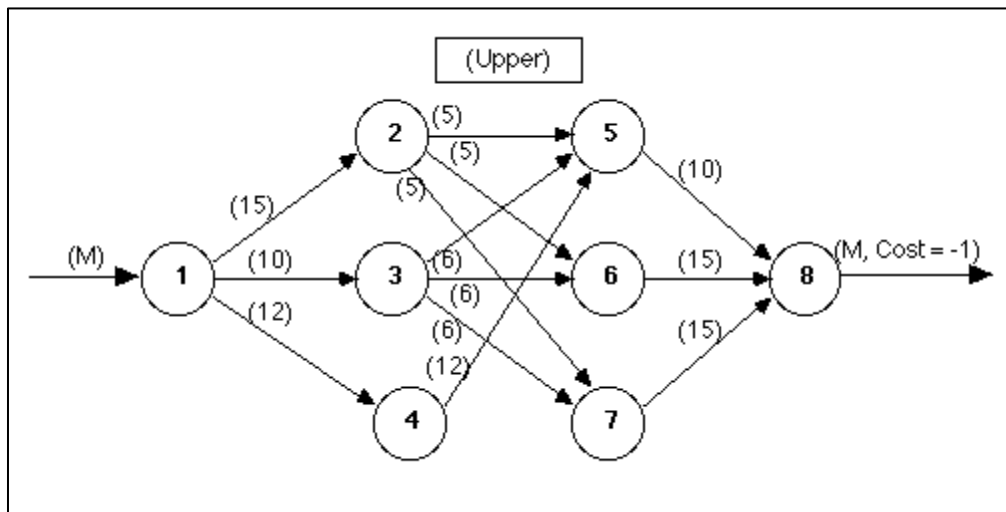


Fig. 1: Flow network diagram

The solution to the example shown in figure 1 is illustrated in the flow shown in figure 2. The maximum flow from node 1 to node 8 is 30 and the flows that yield this flow are shown on the figure 2. The heavy arcs on the figure 2 are called the minimal cut. These arcs are the bottlenecks that are restricting the maximum flow. The fact that the sum of the capacities of the arcs on the minimal cut equals the maximum flow is a famous theorem of network theory called the max flow min cut theorem. The arcs on the minimum cut can be identified using sensitivity analysis.

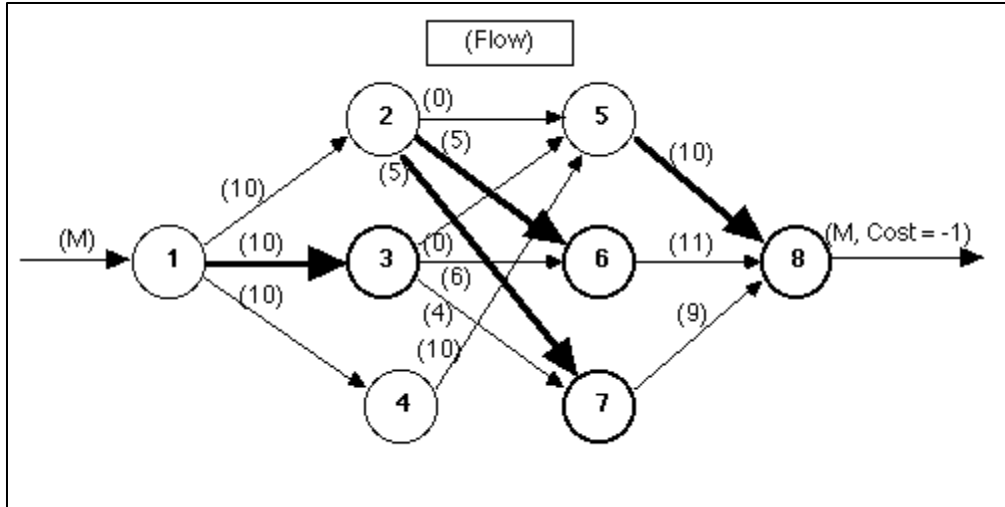


Fig. 2: Minimum cut in flow network

1.2 Flow networks and flows

In graph theory, a flow network (also known as a transportation network) is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge. Often in Operations Research, a directed graph is called a network, the vertices are called nodes and the edges are called arcs. A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, except when it is a source, which has more outgoing flow, or sink, which has more incoming flow.

$G(V, E)$ is a finite directed graph in which every edge $(u, v) \in E$ has a nonnegative, real-valued capacity $c(u, v)$. If $(u, v) \notin E$, we assume that $c(u, v) = 0$. We distinguish two vertices: a source s and a sink t . A flow network is a real function $f : V \times V \rightarrow \mathbb{R}$ with the following two properties for all nodes u and v :

Capacity constraints:	$0 \leq f(u, v) \leq c(u, v)^*$
Flow conservation:	$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$ *

1.3 Parallel Processing

It is simultaneous use of more than one CPU to execute a program. Ideally, parallel processing makes a program run faster because there are more engines (CPUs) running it. In practice, it is often difficult to divide a program in such a way that separate CPUs can execute different portions without interfering with each other. Most computers have just one CPU, but some models have several. There are even computers with thousands of CPUs. With single-CPU

computers, it is possible to perform parallel processing by connecting the computers in a network. However, this type of parallel processing requires very sophisticated software called distributed processing software. Parallel processing differs from multitasking, in which a single CPU executes several programs at once. Parallel processing is also called parallel computing.

2. LITERATURE SURVEY

2.1 Parallel Computing

2.1.1 What is parallel computing

Traditionally software had to be written for the serial computation i.e. to be run on the single computer having single central processing unit (CPU) and where only one instruction may execute at any moment of time.

Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem. In this a problem is broken into discrete parts that can be solved concurrently and instructions from each part execute simultaneously on different CPUs. Converting a sequential program to a parallel program is called Parallelization.

The compute resource might be,

- A single computer with multiple processor;
- An arbitrary number of computers connected in network;
- A combination of both.

The computational problem should be able to,

- Be broken apart into discrete pieces of work that can be solved simultaneously;
- Execute multiple program instructions at any moment in time;
- Be solved in less time with multiple compute resources than with a single compute resource.

2.1.2 Important components of a Parallel Computer

- **Compute Node**
 1. Processor (e.g. Core i7, Xeon, Athlon, Opteron)
 2. CPU Board (e.g. Intel/Supermicro motherboard)
 3. Full-fledged sequential computer (eg. PC, workstation, rackmount server)
- **Interconnection Network**
 1. Bus (eg. Multibus, VME)
 2. Switch (eg. Crossbar, Omega)
 3. Static network of different topologies

2.1.3 Need of Parallel Computers

1. Save time and/or money : The completion time of a task can be shorted by providing it with more resources which can also lead to potential cost saving. Parallel computers can be built from cheap, commodity components.

2. Solve larger problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory. For example: Web search engines/databases processing millions of transactions per second.

3. Provide concurrency: Multiple computing resources can be doing many things simultaneously, as compared to single compute resource which can only do one thing at a time.

4. Limits to serial computing : Serial computers possess many limitations such as,

- The transmission speed in serial computers cannot be increased from a certain limit. Increasing speeds necessitate increasing proximity of processing elements.
- To come up with more processing power there is need to increase number of transistors placed on the chip but even with the molecular or atomic level a limit will reach for how small a component can be.
- It is increasingly expensive to make a single processor faster. While making the use of moderately fast processor in parallel computing to achieve same performance is less expensive.

2.1.4 Classifications of Parallel Computers

There are different ways to classify parallel computers but one of the most widely used classification is Flynn's Taxonomy. Flynn classified programs and computers by whether they were operating using a single set or multiple sets of instructions, whether or not those instructions were using a single or multiple sets of data.

1. Single Instruction, Single Data (SISD) : It is serial computer where only one instruction stream is acted upon by the CPU and only one data stream is being used as input during any one clock cycle

2. Single Instruction, Multiple Data (SIMD) : It is a parallel computer where all processing unit operates on the data independently via separate instruction streams and a units execute the same instruction at any given clock cycle and each processing unit can operate on a different data element.

3. Multiple Instruction, Single Data (MISD) : It is a type of parallel computer where each single data stream is fed into multiple processing units.

4. Multiple Instruction, Multiple Data (MIMD) : It is a type of parallel computer where every processor may be executing a different instruction stream and may be working with a different data stream.

2.1.5 Designing Parallel Programs

1. Understand the Problem and the Program

- The very first step in developing a parallel software is to first understand the problem that is to be solved in parallel. Determine whether or not the problem is one that can actually be parallelized. e.g.
 1. Performing addition of elements of two arrays containing large number of elements and storing them in third array.

for i = 1, 1000 do:

c[i] = a[i] + b[i]

2. Calculation of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula:

$$F(n) = F(n-1) + F(n-2).$$

- Identify the part of the program where most of the real work is done. Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.
- Identify inhibitors to parallelism. One common class of inhibitor is data dependence, as demonstrated by the Fibonacci sequence above.

2. Partitioning

- In this step problem is broken into discrete "chunks" of work that can be distributed to multiple tasks. There are two basic ways to partition computational work among parallel tasks
 1. **Domain Decomposition:** In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.
 2. **Functional Decomposition:** In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.

3. Communications

- Knowledge of which tasks must communicate with each other is critical during the design stage of a parallel code. Two types of communications are possible which can be implemented synchronously or asynchronously.

1. **Point-to-point:** involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
2. **Collective:** involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective. Some common variants are broadcast, scatter, gather, reduction.

4. Synchronization

- Synchronization in parallel programs can be implemented in following ways,
 1. **Barrier:** Each task performs its work until it reaches the barrier. It then stops or blocks. When the last task reaches the barrier, all tasks are synchronized.
 2. **Lock / semaphore:** Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
 3. **Synchronous communication operations:** When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication. For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.

5. Data Dependencies

- A dependency exists between program statements when the order of statement execution affects the results of the program.
- A data dependence results from multiple use of the same location(s) in storage by different tasks.
- Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism.
- To handle the dependencies we can use,
 1. Distributed memory architectures to communicate the required data at synchronization points.
 2. Shared memory architecture to synchronize read/write operations between the tasks.

6. Input / Output

- I/O operations are generally regarded as inhibitors to parallelism.
- In an environment where all tasks see the same file space, write operations can result in file overwriting.

