International Journal of Emerging Research in Management &Technology
ISSN: 2278-9359 (Volume-2, Issue-6)
Research Article | **June 2013**

Parallelizing Network Flow Algorithm Using Push-Relabel Method

Aditya Gaykar* *Dept. of Computer Engg, Fr.C.R.I.T, Navi Mumbai, Maharashtra, India*
Nivedita Sharma *Dept. of Computer Engg, Fr.C.R.I.T, Navi Mumbai, Maharashtra, India*
Abhijit Bhandarkar *Dept. of Computer Engg, Fr.C.R.I.T, Navi Mumbai, Maharashtra, India*
Amroz Siddiqui *Dept. of Computer Engg, Fr.C.R.I.T, Navi Mumbai, Maharashtra, India*

Abstract-

Maximum flow problem is a fundamental network flow problem. The problem in general is find the maximum amount of flow that is allowed in a network at a particular time. Many sequential algorithms such as Ford-Fulkerson algorithm, Edmonds-Karp algorithm, Goldberg's "generic" maximum flow algorithm, have been designed to solve this problem. In this paper, we present parallel implementation of push-relabel in order to improve the efficiency of solving maximum flow problem on Beowulf cluster architecture. The improved efficiency will help generate quick real time results pertaining to the network traffic and also update the flow sent through different segments of the network.

Keywords— Cluster, MPI, Maximum flow, Push-Relabel

I. INTRODUCTION

The maximum flow problem is a basic graph theory problem which involves finding a feasible and optimal flow through a network having a single source and a single sink. Given a network with arcs and arc capacities, the problem is to find the maximum amount of flow that can be sent from a source vertex to a sink vertex.[1] A flow network is a directed graph G = (V,E) 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 [9]. The flow function must satisfy three constraints:

a) Capacity constraint:
The flow that can be sent from a vertex say u to another vertex say v should be less than or equal to the capacity of the edge connecting the vertices u and v i.e.
$f(u,v) \leq c(u,v)$ for all u,v in $V \times V$

b) Skew symmetry:
The flow from any vertex u to a vertex v is equal to the negative of the flow from vertex v to u i.e.
$f(u,v) = -f(v,u)$ for all u,v in $V \times V$

c) Flow conservation:
Total flow out of a vertex other than the source or sink is 0 i.e.
$\Sigma v \in V\ f(u,v) = 0$ for all u in $V - \{s, t\}$

The flow of the network is the net amount of flow from the source vertex s to sink vertex t . The goal of the solution to maximum flow problem is to determine the maximum amount of flow that can be sent from a source to a destination. Various sequential algorithms are designed to solve this problem which are broadly classified into two categories, Augmenting Path Algorithms: In this the minimum residual capacity along the path is checked and accordingly the flow is sent through it.

Preflow Push-Push Algorithms: Here the network is initially flooded depending on the capacities of edges in it. Then the flow is sent in forward and backward residual edges until the excess obtained at each vertex becomes zero Thus 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. The maximum flow problem is not only important from a theoretical point of view but it also has many important practical applications in resource-allocation in networks and a variety of scheduling problems like airline scheduling, road traffic management etc[5]. This problem can be solved by efficient algorithms which can be sequential or parallel. The first maximum-flow algorithm was proposed by Ford and Fulkerson using the concept of augmenting paths'(An augmenting path is a path from source s to sink t which can be used to increase the flow from source to sink) which finds the augmenting path using depth first search with a time complexity of O(|E|f*)[8], where f* is the maximum flow found by the algorithm. Edmonds and Karp improved upon the algorithm by sending flows across the shortest augmenting paths. They showed that using a breadth-first search in the labelling algorithm and selecting the shortest augmenting path always allows the algorithm to terminate in at most O(nm2).Dinic's

algorithm finds all the shortest augmenting paths in a single step, using "layered networks."Karzanov introduced the concept of preflows and the push operation and gave an O(n3) algorithm. Goldberg and Tarjan designed the push-relabel algorithm with the time bound of O(nmlog n2/m). In 1993, computational experiments confirmed that Goldberg's algorithm was the fastest algorithm in practice. In a later paper by Goldberg and Cherkassky several implementations of the push relabel were studied and their results analysed on a variety of graphs [11].We will discuss this algorithm in detail in later section, as our parallel implementation is based upon this sequential approach. Through our implementation we have tried to parallelize the push-relabel algorithm for maximum flow problem over a cluster. Each node has Ubuntu 12.10 operating system installed. These nodes communicate with each other using Message passing Interface (MPI) library. MPI library needs three things: Secure shell(SSH),Network information Service(NIS) and Network file system(NFS).SSH is used to login from one system to another without requiring passwords. The Network Information Service is a client–server directory service protocol for distributing system configuration data such as user and host names between computers on a computer network. We use NIS so that the user account of our master can be accessed by all slaves in the network. NFS allows a system to share directories and files with others over a network. We use it to remotely login into all machines.

