Parallelizing Network Flow Using ε-relaxation Algorithm

Kalpesh Adhatrao 2, Abhishek Aswar 3, Aishwarya Iyer 4, Amroz Siddiqui 1
Department of Computer Engineering
Fr. C. Rodrigues Institute of Technology, Vashi
Email- 1 amrozkamal@yahoo.in, 2 kalpesh.adhatrao@gmail.com, 3 abhishekaswar15@gmail.com, 4 aishuiyer2001@yahoo.com

Abstract – Maximum flow problem is a fundamental network flow problem which involves finding a feasible and optimal flow in network consisting of single source and single sink. It has wide ranges of research and commercial applications. Many sequential algorithms such as Ford-Fulkerson algorithm, Edmonds-Karp algorithm, Goldberg’s “generic” maximum flow algorithm, ϵ-relaxation algorithm have been designed to solve this problem. In this paper, we present parallel implementation of ϵ-relaxation algorithm which has been carried out over a cluster consisting of 1 master and 4 slaves nodes in order to study and improve the efficiency of solving maximum flow problem on multi-computer architecture.

Keywords – Maximum flow, Admissible edges, Push list, MPI

I. INTRODUCTION

The maximum flow problem deals with finding of feasible and optimal flow through a single-source, single-sink flow network. A flow network is a directed graph with the arcs carrying flow. Consider a directed graph G(V, E) with |V| = n vertices and |E| = m edges having source vertex ‘s’ and the sink ‘t’. Each edge (u, v) ∈ E has a positive real value capacity c(u, v) ≥ 0. Also if E contains an edge (u, v), then there is no edge (v, u) in the reverse direction. If (u, v) ∉ E then we define c(u, v) = 0.

A flow between any two vertices u and v in the graph is represented as f(u, v) satisfies following constraints [1],

a) **Capacity constraint:** It states that for all u, v ∈ V, the flow between u and v should not exceed the capacity. Thus we require, 0 ≤ f(u, v) ≤ c(u, v).

b) **Flow conservation constraint:** It states that for all u ∈ V – {s, t}, the net flow to a node is zero. Thus we require, \(\sum_{v \in V} f(u, v) = 0\).

c) **Skew symmetry:** It states that the net flow from vertex u to v is equal and opposite of the flow from vertex v to u. Thus we can say that, \(f(u, v) = -f(v, u)\).

The flow of the network is the net amount of flow from the source vertex s which enters into sink vertex t which can be defined as \(|f| = \sum_{(s,v)} f(s,v) = \sum_{(u,t)} f(u,t)\). The goal of maximum flow problem is to determine the maximum possible value for the |f| along with the flow values associated with the each of the edges in the network. Various sequential algorithms are exist to solve maximum flow problem which are broadly classified into two categories [4],

a) **Augmenting Path Algorithms:** Algorithms in this category incrementally add the flow along the paths from s to t depending upon the minimum capacity along that path. After doing that, it searches for the augmenting path in the residual network (a network consisting of edges with unsaturated capacities). Augmenting path is path from s to t which can increase the flow between unsaturated edge (u, v) ∈ E containing in it without violating the capacity constraint.
b) Preflow Push-Push Algorithms: It consists of algorithms which performs initial flooding of the network depending on the capacities of edges in it and then relieving it by sending flow through unsaturated edges in forward towards $t$ and backward towards $s$ until the excess obtained at each vertex becomes zero.

The maximum flow problem was first solved by Ford and Fulkerson [8] using the concept of augmenting path. Edmond and Karp [7] improved the efficiency of Ford-Fulkerson algorithm by proving that it runs in time $O(nm^2)$ if the augmentation of the flow is done along the shortest path between source and the sink. Also the concept of layered network was introduced by Dinic [6] with the help of which shortest path can be found in single step. This concept resulted into an algorithm with the running time of $O(nm)$. This algorithm was again improved by Karzanov [2] to $O(n^2)$. He introduced concept of preflow in layered network. A preflow is similar to a flow except amount of flow going inside the node may exceed the amount coming out of it. On the basis of the concept of preflow Goldberg and Tarjan [3] designed push-relabel algorithm which was considered as one of the fastest algorithm to solve the maximum flow problem with the running time of $O(nm \log \frac{n^2}{m})$. In an effort to ease the implementation on parallel or distributed environment Dimitri P. Bertsekas [5] designed $\epsilon$-Relaxation algorithm which has a serial complexity of $O(n^2 \log n)$. This algorithm can be easily implemented on distributed system and can provide a sufficient speedup.