- Read operations can be affected by the file server's ability to handle multiple read requests at the same time.
- I/O that must be conducted over the network (NFS, non-local) can cause severe bottlenecks and even crash file servers.
- The solution is to use **Parallel File System**
 1. GPFS: General Parallel File System for AIX (IBM)
 2. Lustre: for Linux clusters (Oracle)

7. Performance Analysis and Tuning

- As with debugging, monitoring and analyzing parallel program execution is significantly more of a challenge than for serial programs.
- A number of parallel tools for execution monitoring and program analysis are available.
 1. Hardware Performance Monitor(HPM) Toolkit
 2. Dynamic Probe Class Library (DPCL)

2.1.6 Different approaches to parallel program development

1. **Parallel Languages** : LINDA, OCCAM etc.
2. **Parallel Extension to serial languages** : High Performance Fortran
3. **Parallel APIs** : OpenMP, MPI(Message Passing Interface)

2.2 Maximum Flow Problem

In the maximum flow problem, we wish to compute the greatest rate at which material can be shipped from the source to the sink without violating any capacity constraints [1]. We are given a directed or undirected graph, most commonly directed in real world applications, where one vertex is considered a source and another is the destination or commonly referred to as the sink. Some object then flows along the edges of the graph from the source to the sink. Each edge along the path is given a maximum capacity that can be transported along that route. The maximum capacity can vary from edge to edge in which case the remainder must either flow along another edge towards the sink or remain at the current vertex for the edge to clear or be reduced. The goal of the maximum flow problem is to determine the maximum amount of throughput in the graph from the source to sink. In real world applications determining the maximum throughput allows the source to know exactly how much of something to produce and send along the path without creating waste [8].

2.2.1 Methods to solve Maximum flow Problem

Let $G = (V, E)$ is a directed graph with $|V| = n$ vertices and $|E| = m$ edges and with two distinguished vertices, the source vertex s and the sink vertex t . Each edge has a positive real-valued capacity function c , and there is a flow function f defined over every vertex pair. The flow function must satisfy three constraints:

1. $f(u, v) \leq c(u, v)$ for all u, v in $V \times V$ (Capacity constraint)
2. $f(u, v) = -f(v, u)$ for all u, v in $V \times V$ (Skew symmetry)
3. $\sum_{v \in V} f(u, v) = 0$ for all u in $V - \{s, t\}$ (Flow conservation)

The flow of the network is the net flow entering the sink vertex t (which is equal to the net flow leaving the source vertex s).

In mathematical terms,

$$|f| = \sum_{u \in V} f(u, t) = \sum_{v \in V} f(s, v).$$

The maximum flow problem (MAX-FLOW) is to determine the maximum possible value for $|f|$ and the corresponding flow values for each vertex pair in the graph [9].

2.2.1.1 Related Concepts

1. Residual Network

The residual capacity of an edge is $c_f(u, v) = c(u, v) - f(u, v)$. This defines a residual network denoted $G_f(V, E_f)$, giving the amount of available capacity. There can be a path from u to v in the residual network, even though there is no path from u to v in the original network. Since flows in opposite directions cancel out, decreasing the flow from v to u is the same as increasing the flow from u to v .

2. Augmenting Paths

An augmenting path is a path (u_1, u_2, \dots, u_k) in the residual network, where $u_1 = s$, $u_k = t$ and $c_f(u_i, u_{i+1}) > 0$. A network is at maximum flow if and only if there is no augmenting path in the residual network.

3. Cuts of flow networks

A cut (S, T) on the flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$.

2.2.1.2 Ford-Fulkerson Method (G, s, t)

1. Initialize flow f to 0
2. **while** there exists an augmenting path p
3. **do augment** flow f along p
4. **return** f

The basic algorithm:

FORD-FULKERSON(G, s, t)

1. **for** each edge $(u, v) \in E[G]$
2. **do** $f[u, v] \leftarrow 0$
3. $f[v, u] \leftarrow 0$
4. **while** there exists a path p from s to t in the residual network G_f
5. **do** $c_f(p) \leftarrow \min \{c_f(u, v) : (u, v) \text{ is in } p\}$
6. **for** each edge (u, v) in p
7. **do** $f[u, v] \leftarrow f[u, v] + c_f(p)$
8. $f[v, u] \leftarrow -f[u, v]$

Analysis:

This method finds the augmenting path using Depth first search. Time complexity is $O(|E|f^*)$, where f^* is the maximum flow found by the algorithm. The analysis is as follows,

Lines 1-3 take time $O(E)$. The while loop of lines 4-8 is executed at most $|f^*|$ times, since the flow value increases by at least one unit in each iteration. The Edmonds-Karp algorithm is a specification of the Ford-Fulkerson algorithm. It finds the augmented path using Breadth first search. Time complexity of Edmonds-Karp algorithm is $O(VE^2)$ [10].

2.2.1.3 Push Relabel Method

Consider a flow network $G = (V, E)$ where $n = |V|$, $m = |E|$. The Push-Relabel algorithm explained here is Goldberg's generic maximum-flow algorithm, which runs in $O(n^2m)$ time, an improvement on the $O(nm^2)$ time of the Edmonds-Karp algorithm [6].

Instead of maintaining the flow conservation property, we maintain a preflow f that is anti-symmetric and satisfies the capacity constraints, and allow excess flow $e(u) = f(V, u)$ at the vertices. We maintain a height function to control how flow is pushed through the network: Height $h(u) \leq h(v) + 1$ if $(u, v) \in E_p$, where E_p is the residual network. The height of the source s is fixed at $|V|$ and the height of the sink is fixed at 0. All other vertices start with height 0, and are increased as the algorithm progresses, and flow is only allowed from a higher vertex to a lower vertex.

If there is an edge from u to v in the residual network, then we are going to push excess flow from u to v if the height difference is exactly 1, subject to capacity constraints. That is, v is lower than u , but just by 1.

The idea is that we are draining the excess flow from the source, and at the time there is no excess flow, it is a flow.

- **Lemma 1.** For any u, v in V , if $h(u) < h(v)+1$ then (u, v) is not in E_f . This follows directly from the height function [7].
- **Lemma 2.** After a non-saturating push from u to v $e(u) = 0$. This is obvious from the Push operation. Whatever excess flow we had on u , we have let it go.
- **Lemma 3.** An overflowing vertex can either be pushed or relabeled. Proof. If Push is not applicable, $h(u) < h(v) + 1 \Rightarrow h(u) \leq h(v)$ so we can Relabel, since f is a preflow.
- **Lemma 4.** Vertex heights never decrease. Proof. Relabel only increases the height, and push doesn't change it. By the preconditions on u in Relabel, $h(u) \geq h(v) \forall v.s.t. (u, v) \in E_f$. Thus, Relabel increases the height of u by at least 1.

2.2.1.4 ϵ - Relaxation Method

ϵ -Relaxation algorithm is an auction method basically designed to solve the minimum cost flow problem [2]. The algorithm can also be applied to the maximum flow problem with some modifications in its actual implementation. A maximum flow problem can be viewed as minimum cost flow problem once we introduce a feedback arc connecting the sink with the source and having cost as -1 which represents the source price as 1 and sink price as 0, as shown in the figure 3,

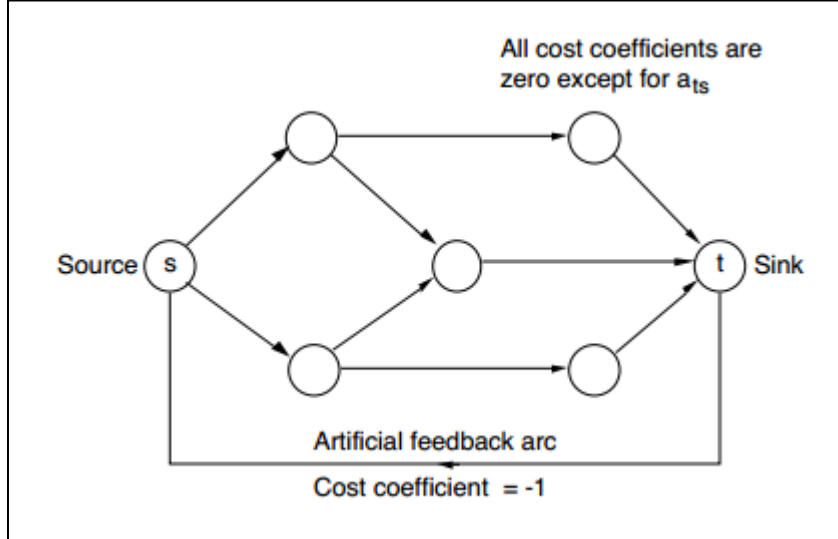


Fig. 3: Conversion of maximum flow to minimum flow problem

with the flow bounds as minimum flow $b_{ij} = 0$ and maximum flow as $c_{is} = \sum_{i \in N} c_{si}$. Also the cost associated with each node i.e. a_{ij} is set to zero.

In practice, the ϵ - relaxation method initialized with zero flow and zero price vectors often finds a minimum cut very quickly. It may then work quite a bit more to set the remaining positive surpluses to zero. Thus, if one is interested in just a minimum cut or just the value of the maximum flow, it is worth testing periodically to see whether a minimum cut can be determined before the algorithm terminates. Given a minimum cut, one may find a maximum flow by continuing the algorithm until all node surpluses are zero, or by employing a version of the Ford-Fulkerson algorithm to return the positive surpluses to the source.

In practice one can also typically forego ϵ -scaling, but some additional computational tricks are needed to nullify the effects of price wars. If ϵ -scaling is not used, one can change the cost of the feedback arc (t, s) to at , $\epsilon = -(N + 1)$ and use $\epsilon = 1$ throughout. When this is done, the ϵ -relaxation method bears a close resemblance with a max-flow algorithm called as “push relabel algorithm”. [3]

Complexity analysis

The ϵ -relaxation method, with the use of some fairly simple data structures, but without the use of scaling, can be shown to have a worst case running time of $O(N^3 + N^2 L/\epsilon)$ where L is the maximum over the lengths of all simple paths, with the length of each arc (i, j) being the absolute reduced cost value $|p_j + a_{ij} - p_i|$, and p being the initial price vector. [4]