We are doing parallelization because it helps to achieve speedup and thus save a lot of time. Parallel processing is the simultaneous use of more than one CPU to execute a program. Ideally, parallel processing makes a program run faster because there are more processors 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. To achieve parallelization, we need to identify the various dependencies in the algorithm and then try to parallelize those parts of the algorithm which are independent of each other.

II. SERIAL ALGORITHM

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.

Data :
(1) A directed graph G = (V,E) of |V| = n and |E| = m with two distinguished vertices source s and sink t
(2) Each vertex v ∈V has an adjacency list λ(v) which has all outgoing edges outgoing from v
(3) Each edge e = (u,v) ∈ E has a capacity of c(u,v) which is the maximum flow which can be passed through the edge

Initialization:
Set the source label d(s) = n, the sink label to d(t) = 0, and the labels on the remaining vertices to d(v)=0 for all v ∈V −{s, t}.

```
GENERIC-PUSH-RELABEL(G):
        INITIALIZE-PREFLOW(G, s)
        while there exists an applicable push or relabel operation
        do select an applicable push or relabel operation and perform it

INITIALIZE-PREFLOW(G, s):
        for each vertex u ∈ V[G]
        do h[u] ← 0
        e[u] ← 0
        for each edge (u, v) ∈ E[G]
        do f [u, v] ← 0
        f [v, u] ← 0
        h[s] ← |V[G]|
        for each vertex u ∈ Adj[s]
        do f [s, u] ← c(s, u)
        f [u, s]←−c(s, u)
        e[u] ← c(s, u)
        e[s] ← e[s] − c(s, u)

Push(u, v):
        Applicability: u is overflowing, cf(u, v) > 0, and h[u] = h[v] + 1.
        Action: Push df (u, v) = min(e[u], cf (u, v)) units of flow from u to v.
         d f (u, v) ← min(e[u], c f(u, v))
         f [u, v] ← f [u, v] + d f (u, v)
         f [v, u]←−f [u, v]
         e[u] ← e[u] − d f (u, v)
         e[v] ← e[v] + d f (u, v)
```

RELABEL(u):

Applicability: when: u is overflowing and for all v ∈ V such that (u, v) ∈ E f , we have h[u] ≤ h[v].
Action: Increase the height of u. h[u] ← 1 + min {h[v] : (u, v) ∈ Ef }

Result:

The maximum flow f (s, t) which can be routed through the graph i.e. from the source s to the sink t.

III.PROPOSED PARALLEL ALGORITHM

The parallel implementation of push relabel 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 waiting for the 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 then provided as an input to the discharge operation of push-relabel on the master process. The discharge operation for push-relabel is executed at the master itself since it cannot be parallelized. After complete execution of the discharge operation, we get the required maximum flow for the input graph.

PARALLEL PUSH RELABEL :

```
Number of process = Number of vertices of input graph
if ( rank != 0 )  // slave process
{
Receive capacity and reflow matrix from the master process
calculate node excess
for i = 1 to rank - 1
{
        if( capacity[i][rank] != 0 )
                receive flow from ith process
}
for i = rank +1 to sink
{
        if ( capacity[rank][i] != 0 && excess != 0  )
        {
                push_flow = MIN( capacity[rank][i] , excess )
                flow[i] = push_flow
                send push_flow to the ith process
        }
}
send updated flow and excess information to master process
}
else //master process
{
        Read input graph file
        Generate capacity and preflow matrix
        Broadcast capacity and preflow matrix to all the slaves
        for i = 1 to sink
```

```
        {
                Receive flow and excess information from all the vertices except the source
        }
        Generate updates flow matrix and excess array
        DISCHARGE(capacity, flow, excess)
        MAXIMUM FLOW
}
```

The entire operation can be visualized by taking a small network flow graph as shown in Fig. 1,