There are many theoretical as well as practical parallel implementations of these algorithms. The very first practical parallel implementation was given by Aderson and Setubal [9] for push-relabel algorithm which makes the use of global relabeling operation to make the algorithm to work in the parallel environment efficiently.

In this paper we present parallel version for $\epsilon$-relaxation algorithm for maximum flow problem which has been tested over a cluster consisting of 5 nodes (1 master and 4 slaves) interconnected with high speed switch. We have used Open MPI which is an open source, thread-safe MPI (Message passing interface) implementation to allow the communication between processes executing in parallel on different node.

II. $\epsilon$-RELAXATION ALGORITHM

Consider a flow network given by directed graph $G(V, E)$ possessing capacities on each edges which is the maximum flow possible on that edge. Thus each edge $(i, j) \in E$ has flow bound with the minimum flow as $b_{ij} = 0$ and the maximum flow $c_{ij}$ which is capacity of that edge.

A. Data structures used in the algorithm

a. The admissible graph

Operating on a particular vertex in the flow network, the algorithm can change the flow of two kinds of arcs: outgoing $\epsilon^+$ balanced arc $(i, j)$ with $f_{ij} < c_{ij}$ and incoming $\epsilon^-$ balanced arc $(j, i)$ with $f_{ji} > b_{ji}$. These two types of arcs are called as admissible arcs. An admissible graph is one which contains these admissible arcs for a particular vertex.

b. Push List

It is a data structure used to store the admissible graph. Thus for a particular node in the flow network, its push list contains the list of all the admissible arcs whose flow can be changed in order to reduce the surplus i.e. difference between the incoming and outgoing flow present on it.

The algorithm starts with the creation push list data structure for all the nodes present in the flow network except the source and sink. This push list contains the admissible edges i.e. $\epsilon^+$ balanced arcs and $\epsilon^-$ balanced arcs for a particular vertex in the flow network.

Once this initialization is done, preflow operation is performed by the source which sends initial flow to all the vertices connected to the source vertex. The magnitude of this flow equals to the capacity of the edges connecting source and the neighboring vertices. This preflow operation is followed by the calculation of the surplus $g$ present on every intermediate node $i \in V - \{s, t\}$.

After that this method repeatedly selects intermediate node $i \in V - \{s, t\}$ having positive surplus i.e. $g_i > 0$ and start relaxing the surplus present on it. This operation called as up iteration. This process will eventually drive all
Parallelizing Network Flow Using $\epsilon$-relaxation Algorithm

the nodes surpluses to zero. Once this is done the algorithm terminates giving maximum flow as sum of all the flows from the vertices connected to the sink i.e $\Sigma_{j \in V} f_{jr}$.

B. Steps involved in Up Iteration

**Stage 1:** Taking an arc from the top of node i’s push list, if $g_i > 0$ and the arc is outgoing arc $(i,j)$ then go to Step 2. If $g_i > 0$ and the arc is incoming arc $(j,i)$ then go to Step 3. If there is no arc in the push list then go to Step 4. If an arc was found but the surplus $g_i = 0$ then stop.

**Step 2:** It deals with decreasing the surplus $g_i$ by increasing the outgoing flow $f_{ij}$ which can be give as,

$$f_{ij} = f_{ij} + \delta$$

$$g_i = g_i - \delta$$

$$g_j = g_j + \delta$$

Where $\delta = \min \{ \epsilon, c_{ij} - f_{ij} \}$. If $\delta = f_{ij} - c_{ij}$, delete arc $(i, j)$ from i’s push list. Go to step 1.

**Step 3:** It deals with decreasing the surplus $g_i$ by decreasing the incoming flow $f_{ji}$ which can be give as,

$$f_{ji} = f_{ji} - \delta$$

$$g_i = g_i + \delta$$

$$g_j = g_j - \delta$$

Where $\delta = \min \{ \epsilon, c_{ji} - f_{ji} \}$. If $\delta = f_{ji} - c_{ji}$, delete arc $(j, i)$ from i’s push list. Go to step 1.