3. DESIGN

3.1 Push Relabel Method

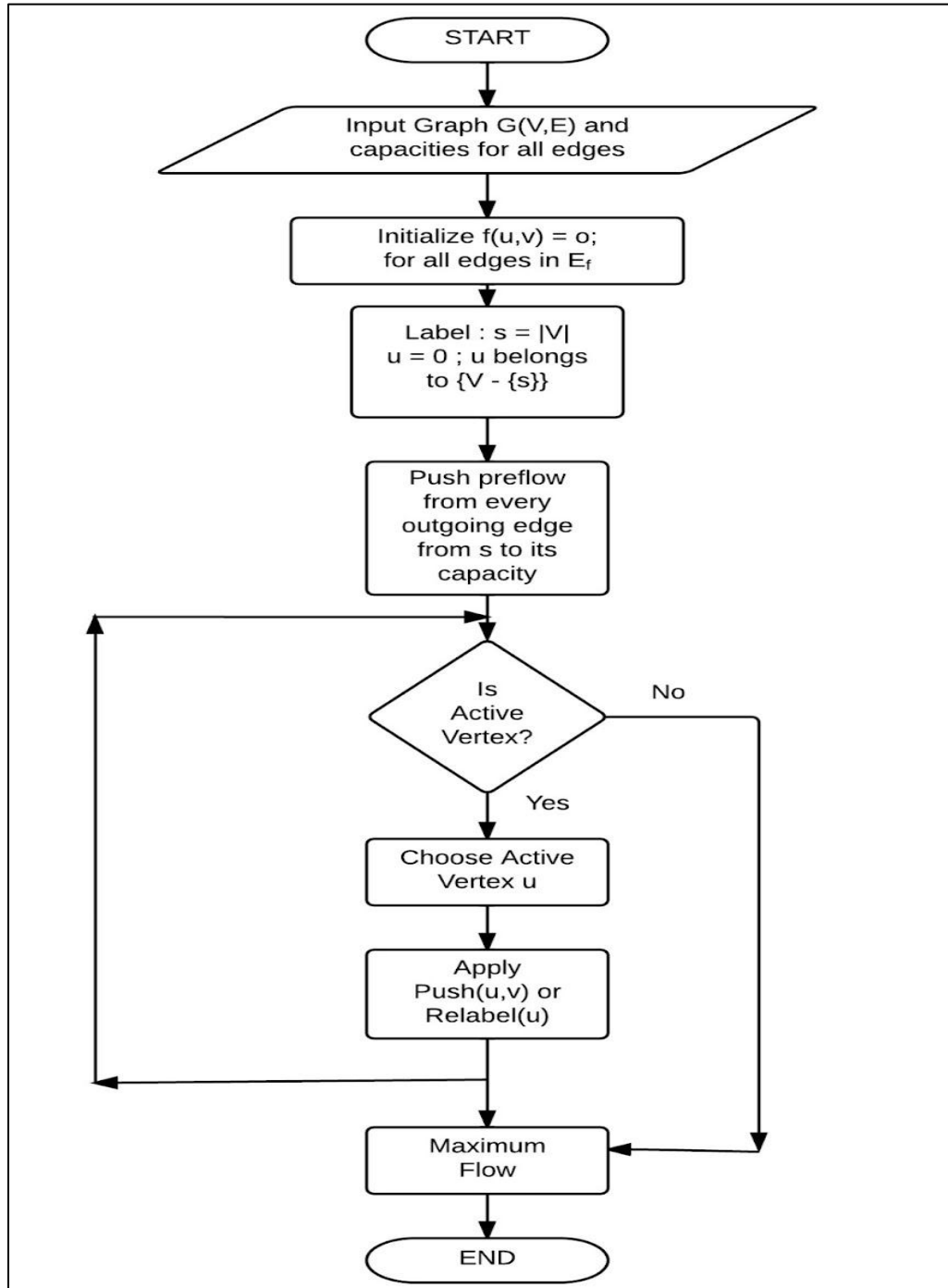


Fig. 4: Push Relabel Algorithm

Consider a flow network $G = (V, E)$ where $n = |V|$, $m = |E|$. Instead of maintaining the flow conservation property, we maintain a preflow f that is anti-symmetric and satisfies the capacity constraints, and allow excess flow $e(u) = f(V, u)$ at the vertices. We maintain a height function to control how flow is pushed through the network: Height: $h(u) \leq h(v) + 1$ if $(u, v) \in E_f$, where E_f is the residual network.

The height of the source s is fixed at $|V|$ and the height of the sink is fixed at 0. All other vertices start with height 0, and are increased as the algorithm progresses, and flow is only allowed from a higher vertex to a lower vertex. If there is an edge from u to v in the residual network, then we are going to push excess flow from u to v if the height difference is exactly 1, subject to capacity constraints. That is, v is lower than u , but just by 1.

3.1.1 Push Operation

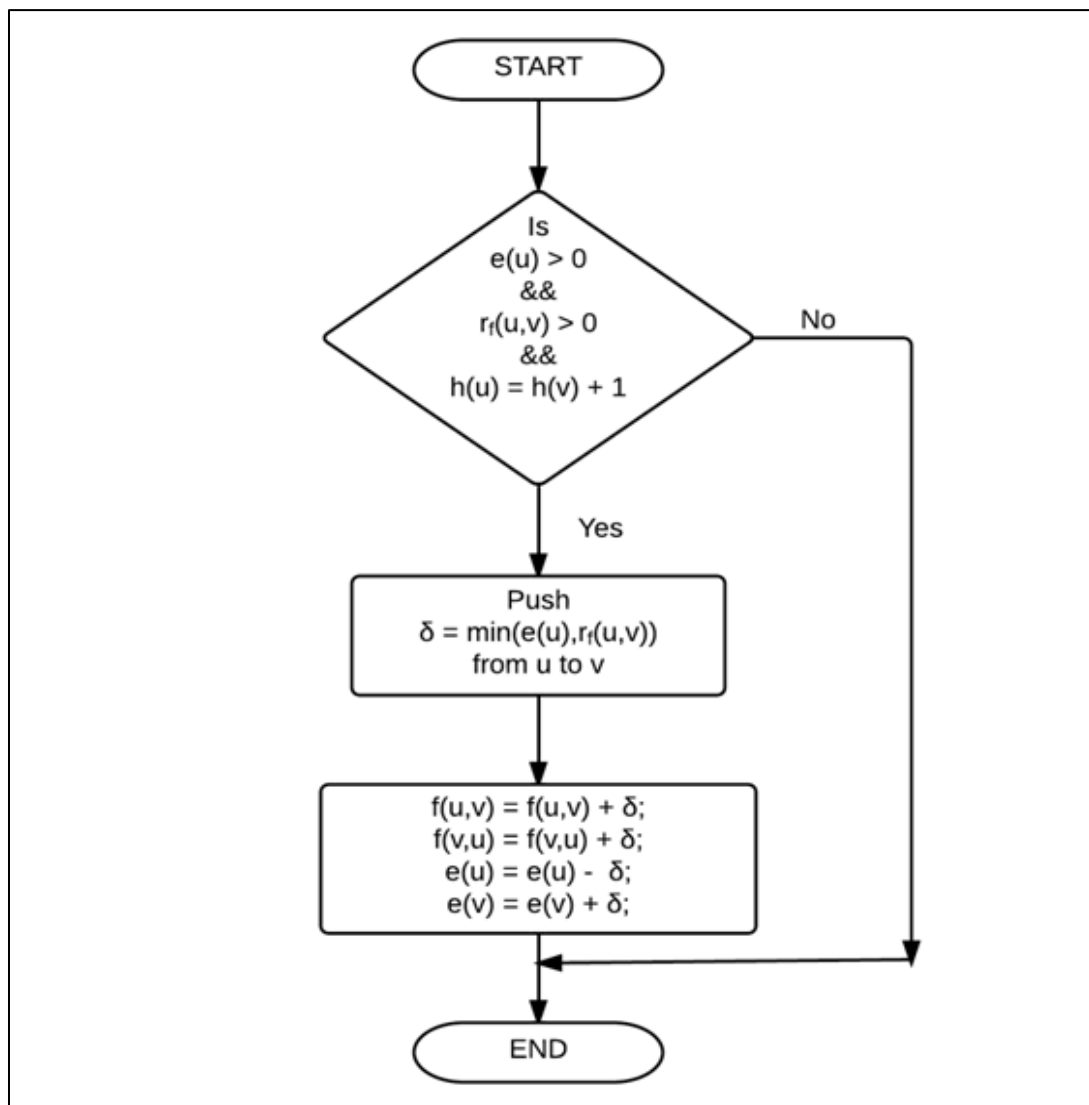


Fig. 5: Push Operation

PUSH(u,v):

Conditions: $e(u) > 0$, $c(u, v) > 0$, $h(u) = h(v) + 1$

Action: Push $df(u, v) = \min(e(u), cf(u, v))$ through (u,v)

$e(u) = e(u) - df(u, v)$

$e(v) = e(v) + df(u, v)$

It is a saturating push if $cf(u,v)=0$ afterward, ie. if we push the maximum possible amount. It is a non-saturating push if $e(u)$ was the amount pushed.

3.1.2 Relabel Operation

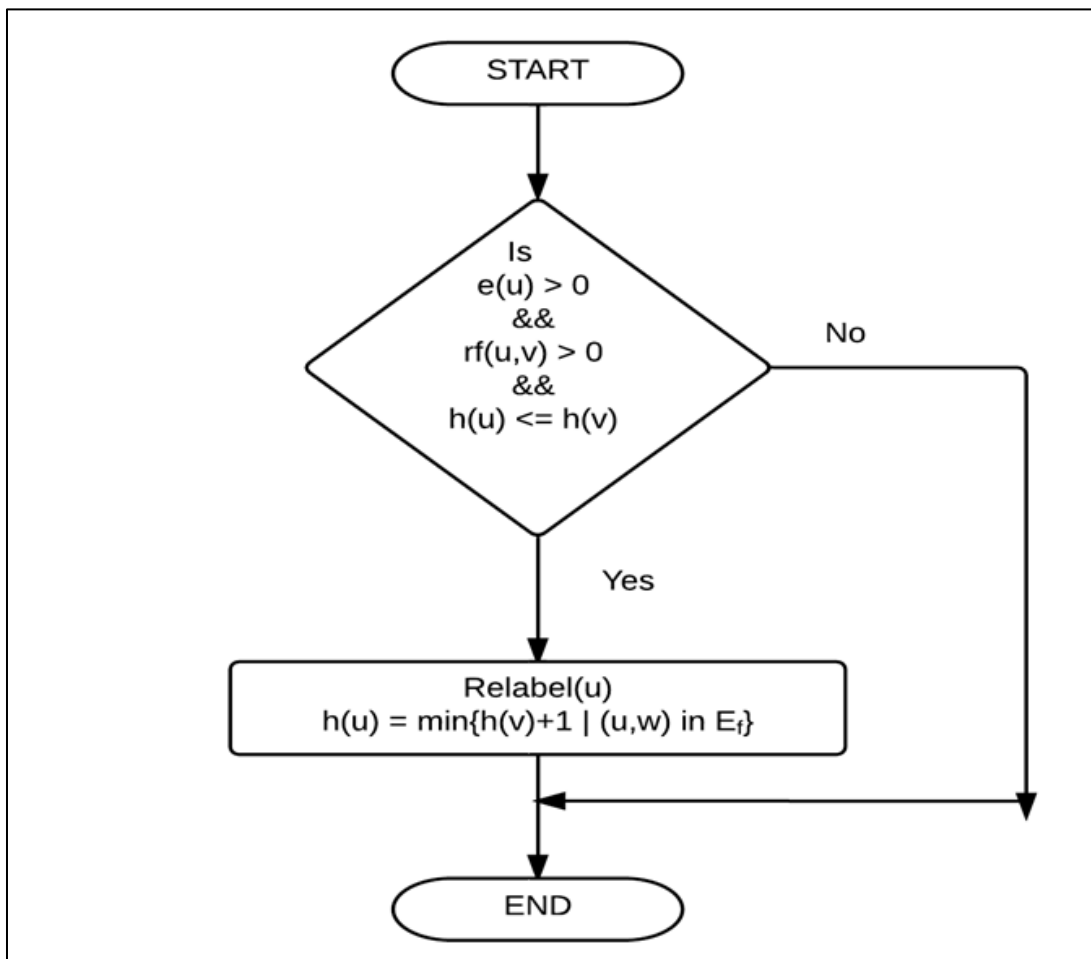


Fig. 6: Relabel Operation

If u is lower than all surrounding vertices, then we cannot Push. In particular, when we have a vertex u with excess, but the height of u is less than or equal to all residual edges, then we “relabel” the height of u to 1 above the minimum height v . Then we can Push after the Relabel.