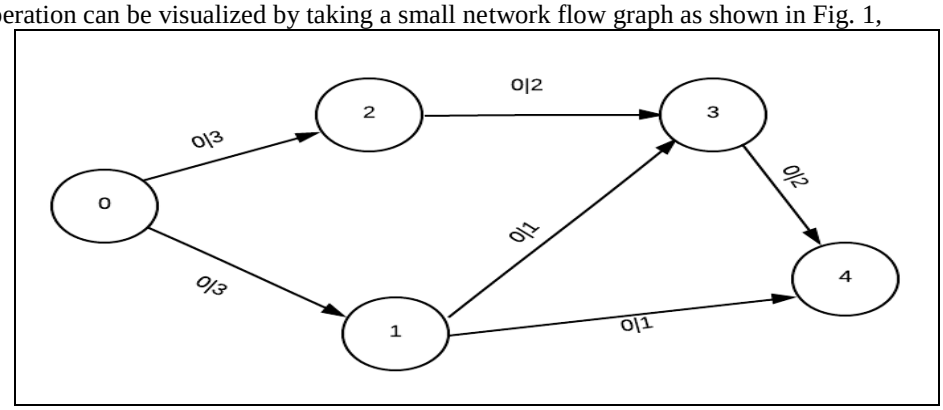

Fig. 1 Sample network flow graph

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. 1 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. 2 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.

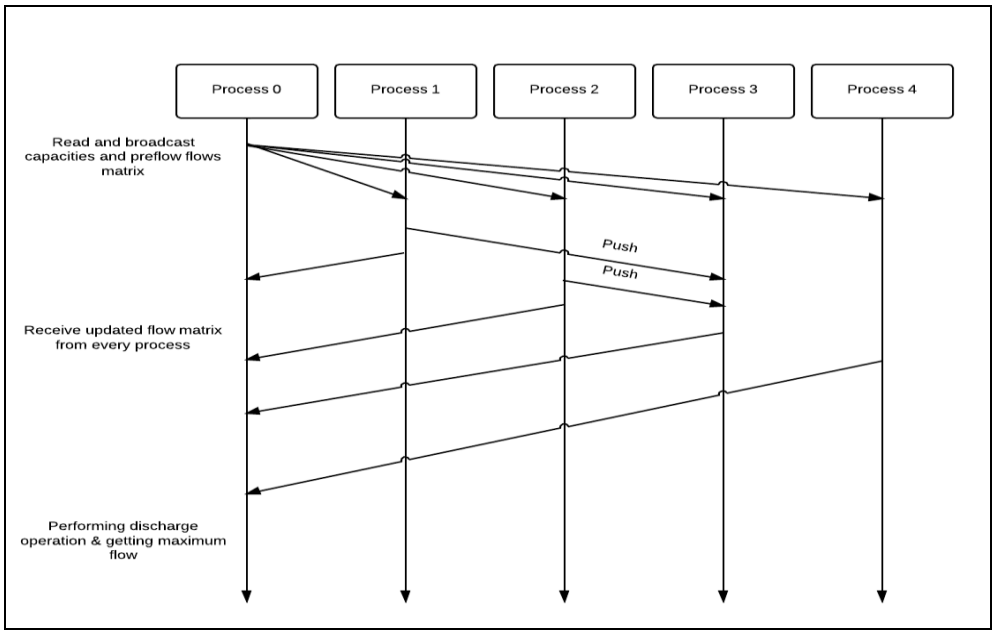

Fig. 2 Time line chart for processes

IV. IMPLEMENTATION

The 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:

a) 1 Master and 4 Slave nodes with Intel P4 processor and 1GB RAM
b) High Speed Ethernet switch
c) KVM Switch

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

a) Network File System (NFS)
b) Network Information Services(NIS)
c) Secure Shell (SSH)
d) Message Passing Interface (MPICH2)

A. 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 programmed by our team to generate random graphs. As shown in the fig. 3. 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.

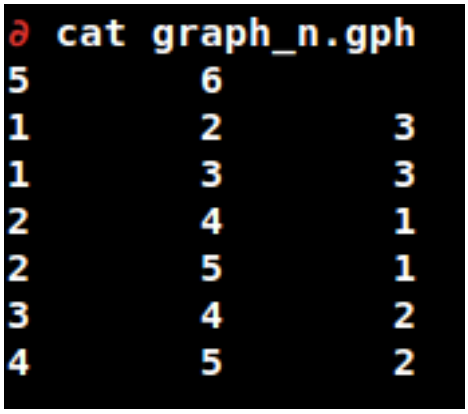

Fig. 3 Input Graph file

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.