**Step 4:** construct new push list for i, containing only those arcs which are admissible. Go to Step 1.

The up iteration is carried out till the surplus on every intermediate vertex present in the graph becomes zero. Once getting zero surpluses on every intermediate vertex, the algorithm terminates giving the maximum flow at the sink.

III. PROPOSED PARALLEL DESIGN

Parallel version of $\epsilon$ - Relaxation algorithm has been designed by taking into consideration of its parallel execution over a Linux cluster using Message Passing Interface (MPI). The basic concept of this parallel design is to assign a separate process to each vertex in the flow network which will handle the operation related to that vertex. Thus while executing the parallel code, number of processes to be generated should be made equal to the number of vertices present in the input flow network. There are three stages in this implementation.

**Stage 1:** Initialization and distribution of jobs by the master process.

During execution at the stage of MPI initialization, the MPI interface assigns a unique rank to the each of the process generated. The operation related to the source vertex is handled by master process i.e. process with rank 0. This master process performs the tasks of reading input graph file and then generates the capacity matrix. The capacity matrix is $n \times n$ matrix where $n$ is the number of vertices in the flow network. For edge $(i, j)$, its capacity is given by the value in the capacity matrix in $i^{th}$ row and $j^{th}$ column. In the same way the master process creates the flow matrix of dimension $n \times n$ and initialized it with 0 since there is no flow in the graph initially. It then performs preflow operation and then updates flow matrix accordingly. These capacity and flow matrix is then broadcasted by the master process to the remaining processes.

**Stage 2:** Parallel delta push operation performed by the remaining processes.

After receiving the capacity and the flow matrix from the master process, the process operating on an intermediate vertex first generates the push list for that vertex. Considering the incoming arcs in the push list, the process gets information about vertices from which the vertex under consideration is supposed to receive the flow. If these flows are not received then the process on that vertex will wait for their reception. But since there are processes for vertices in the network connected to the source who have already received flow through initial preflow operation, they start pushing that flow in parallel through the outgoing arcs containing in the push list. In this way waiting processes on reception of these flows start pushing it in parallel to processes for the vertices adjacent to them.

**Stage 3:** Gathering the updated flows and up-iteration by master process.
Once this push operation is performed, the processes send the updated flows for their respective vertices to the master process. Master gathers these updated flows from all remaining processes and updates its local flow matrix. This updated flow matrix for the network is calculated and is provided as an input to the up-iteration operation performed by the master process. The up-iteration operation at the master process calculates the surpluses at every intermediate vertex. It then relaxes these surpluses by pushing the flows through the admissible edges containing in the push list associated with that particular vertex. Once the master process gets all the surpluses at the intermediate vertices as zero, the execution terminates giving the maximum flow.

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

```plaintext
Number of process = Number of vertices of input graph 
if ( rank != 0 ) { // slave process
    Receive capacity and preflow matrix from the master process 
    Calculate node surplus 
    for i = 1 to rank - 1{
        if ( capacity[i][rank] != 0 )
            receive flow from i\textsuperscript{th} process
    }
    for i = rank +1 to sink{
        if ( capacity[rank][i] != 0 && surplus != 0 ){
            delta_push = MIN( capacity[rank][i] , surplus )
            flow[i] = delta_push
            surplus = surplus – delta_push
            send delta_push to the ith process
        }
    }
    send updated flow information to master process
}
else { //master process
    Read input graph file
    Generate capacity and flow matrix
    Perform preflow
    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 surplus array
    UP-ITERATION(capacity, flow, surplus)
}

Algorithm: Parallel $\epsilon$ - Relaxation
```

The entire operation can be visualized by taking a small flow network as shown in fig. 1,
Parallelizing Network Flow Using $\epsilon$-relaxation Algorithm

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 1 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 2, 3, 4, 5 will be processed by the processes with rank 1, 2, 3, 4 respectively.

It can be seen in fig. 1 the processes for vertices 2 and 3 which are receiving the flow from master can send the flow in parallel through the admissible edges containing in their push list while the process on vertex 4 will wait for the processes on vertices 2 and 3 to complete their delta push operation and once it has been done process on vertex 4 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 the job of up-iteration to relax the flow until surplus value of all vertices becomes 0 and the execution terminates giving maximum flow.

IV. IMPLEMENTATION

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. 1 Master and 4 Slave nodes with Intel P4 processor and 1GB RAM
2. High Speed Ethernet switch to provide connection between the nodes
3. KVM Switch to access slaves nodes separately from single system.