RELABEL(u):

Condition: $e(u) > 0$, s.t. with $(u, v) \in E_p$, $h(u) \geq h(v) \forall v$

Action: $h(u) = 1 + \min\{h(v) : (u, v) \in E_f\}$

INITIALIZE():

$h(s) = |V| = n$, source is initially the number of v .

$h(t) = 0$

All edges of the form (s, u) have

$cf(s, u) = 0$,

$e(u) = c(s, u)$, all the edges going out of s have an excess flow at u .

$e(s) = -c(s, u)$ thus the source vertex has a negative flow.

3.2 Proposed parallel implementation

The parallel implementation of the push relabel algorithm, involves parallelizing the entire execution of the algorithm over the Linux Cluster, using Message Passing Interface (MPI). The algorithm begins by considering every vertex of the input graph as an individual process. Thus while executing the parallel code using MPI, the number of processes are decided on the basis of the total number of vertices in the input graph. On execution the MPI interface assigns a unique rank to every process. The process with rank 0 is considered to be the master process handling the operation of the source vertex of the input graph. The master process initially reads the input graph file and generates the required capacity matrix and flow matrix including preflow. Once these matrices are generated they are then broadcasted by the master process to all the remaining slave processes. The slave processes acting as the intermediate and the sink vertex of the input graph, receive the required matrices through broadcast functionality provided by the MPI interface. There are certain intermediate vertices which are directly connected to the source and are not awaiting flows from any other vertex, processes handling such an intermediate vertex of the graph can start pushing the flows in parallel to the process of the vertex adjacent to it, as soon as they have received excess from the source vertex. Thus these processes will further send flow to the adjacent vertex process, if and only if there exists an edge with capacity greater than zero between the two vertices and the excess on the vertex of the sending process is not zero. These receiving processes which have received some flow from the all the other connecting vertices in the network from whom it was supposed to receive some flow, will in turn will keep executing the push operations in parallel. As soon as, all the push operations are performed on a respective process it sends the updated flow information and the residual excess on that vertex, to the master process. This is performed by all the process in the network except the source vertex or

the master process. The master then gathers all the received flow and excess information from the network and updates its own local flow matrix along with individual excess of every vertex. These updated flow matrix and excess array for the network is provided as an input to the discharge operation on the master process. The the discharge operation is executed at the master itself since it cannot be parallelized. After complete execution of the discharge operation of the push relabel algorithm, we get the required maximum flow for the input graph.

The entire operation can be visualized by taking a small network flow graph as shown in fig. 7,

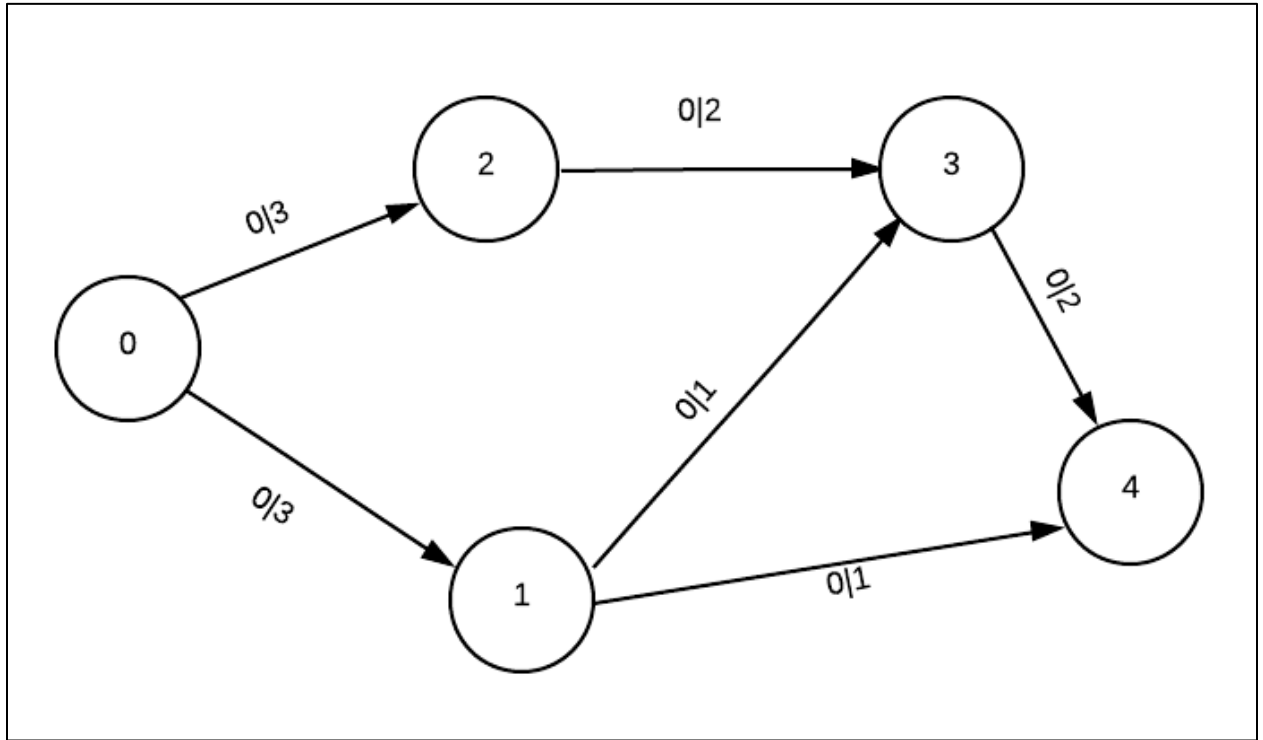


Fig. 7: Sample network flow graph

As the graph is having 5 vertices, 5 processes will be created to operate on each of these vertices. The source vertex i.e. vertex 0 will be processed by the master process with rank 0. This master process will do all the specified operations of reading of graph file, preflow and generating and broadcasting the capacity and flow matrix. The remaining vertices 1, 2, 3, 4 will be processed by the processes with rank 1, 2, 3, 4 respectively.

It can be seen in fig. 7 the processes for vertices 1 and 2 which are receiving the flow from master can push the flow in parallel through the outgoing edges connected to them. While the process on vertex 3 will wait for the processes on vertices 1 and 2 to complete their push operation and once it has been done process on vertex 3 can carry on with its execution. After finishing their job these processes send the initial flows to the master process as shown in the fig.

8 and then master does discharge operation to make excess on each of intermediate vertices 0. Once it has been done the execution terminates giving maximum flow.

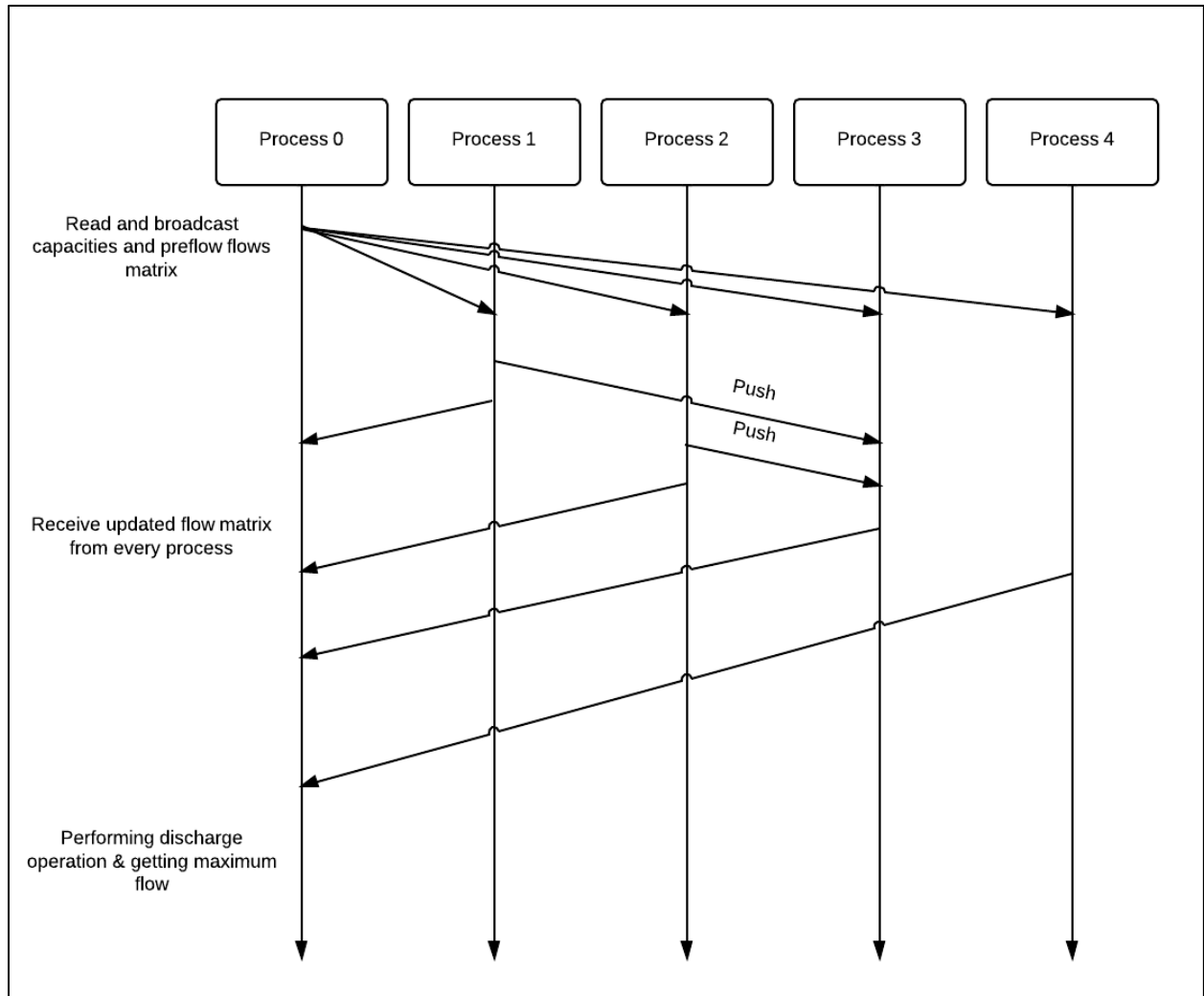


Fig. 8: Time line chart for processes

4. HARDWARE AND SOFTWARE REQUIREMENTS

4.1. Hardware Requirements

4.1.1 Cluster

It consists of many of same or similar types of machines tightly coupled using dedicated network connections. All machines share resources such as a common home directory using distributed file system such as NFS (Network File System) and they must have software such as an MPI (Message Passing Interface) implementation installed to allow programs to be run across all nodes.