B. 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. 4 shows the format of the output file.

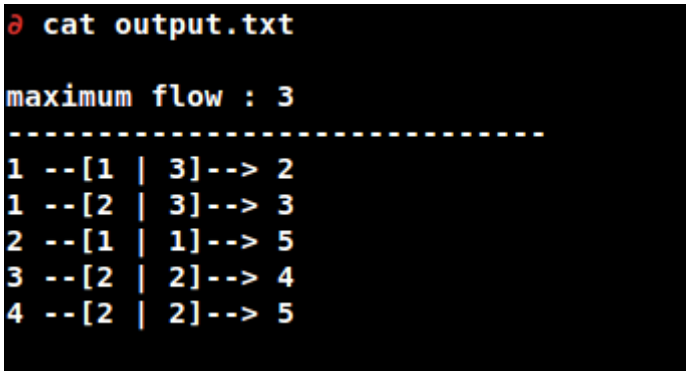

Fig. 4 Output file format

C. 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 Fig. 5.

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

Fig. 5 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**'. Fig. 6 illustrates the execution syntax:

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

Fig. 6 Command for executing parallel code

Here, -n argument specifies the number of processes to be generated -f argument specifies the name of the machine file 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.

V. EXPERIMENT AND RESULT

Table 1: Serial and parallel code execution time comparison

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 2: Serial and parallel code CPU utilization 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

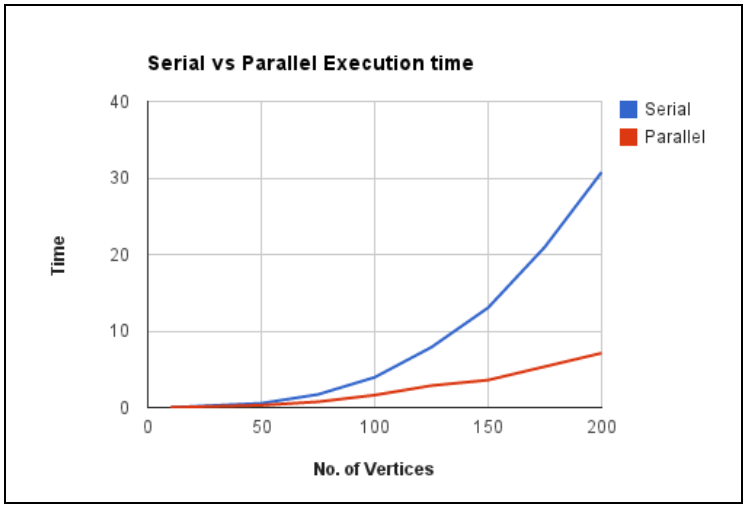

Fig. 7 Serial vs. Parallel execution time

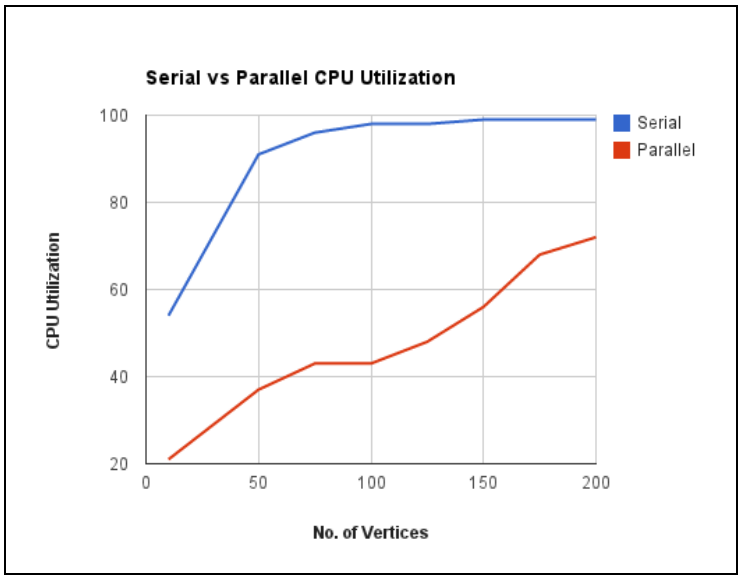

Fig. 8 Serial vs. Parallel CPU Utilization

VI. CONCLUSION

A parallel algorithm for push-relabel method to solve the maximum flow problem 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.

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 Prob-lems," 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, TamalK.. Lecture for CSE 794: Advanced Algorithms, Department of Computer and InformationSciences, 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 MatchingAdvanced Algorithms" (CSE 794) Instructor: Tamal K. DeyScribe: Ted Bulwinkle, April 30, 2009.
[10] R. Anderson and J. Setubal, "On the parallelimplementation of goldberg's maximum flow algorithm,"in 4th Annual Symbosium Parallel Algorithms and Architectures (SPAA92)
[11] San Diego, CA, July 1992, pp.168–177.Hassin, R., Johnson, D.B. 1985: An 0(nlog2n) algorithm for maximum flow in undirected planar network. SIAM J. Comput. 14, 612-624.
[12] Sleator, D.D. 1980: An O(nmlogn) algorithm for maximum network flow. Technical Report STAN-CS-80-831, Department of Computer Science, Stanford University.