All nodes (including the master node) run the following services:
1. Network File System (NFS)
2. Network Information Services (NIS)
3. Secure Shell (SSH)
4. 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. 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.

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.
Parallelizing Network Flow Using \(\epsilon\)-relaxation Algorithm

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 the following fig.5,

![Fig. 5: Commands for compiling the parallel code](image)

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.6 illustrates the execution syntax,

![Fig. 6: Command for executing parallel code](image)

Here,
- \(-n\) argument specifies the number of processes to be generated
- \(-f\) argument takes 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.

D. Basic functions of MPI library used in the implementation

1. MPI_Send()

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

   Here,
   - **buf**: starting address of data buffer to be sent
   - **count**: number of elements in data buffer
   - **datatype**: datatype of each element in data buffer
   - **receiver**: rank of receiver process
   - **tag**: tag id of the message
   - **comm**: communicator

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

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

Here,
- **buf**: starting address of data buffer in which sent data to be received
- **count**: maximum number of elements in data buffer
- **datatype**: datatype of each element data buffer
- **sender**: rank of sender process
- **tag**: tag id of the message
- **comm**: communicator

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

3. MPI_Bcast()

```c
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int broadcastRoot, MPI_Comm comm)
```

Here,
- **buf**: starting address of data buffer
- **count**: number of elements in data buffer
- **datatype**: data type of each element in data buffer
- **broadcastRoot**: rank of the message broadcasting process
- **comm**: communicator

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 processes. It is again a blocking routine.

V. EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>Sr.no</th>
<th>Number of vertices</th>
<th>Serial Execution Time</th>
<th>Parallel Execution Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>0.0080</td>
<td>0.0071</td>
<td>1.1268</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>0.3480</td>
<td>0.2090</td>
<td>1.6650</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>0.4490</td>
<td>0.4367</td>
<td>1.0281</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>2.1550</td>
<td>1.4164</td>
<td>1.5215</td>
</tr>
<tr>
<td>5</td>
<td>125</td>
<td>3.9500</td>
<td>2.5039</td>
<td>1.5773</td>
</tr>
<tr>
<td>6</td>
<td>150</td>
<td>6.9470</td>
<td>2.9427</td>
<td>2.3608</td>
</tr>
<tr>
<td>7</td>
<td>175</td>
<td>10.9570</td>
<td>4.1619</td>
<td>2.6327</td>
</tr>
<tr>
<td>8</td>
<td>200</td>
<td>16.0800</td>
<td>4.8083</td>
<td>3.3442</td>
</tr>
</tbody>
</table>

**Table 1: Serial and parallel code execution time comparison**

<table>
<thead>
<tr>
<th>Sr.no</th>
<th>Number of vertices</th>
<th>Serial CPU Utilization</th>
<th>Parallel CPU Utilization</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>52</td>
<td>15</td>
<td>3.4667</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
<td>79</td>
<td>34</td>
<td>2.3235</td>
</tr>
<tr>
<td>3</td>
<td>75</td>
<td>91</td>
<td>40</td>
<td>2.2750</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>94</td>
<td>49</td>
<td>1.9180</td>
</tr>
<tr>
<td>5</td>
<td>125</td>
<td>93</td>
<td>48</td>
<td>1.9375</td>
</tr>
<tr>
<td>6</td>
<td>150</td>
<td>97</td>
<td>53</td>
<td>1.8302</td>
</tr>
<tr>
<td>7</td>
<td>175</td>
<td>98</td>
<td>63</td>
<td>1.5556</td>
</tr>
<tr>
<td>8</td>
<td>200</td>
<td>95</td>
<td>66</td>
<td>1.4394</td>
</tr>
</tbody>
</table>

**Table 2: Serial and parallel code CPU utilization comparison**
Parallelizing Network Flow Using $\epsilon$-relaxation Algorithm

VI. CONCLUSION

A parallel algorithm for $\epsilon$-relaxation method to solve 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 in 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 bottleneck situations of over jamming. Even applications in traffic management systems and water management systems can be implemented effectively.

VII. ACKNOWLEDGEMENT

We wish to thank Prof. Dimitri P. Bertsekas (professor at the department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology) for his suggestions regarding resources to understand $\epsilon$-relaxation algorithm. 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 paper.

REFERENCES