4.1.2 Node

It is a standalone "computer in a box" usually comprised of multiple CPUs/processors/cores. A cluster consists of many nodes in which one acts as a master and remaining as slaves.

4.1.3 High Speed Ethernet Switch

It serves as medium for interconnection from master to other slave nodes. It possesses speed upto 1000 Mbps.

4.1.4 KVM switch

KVM (Keyboard, Video and Mouse) switch is a device which allows user to control multiple nodes from one or more keyboard, video monitor and mouse. It is used to access and control each slave node in the cluster and start the required services on them.

4.2. Software Requirements

4.2.1 Network Information Service

It is a client-server directory service protocol. It is used in distributing system configuration data such as users and hostnames between computers on computer network. NIS can allow users to log in on any machine on the network, as long as the machine has the NIS client programs running and the user's password is recorded in the NIS passwd database. It consist of "ypserv" utility server that distributes NIS databases to client systems within an NIS domain and "ypbind" daemon that binds NIS clients to an NIS domain.

4.2.2 Network File System

Network File System (NFS) is a distributed file system protocol which allows a system to share directories and files with others over a network. By using NFS, users and programs can access

files on remote systems almost as if they were local files. NFS consists of at least two main parts, a server and one or more clients. The client remotely accesses the data that is stored on the server machine. In a cluster the master runs nfs-server utility and slaves which acts as clients run nfs.

4.2.3 Passwordless SSH

Secure shell (SSH) is a cryptographic network protocol for securely getting access to the remote computer. Parallel programming over multiple machines will need to be able to communicate amongst the machines while running on behalf of a user. It means that users need to be able to SSH into and amongst the worker nodes without being prompted for a password and for this particular purpose the concept of passwordless SSH can be used.

4.2.4 Message Passing Interface

Message Passing Interface (MPI) is a standardized and portable message-passing system used for performing distributed-memory parallel computing. It offers the programmer the ability to perform: point-to-point communication, collective communication, one-sided communication, parallel I/O, and even dynamic process management for parallel computing.

5. IMPLEMENTATION

5.1 Implementation Details

Implementation of the parallel algorithm was carried over an Ubuntu cluster with all nodes running 12.10 release of the Ubuntu OS. The cluster consists of the following hardware parts:

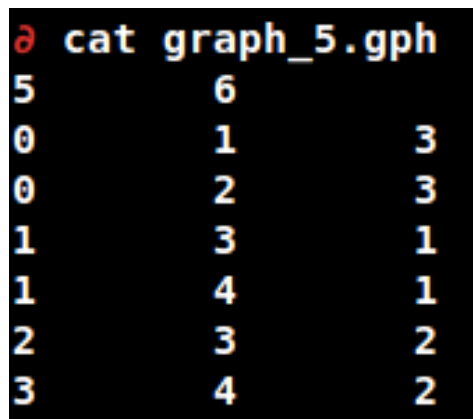
- 1 Master and 4 Slave nodes with Intel P4 processor and 1GB RAM
- High Speed Ethernet switch
- KVM Switch

All nodes (including the master node) run the following software:

- Network File System (NFS)
- Network Information Services(NIS)
- Secure Shell (SSH)
- Message Passing Interface (MPICH2)

5.1.1 Generation of graph file

Both serial as well as parallel implementations of the algorithm read the input graph from a standard text file generated using random graph generator utility. As shown in the fig. 9 this input file has first line specifying the number of vertices and the number of edges. While the remaining lines represent the edges in the graph along with their capacities as from vertex no., to vertex no. and capacity separated by spaces in between them. This is the basic standard for the input graph file followed by both the implementation.



```
cat graph_5.gph
5          6
0          1          3
0          2          3
1          3          1
1          4          1
2          3          2
3          4          2
```

Fig. 9: Input graph file

To generate such graph file using graph generator utility, user has to run its executable along with the arguments as,

1. Number of vertices
2. Maximum capacity of a particular edge
3. Starting vertex index
4. Number of stages in the graph
5. Name of the output file containing graph

Thus by running the executable with these specified arguments an output graph file with specified name can be obtained. This graph file is further used as input for both serial and parallel implementation of the algorithm.

5.1.2 Serial Implementation

The serial code for the algorithm is written in ANSI C language. It has been compiled and executed using GCC compiler. For execution it takes only one argument as input graph file. The output is stored in a separate text file consisting of maximum flow along with final flow at each edge shown with its respective capacity. Fig. 10 shows the format of the output file.

```
➤ cat output.txt
-----
Maximum flow : 3
-----
0 --[1 | 3]--> 1
0 --[2 | 3]--> 2
1 --[1 | 1]--> 4
2 --[2 | 2]--> 3
3 --[2 | 2]--> 4
-----
```

Fig. 10: Output file format

5.1.3 Parallel Implementation

The parallel code for the algorithm is also written in ANSI C language supported with Message Passing Interface (MPI) library for C language. This library is included as a header file (i.e. mpi.h). Since the entire message passing environment has been setup using MPI interface, MPI specific coding guidelines had to be followed. The parallel code for the entire cluster is written in a file with the mpi.h header file included. The entire code of the implementation is kept in a shared folder which is mapped on all the nodes of the cluster using Network File Server (NFS) linux utility. Thus whenever the code is modified the updated copy of the code is available at all the nodes at runtime.

The parallel code is compiled using a special compiler provided under MPICH2 implementation of mpi library using Makefile utility as shown in the following fig.11,

```
∅ make
mpicc -std=c99 -c para_push2.c
mpicc -o para_push2 para_push2.o -lm
```

Fig. 11: Commands for compiling the parallel code

Now the executable file is generated and is available on all the nodes. To run this executable in parallel environment MPICH2 provides a special utility named '**mpiexec**'. The following fig.12 illustrates the execution syntax:

```
∅ mpiexec -n 100 -f machinefile ./para_push2 rgraph_100.gph
```

Fig. 12: Command for executing parallel code

Here, -n argument specifies the number of processes to be generated

-f argument specifies the name of the machinefile which contains the hostnames of all the nodes present in the cluster.

These arguments are followed by the executable file along with the input graph file. The output for the parallel execution has same format as that in case of output of serial code.

5.1.4 Basic functions of MPI library used in the implementation

1. MPI_Send()

Prototype :

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
MPI_Comm comm)
```

Here,

buf : initial address of send buffer (choice)

count : number of elements in send buffer (nonnegative integer)

datatype : datatype of each send buffer element (handle)

dest : rank of destination (integer)

tag : message tag (integer)

comm : communicator (handle)

It has been used to send the flow value to the intended process. It is blocking routine thus the sending process is blocked until the message is received by the destination process.

2. MPI_Receive()

Prototype :

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int
tag, MPI_Comm comm, MPI_Status *status)
```

Here,

buf : initial address of receive buffer (choice)

count : maximum number of elements in receive buffer (integer)

datatype : datatype of each receive buffer element (handle)

source : rank of source (integer)

tag : message tag (integer)

comm : communicator (handle)

It has been used to receive the flow value from the intended process. It is blocking routine thus the receiving process is blocked until the message sent by the sender process is received.

3. MPI_Bcast()

Prototype

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
MPI_Comm comm )
```

Here,

buffer : starting address of buffer (choice)

count : number of entries in buffer (integer)

datatype : data type of buffer (handle)

root : rank of broadcast root (integer)

comm : communicator (handle)

It has been used by the master process i.e. process with rank 0 to broadcast the capacities as well as initial flow matrix after the preflow to all remaining process. It is again a blocking routine.

5.2 Implementation Issues

5.2.1 Communication overhead due to the interdependencies

For the interdependency present in the graph the process working on a particular node has to wait for the other process from which it is supposed to receive the pushed flow value. Also for a considerable amount of interdependencies a sufficient amount of parallelization and thus the speedup can be obtained. But if the graph contains large amount interdependencies, there will be large number of processes waiting for the other processes to finish their job. This is associated with the high amount of message passing for the flow updates which can create a bottleneck in the communication between the processes.

5.2.2 Crashing of node in cluster

Crashing of any node in the cluster during the execution may halt the execution by giving the communication error. As each node in the cluster executes a set of processes, crashing of it will crash all the processes running on it. Thus if for the processes running on the other node want to send or receive the some data from processes of crashed node, they will not be able to do so as the required processes will not be available.

5.2.3 Purely serial code

While designing a parallel implementation ideally all work is considered to be parallelized. But it may not be true in all the cases as there can be some part of algorithm which can not be parallelized as in the case of push relabel algorithm the initial push operation is performed in parallel but the discharge operation to make excess on each vertex zero is performed serially.

5.2.4 Unused resources

Once processes are allocated to a particular node they execute on that node only. But there can be conditions where processes running on a particular node may finish their execution while processes on other nodes still running. In such cases there won't be proper utilization of processing power. The resources will be left unused.

5.3 Screenshots



Fig. 13: Cluster setup in R & D lab

6. TESTING

Sr.no	Number of vertices	Serial Execution Time	Parallel Execution Time	Speedup
1	10	0.0070	0.0199	0.3516
2	50	0.5490	0.2932	1.8724
3	75	1.7060	0.7519	2.2689
4	100	3.9490	1.6214	2.4355
5	125	7.8800	2.8659	2.7496
6	150	13.0500	3.5886	3.6365
7	175	20.9880	5.3616	3.9145
8	200	30.7550	7.1105	4.3253

Table 1: Serial and parallel code execution time comparison

Sr.no	Number of vertices	Serial CPU Utilization	Parallel CPU Utilization	Speedup
1	10	54	21	2.5714
2	50	91	37	2.4594
3	75	96	43	2.2325
4	100	98	43	2.2791
5	125	98	48	2.0417
6	150	99	56	1.7679
7	175	97	68	1.4264
8	200	99	72	1.3750

Table 2: Serial and parallel code CPU utilization comparison

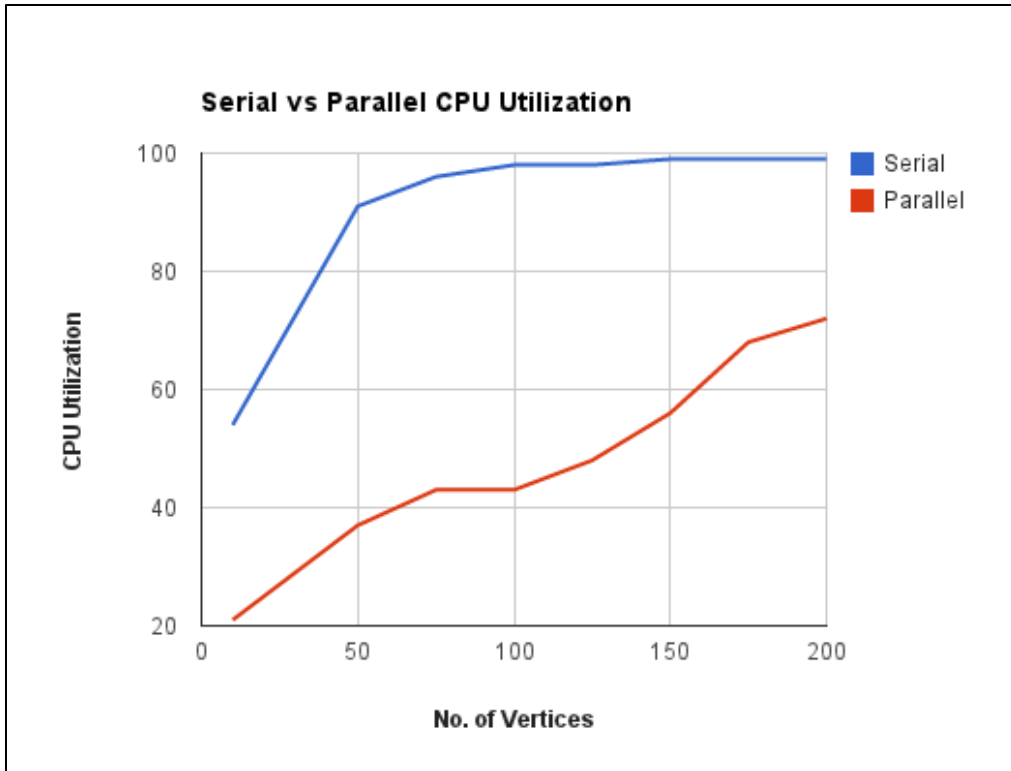


Fig. 14: Serial vs. parallel execution time

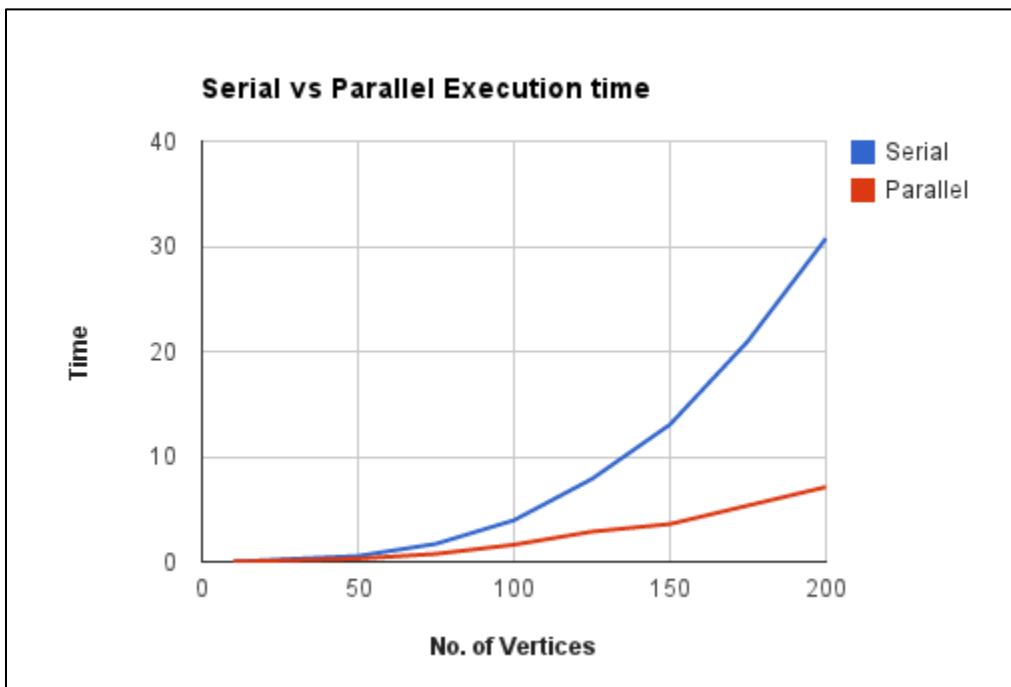


Fig. 15: Serial vs. parallel CPU utilization

7. FUTURE SCOPE

Our current system focuses on multiprocessor model with no shared address space on a Linux cluster. So this in certain cases causes a little overhead for processing the entire parallel algorithm, like in cases where we have an input graph with minimum stages or less number of independent vertices, causes a considerable drop in speedup due to less parallelism. This is mainly due to the message passing overhead which increases on such input graphs. Thus if we improve our implementation in future for multi core processors along with the processing power of the onboard GPU processing capabilities, then we can reduce this overhead by multithreading models.

8. CONCLUSION

The Maximum flow problem was studied and a parallel algorithm to solve the same was implemented successfully over the cluster. While testing the parallel algorithm we could observe considerable amount of speedup in terms of execution times and CPU utilization on each node on the cluster. This parallel implementation has a great scope for application oriented utilization in various forms of networks, especially to solve real time network problems. This parallel implementation can help in evaluating real time updates for maximum flow evaluation on any type of network. E.g. Flow of flights from various different airports in the world can be scheduled effectively to avoid bottle neck situations of over jamming. Even applications in traffic management systems and water management systems can be implemented effectively.

9. REFERENCES

- [1] Cormen, Leiserson, and Rivest. Introduction to Algorithms. The MIT Press. 1990.
- [2] Bertsekas, D. P., "A Distributed Asynchronous Relaxation Algorithm for the Assignment Problem," Proc. 24th IEEE Conf. Dec. & Contr., 1985, pp. 1703-1704.
- [3] Bertsekas, D. P., "Distributed Asynchronous Relaxation Methods for Linear Network Flow Problems," Lab. for Information and Decision Systems Report P-1606, M.I.T., November 1986.
- [4] Bertsekas, D. P., Linear Network Optimization: Algorithms and Codes, M.I.T. Press, Cambridge, Mass., 1991.
- [5] Bertsekas, D. P., and Tsitsiklis, J. N., Parallel and Distributed Computation: Numerical Methods, Prentice-Hall, Englewood Cliffs, N. J., 1989.
- [6] B. V. Cherkassky and A. V. Goldberg, "On implementing push-relabel method for the maximum flow problem," Stanford University, Stanford, CA, USA, Tech. Rep., 1994.
- [7] Dey, Tamal K.. Lecture for CSE 794: Advanced Algorithms, Department of Computer and Information Sciences, The Ohio State University, Columbus, Ohio, USA. April 7, 2009.
- [8] Jensen, P.A., Barnes, J.W. 1980: Network flow programming. J. Wiley & Sons, New York, NY.
- [9] "Push-Relabel and Bipartite Matching Advanced Algorithms" (CSE 794) Instructor: Tamal K. Dey Scribe: Ted Bulwinkle, April 30, 2009.
- [10] R. Anderson and J. Setubal, "On the parallel implementation of goldberg's maximum flow algorithm," in 4th Annual Symposium Parallel Algorithms and Architectures (SPAA92) San Diego, CA, July 1992, pp. 168–177.
- [11] Hassin, R., Johnson, D.B. 1985: An $O(n \log^2 n)$ algorithm for maximum flow in undirected planar network. SIAM J. Comput. 14, 612-624.
- [12] Sleator, D.D. 1980: An $O(nm \log n)$ algorithm for maximum network flow. Technical Report STAN-CS-80-831, Department of Computer Science, Stanford University.

10. APPENDIX

/* 1. Graph generator utility to generate random graph with specified number of vertices, minimum edge capacity and stages */

/* gengraph.c */

#include<stdio.h>

#include<stdlib.h>

static int ste = 31;

/*

* randint() function is called to generate random capacities for different arcs

*/

int randint(int max)

{

int min = 2;

srand(time(NULL) - ste);

ste *= 313;

return min+(int)((int)(max-min+1)*(rand()/(RAND_MAX+1.0)));

}

int main(int argc, char* argv[])

{

/*

* Checking whether the user has passed correct number of arguments

*/

if(argc != 11){

printf(" Format Error : exit(0)\n Usage: ./gen -v <vertices> -c <capacity> -s
<start-vertex> -st <stages> -f <filename> \n ");

exit(0);

}

FILE *f = fopen(argv[10],"w");

```

int i,j;

int v = atoi(argv[2]);

int st = atoi(argv[8]);


int c = atoi(argv[4]);

int s = atoi(argv[6]);


int **graph;

int limit = v + 1;


graph = (int **) calloc(limit, sizeof(int *));
for (i = s; i < limit; ++i){
    graph[i] = (int *) calloc(limit, sizeof(int));
}

/*
*Count variable to count the total number of arcs
*/

int counter = 0;

if (v - 2 < st || st == 0){
    printf("\nInvalid number of stages\nEXIT[0]\n");
    exit(0);
}

for (i = 1; i <= st; ++i){
    graph[1][1 + i] = randint(c);
    counter ++;
    for (j = 1 + i; j < v; j = j + st){
        if (j + st < v){

```



```

        graph[j][j + st] = randint(c);
        counter ++;
    }
}
graph[j - st][v] = randint(c);
counter ++;
}

```

//adding random arcs in the graph

```

int ran_arc_num = v / 4;
int a, b, from, to;
for (i = 0; i < ran_arc_num; ++i){
    do{
        a = randint(v - 1);
        b = randint(v - 1);
    }while(a == b);
    if (a < b){
        from = a;
        to = b;
    }else{
        from = b;
        to = a;
    }
    if (graph[from][to] == 0){
        graph[from][to] = randint(c);
        counter ++;
    }
}

```

```

    }

    //adding first line in the output file
    fprintf(f,"%d\t%d\n",v,counter);

    //generating the graph files with respect the source vertex
    for (i = 1; i < limit; ++i){
        for (j = 1; j < limit; ++j){
            if (graph[i][j] != 0){
                if (s == 1)
                {
                    fprintf(f,"%d\t%d\t%d\n", i, j,graph[i][j]);
                }else{
                    fprintf(f,"%d\t%d\t%d\n", i - 1, j - 1, graph[i][j]);
                }
            }
        }
    }

    printf("\n File created : %s \t[ OK ]\n ARC COUNT : %d \t[ OK ]\n\n",argv[10],counter);
    fclose(f);
    return 0;
}

/* Makefile for graph generator */

CC = gcc
TARGET = gen

${TARGET}:

```

```
    ${CC} -o ${TARGET} gengraph.c

clean :

    rm gen
```

```
/* 2. Serial code for push relabel algorithm */
```

```
/* header file: push_relabel.h */
```

```
#include <stdlib.h>
```

```
#include <stdio.h>
```

```
#include <time.h>
```

```
#define MIN(X,Y) X < Y ? X : Y
```

```
#define INFINITE 10000000
```

```
int num_vertices, num_edges;
```

```
FILE *fo;
```

```
void print_matrix(int **);
```

```
void print_result(int **, int **);
```

```
int push(int **, int **, int *, int, int);
```

```
void relabel(int **, int **, int *, int);
```

```
int push_relabel(int **, int **, int, int);
```

```
void preflow(int **, int **, int *, int);
```

```
void send_flow(int **, int **, int *, int *, int);
```

```
void rearrange(int, int *);
```

```

/* Source code : push_relabel.c */

#include "push-relabel.h"

void print_mat(int **mat) {
    int i, j;

    printf("-----\n");

    for (i = 0; i < num_vertices; i++) {
        for (j = 0; j < num_vertices; j++)
            if (mat[i][j] > 0) {
                printf("from %d to %d : %d\n", i, j, mat[i][j]);
            }
    }

    printf("\n");
    printf("-----\n\n");
}

void print_result(int **c, int **f) {
    int i, j;

    fprintf(fo, "-----\n");

    for (i = 0; i < num_vertices; i++) {
        for (j = 0; j < num_vertices; j++)
            if (f[i][j] > 0) {

```

```

        fprintf(fo, "%d --[%d | %d]--> %d\n", i, f[i][j], c[i][j], j);
    }
}

fprintf(fo, "\n");
fprintf(fo, "-----\n\n");
}

```

//to provide initial preflow from source to connected vertices

```
void preflow(int **c, int **f, int *excess, int source){
```

```

    for (int i = 0; i < num_vertices; i++) {
        if (c[source][i] != 0) {
            f[source][i] += c[source][i];
            f[i][source] -= c[source][i];
            excess[i]    = c[source][i];
            excess[source] -= c[source][i];
        }
    }
}

```

//push operation is easy to understand

```

int push(int **c, int **f, int *excess, int u, int v){
    int push_flow = MIN(excess[u], c[u][v] - f[u][v]);
    f[u][v] += push_flow;
    f[v][u] -= push_flow;
}

```

```

    excess[u] -= push_flow;
    excess[v] += push_flow;
    return push_flow;
}

```

//relabeling operation will increase the height of the
 //active vertex w.r.t the connected vertex with minimum height
 //and for which the edge is following capacity and flow constraints

```

void relabel(int **c, int **f, int *height, int u){
    int min_height = INFINITE;
    for (int v = 0; v < num_vertices; v++) {
        if (c[u][v] - f[u][v] > 0) {
            min_height = MIN(min_height, height[v]);
            height[u] = min_height + 1;
        }
    }
}

```

//required push and relabel operations are performed in this function
 //as shown all the vertices connected to active vertex are checked for
 //push of flow. Height constraints are also included.

```

void send_flow(int **c, int **f, int *excess, int *height, int u){
    int v = 0, sent;
    while(excess[u] > 0){
        if (v < num_vertices) {
            if ((c[u][v] - f[u][v] > 0) && (height[u] > height[v])) {

```

```

    printf("Active vertex found : %d\n", u);
    printf("Relabeled %d to : %d\n", u, height[u]);
    sent = push(c, f, excess, u, v);
    printf("%d-[%d]->%d\n", u, sent, v);
}
v++;
}else {
    relabel(c, f, height, u);
    v = 0;
}
}
}

//function is called to reinitialize the active vertex routine
//in this the processed vertex brought to front and the remaining
//vertices are checked again for activeness in further routine
void rearrange(int k, int *intermediate){
    int temp = intermediate[k];
    for (int j = k; j > 0 ; j--) {
        intermediate[j] = intermediate[j - 1];
    }
    intermediate[0] = temp;
}

int push_relabel(int **c, int **f, int source, int sink){
    int i, j = 0, old_height, sum = 0;

```

```

int *excess    = (int *)calloc(num_vertices, sizeof(int));
int *excess1   = (int *)calloc(num_vertices, sizeof(int));
int *height    = (int *)calloc(num_vertices, sizeof(int));
int *height1   = (int *)calloc(num_vertices, sizeof(int));
int *intermediate = (int *)calloc(num_vertices, sizeof(int));
for (i = 0; i < num_vertices*2; i++) {
    for (j = 0; j < num_vertices*2; j++) {
        for (int k = 0; k < num_vertices; k++) {
            excess1[k] = j;
            height1[k] = k;
        }
    }
}

//initializing with preflow
preflow(c, f, excess, source);

//initializing the height of source
height[source] = num_vertices;

//getting intermediate nodes for processing
for (i = 0; i < num_vertices; i++) {
    if (i != 0 && i != num_vertices - 1) {
        intermediate[i - 1] = i;
    }
}

//performing operations considering single intermediate vertex at a time

```



```

printf("Format : from-[sent flow]->to\n");
for (i = 0; i < num_vertices*10; i++) {
    for (j = 0; j < num_vertices*10; j++) {
        for (int k = 0; k < num_vertices; k++) {
            excess1[k] = k;
            height1[k] = i;
        }
    }
}

for (i = 0; i < num_vertices - 2; i++) {
    int u = intermediate[i];
    old_height = height[u];

    //function will check if the provided vertex is active or not.

    //in the case of active it will push the flow accordingly
    send_flow(c, f, excess, height, u);

    //when old_height is lesser than the new one the active vertex has been found,
    //the graph's flow has been changed and we need to reinitialize the the algorithm
    //because there is a possibility that the vertex which has been processed earlier
    //may become active again.
    if (old_height < height[u]) {
        rearrange(i, intermediate);
        i = 0;
    }
}

for (i = 0; i < num_vertices*10; i++) {

```

```

    for (j = 0; j < num_vertices*10; j++) {
        for (int k = 0; k < num_vertices; k++) {
            excess1[k] = i;
            height1[k] = j;
        }
    }
}

//final labels on the vertices
printf("\nFinal heights : \n");
printf("-----\n");

for (i = 0; i < num_vertices; i++) {
    printf("Height[%d] : %d\n", i, height[i]);
}

for (i = 0; i < num_vertices*10; i++) {
    for (j = 0; j < num_vertices*10; j++) {
        for (int k = 0; k < num_vertices; k++) {
            excess1[k] = k;
            height1[k] = j;
        }
    }
}

printf("-----\n");
fprintf(fo, "-----\n");

//Maximum flow obtained at the sink

```

```

for (i = 0; i < num_vertices; i++) {
    if (f[i][num_vertices - 1] > 0) {
        sum += f[i][num_vertices - 1];
    }
}

for (i = 0; i < num_vertices*10; i++) {
    for (j = 0; j < num_vertices*10; j++) {
        for (int k = 0; k < num_vertices; k++) {
            excess1[k] = 0;
            height1[k] = 0;
        }
    }
}

printf("Maximum flow : %d\n", sum);
fprintf(fo, "Maximum flow : %d\n", sum);

}

int main(int argc, const char *argv[]) {

    clock_t tic = clock();

    int **flow, **capacities, i, from, to, linenumber = 1, capacity;
    FILE *fp;

    if (argv[1] == NULL) {
        printf("Format is : ./push-relabel <graphfile>\n");
        exit(0);
    } else {

```

```

//output file
fo = fopen("output.txt","w");

//input file
fp = fopen(argv[1], "r");
}

//getting number of vertices and edges
fscanf(fp, "%d\t%d", &num_vertices, &num_edges);

//initializing the flows and the capacities
flow    = (int **) calloc(num_vertices, sizeof(int*));
capacities = (int **) calloc(num_vertices, sizeof(int*));

for (i = 0; i < num_vertices; i++) {
    flow[i]    = (int *) calloc(num_vertices, sizeof(int));
    capacities[i] = (int *) calloc(num_vertices, sizeof(int));
}

//getting the info about capacities and edges
for (i = 0; i < num_edges; i++) {

    fscanf(fp, "%d\t%d\t%d", &from, &to, &capacity);
    capacities[from][to] = capacity;
}

//printing the capacities of the matrix
printf("\nCapacities:\n");
print_mat(capacities);

```

```

push_relabel(capacities, flow, 0, num_vertices - 1);

//printing the final flow in the network
printf("\nFlows:\n");
print_mat(flow);
print_result(capacities, flow);
clock_t toc = clock();
printf("Execution time : %f seconds\n", (double)(toc - tic)/1000000);
    printf("-----\n");
return 0;
}

```

/* Makefile for serial code */

CC = gcc

CCFLAGS = -lm

SOURCE = push-relabel.c

OBJS = push-relabel.o

HEADERS = push-relabel.h

TGT = push-relabel

all: \$(TGT)

\$(TGT): \$(OBJS)

\$(CC) -o \$(TGT) \$(OBJS) \$(CCFLAGS)

push-relabel.o: \$(SOURCE) \$(HEADERS)

\$(CC) -std=c99 -g -c \$<

clean:

```
rm -rf *.o $(TGT)
```

/* 3. Parallel code for push relabel algorithm */

/* header file: para_push2.h */

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
#include <mpi.h>
```

```
#include <math.h>
```

```
#include <time.h>
```

```
#define MAX_NAME_LEN 10
```

```
#define INFINITE 1000000
```

```
#define MIN(X,Y) X < Y ? X : Y
```

```
int num_vertices, num_edges, mat_size;
```

```
FILE *fo;
```

```
void print_mat(int **);
```

```
int *get_mat_column(int, int **, int *);
```

```
int get_mat_column_sum(int index, int **mat);
```

```
void preflow(int **, int **, int);
```

```
int par_push(int **, int **, int, int, int);
```

```
int push(int **, int **, int *, int, int);
```

```
void relabel(int **, int **, int *, int);
```

```

void send_flow(int **, int **, int *, int *, int);

void rearrange(int, int *);

int push_relabel(int **, int **, int *, int, int);


/* Source file : para_push.c */


#include "para_push2.h"


// Printing the matrix values
void print_mat(int **mat) {
    int i, j;

    printf("-----\n");

    for (i = 0; i < num_vertices; i++) {
        for (j = 0; j < num_vertices; j++)
            if (mat[i][j] > 0) {
                printf("from %d to %d : %d\n", i, j, mat[i][j]);
            }
    }

    printf("\n");
    printf("-----\n\n");
}


// getting the column of a matrix
int *get_mat_column(int index, int **mat, int *temp){

    int count = 0;
    for (int i = 0; i < num_vertices; ++i){

```

```

        temp[i] = mat[count][index];
        count ++;
    }
    return temp;
}

// getting the column sum for particular index of the matrix
int get_mat_column_sum(int index, int **mat){

    int sum = 0;
    for (int i = 0; i < num_vertices; ++i){

        sum += mat[i][index];
    }
    return sum;
}

/*-----CODE OF ACTUAL SERIAL ALGO-----*/

// to provide initial preflow from source to connected vertices
void preflow(int **c, int **f, int source){

    for (int i = 0; i < num_vertices; i++) {
        if (c[source][i] != 0) {
            f[source][i] += c[source][i];
        }
    }
}

// Used in parallel pushes

```



```

int par_push(int **c, int **f, int excess, int u, int v){

    // getting the amount of flow to be pushed
    int push_flow = MIN(excess, c[u][v] - f[u][v]);

    f[u][v] += push_flow;
    return push_flow;
}

// Used for remaining pushes
int push(int **c, int **f, int *excess, int u, int v){
    int push_flow = MIN(excess[u], c[u][v] - f[u][v]);
    f[u][v] += push_flow;
    f[v][u] -= push_flow;
    excess[u] -= push_flow;
    excess[v] += push_flow;

    return push_flow;
}

void relabel(int **c, int **f, int *height, int u){

    int min_height = INFINITE;
    for (int v = 0; v < num_vertices; v++) {
        if (c[u][v] - f[u][v] > 0) {
            min_height = MIN(min_height, height[v]);
            height[u] = min_height + 1;
        }
    }
}

```

```

void send_flow(int **c, int **f, int *excess, int *height, int u){
    int v = 0, sent;
    while(excess[u] > 0){
        if (v < num_vertices) {
            if (( c[u][v] - f[u][v] > 0 ) && ( height[u] > height[v] )) {

                fprintf(fo, "Active vertex found : %d\n", u);
                fprintf(fo, "Relabeled %d to : %d\n", u, height[u]);
                sent = push(c, f, excess, u, v);
                fprintf(fo, "%d-[%d]->%d\n", u, sent, v);

            }
            v++;
        }else {
            relabel(c, f, height, u);
            v = 0;
        }
    }
}

```

```

void rearrange(int k, int *intermediate){
    int temp = intermediate[k];
    for (int j = k; j > 0 ; j--) {
        intermediate[j] = intermediate[j - 1];
    }
    intermediate[0] = temp;
}

```

```

int push_relabel(int **c, int **f, int *excess, int source, int sink){

    int i, j = 0, old_height, sum = 0;

```

```

int *height    = (int *)calloc(num_vertices, sizeof(int));
int *intermediate = (int *)calloc(num_vertices, sizeof(int));

//initializing the height of source
height[source] = num_vertices;

//getting intermediate nodes for processing
for (i = 0; i < num_vertices; i++) {
    if (i != 0 && i != num_vertices - 1) {
        intermediate[i - 1] = i;
    }
}

//performing operations considering single intermediate vertex at a time
fprintf(fo, "Format : from-[sent flow]->to\n");

for (i = 0; i < num_vertices - 2; i++) {
    int u = intermediate[i];
    old_height = height[u];
    //function will check if the provided vertex is active or not.
    //in the case of active it will push the flow accordingly
    send_flow(c, f, excess, height, u);
    //when old_height is lesser than the new one the active vertex has been found,
    //the graph's flow has been changed and we need to reinitialize the the algorithm
    //because there is a possibility that the vertex which has been processed earlier
    //may become active again.
    if (old_height < height[u]) {
        rearrange(i, intermediate);
        i = 0;
    }
}

```

```

//final labels on the vertices
fprintf(fo, "\nFinal heights : \n");
fprintf(fo, "-----\n");

for (i = 0; i < num_vertices; i++) {
    fprintf(fo, "Height[%d] : %d\n", i, height[i]);
}

fprintf(fo, "-----\n");
//Maximum flow obtained at the sink

for (i = 0; i < num_vertices; i++) {
    if (f[i][num_vertices - 1] > 0) {
        sum += f[i][num_vertices - 1];
    }
}
fprintf(fo, "Maximum flow : %d\n", sum);
}
/*-----*/

int main(int argc, char const *argv[])
{
    clock_t tic = clock();

    MPI_Init(NULL, NULL);

    int size, rank;

    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

```

```

double start = MPI_Wtime();
int **flow, **capacities, i, from, to, linenum = 1, capacity;

// Master
//initializing the flows and the capacities
flow = (int **) calloc(size, sizeof(int*));
capacities = (int **) calloc(size, sizeof(int*));

for (i = 0; i < size; i++) {
    flow[i] = (int *) calloc(size, sizeof(int));
    capacities[i] = (int *) calloc(size, sizeof(int));
}

if (rank == 0){

    FILE *fp;
    if (argv[1] == NULL) {

        exit(0);

    }else {

        //output file
        fo = fopen("output.txt", "w");

        //input file
        fp = fopen(argv[1], "r");
    }

    // getting number of vertices and edges
    fscanf(fp, "%d\t%d", &num_vertices, &num_edges);

```

```

// number of processes should be equal to number of vertices
if (num_vertices != size) {
    printf("Number of processes should be %d\n", num_vertices);
    printf("Exiting.....\n");
    exit(0);
}

// getting the info about capacities and edges
for (i = 0; i < num_edges; i++) {

    fscanf(fp, "%d\t%d\t%d", &from, &to, &capacity);
    capacities[from][to] = capacity;
}

// printing the capacities of the matrix
printf("\nCapacities:\n");
print_mat(capacities);

preflow(capacities, flow, 0);

//TESTING
printf("\nAfter preflow:\n");
print_mat(flow);
}

// broadcasting number of vertices and edges to the other processes.
MPI_Bcast(&num_vertices, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&num_edges, 1, MPI_INT, 0, MPI_COMM_WORLD);

```

```

// broadcasting the capacities and flows to other processes
for (i = 0; i < num_vertices; i++) {

    MPI_Bcast(&capacities[i][0], num_vertices, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Bcast(&flow[i][0], num_vertices, MPI_INT, 0, MPI_COMM_WORLD);
}

if ((rank != 0) && (rank != num_vertices - 1)){

    // local variables required for the other processes
    int *loc_outqueue;
    loc_outqueue = flow[rank];
    // int capacities[num_vertices][num_vertices];
    int *send_master, *source_nodes;

    //to send to the receiving vertex
    int pushed_amount;

    send_master = (int *) calloc(num_vertices, sizeof(int));
    source_nodes = (int *) calloc(num_vertices, sizeof(int));

    int in_sum = 0, out_sum = 0, index = 0, i, j, excess;
    int vertex_out, vertex_in;

    // pushing the flow from the process where it is possible
    source_nodes = get_mat_column(rank, capacities, source_nodes);
    for (int i = 0; i < num_vertices; ++i){

        if (source_nodes[i] != 0){

```

```

    if (flow[i][rank] == 0) {

        MPI_Recv(&pushed_amount, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);

        //TESTING

        //printf("I am vertex %d and from '%d' i received : %d\n", rank + 1, i + 1,
pushed_amount);

        // updating the flow matrix
        flow[i][rank] += pushed_amount;
    }
}

in_sum = get_mat_column_sum(rank, flow);

for (int i = 0; i < num_vertices; i++) {
    out_sum += loc_outqueue[i];
}

excess = in_sum - out_sum;
for (i = 1; i < num_vertices; i++) {
    if(capacities[rank][i] != 0){

        // TESTING

        //printf("for vertex %d surplus is %d\n", rank + 1, surplus);

        pushed_amount = par_push(capacities, flow, excess, rank, i);

        // TESTING

```



```

//printf("vertex %d sending to %d the flow %d\n", rank + 1, i, pushed_amount);

// TESTING
//printf("Rank %d sending to %d the flow %d \n", rank, i - 1, pushed_amount);

if (i != num_vertices - 1) {
    MPI_Send(&pushed_amount, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
}
excess -= pushed_amount;

// TESTING
//printf("Surplus after push %d\n", surplus);
}
}

/*TESTING
*printf("For process %d\n", rank);
*printf("local flows :\n");
*print_mat(flow);
*/

// sending updated flows and surplus to the master
send_master = flow[rank];
MPI_Send(&send_master[0], num_vertices, MPI_INT, 0, 0, MPI_COMM_WORLD);
MPI_Send(&excess, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
}

if(rank == 0){
    int *updated_flow, *excess, temp_surplus;
    updated_flow = (int *)calloc(num_vertices, sizeof(int));
    excess      = (int *)calloc(num_vertices, sizeof(int));

```

```

// receiving updates from childrens
for (int i = 1; i < num_vertices - 1; ++i){
    MPI_Recv(&updated_flow[0], num_vertices, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    MPI_Recv(&temp_surplus, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
    //printf("From %d the surplus received : %d\n", i, temp_surplus);
    excess[i] = temp_surplus;
    for (int j = 0; j < num_vertices; ++j){

        flow[i][j] = updated_flow[j];

        //printf("flow from %d to %d is :%d\n", j, i+1, flow[j][i+1]);
    }
}

// master copy of flow
printf("Master copy of updated flows : \n");
print_mat(flow);

for (int i = 0; i < num_vertices; ++i)
{
    for (int j = 0; j < num_vertices; ++j)
    {
        if (flow[i][j] != 0)
        {
            flow[j][i] = -flow[i][j];
        }
    }
}
}

```

```

// final updation
push_relabel(capacities, flow, excess, 0, num_vertices - 1);
// Final flows
printf("Master copy of updated flows : \n");
print_mat(flow);
double end = MPI_Wtime();
printf("\n-----\n");
printf("Excution time : %lf\n\n",end-start);

}
MPI_Finalize();

}

```

/* Makefile for parallel code */

CC = mpicc

CCFLAGS = -lm

SOURCE = para_push2.c

OBJS = para_push2.o

HEADERS = para_push.h

TGT = para_push2

all: \$(TGT)

\$(TGT): \$(OBJS)

\$(CC) -o \$(TGT) \$(OBJS) \$(CCFLAGS)

\$(OBJS): \$(SOURCE) \$(HEADERS)

\$(CC) -std=c99 -c \$<

clean:

rm -rf *.o \$(TGT)