*Asynchronous Multi-Threaded Implementation of Push Relabel algorithm and its performance analysis on Intel Xeon architecture*

Sharmila Soundararajan

# Department of Electrical and Computer Engineering,

# University Of North Carolina at Charlotte

United States of America

ssounda3@uncc.edu

***Abstract*—In this paper, I implemented an asynchronous multithreaded algorithm for a network to find its maximum flow. The push relabel technique is used, which is essentially sequential and requires lot of locks and barriers if has to be implemented in a parallel way. This asynchronous multithreaded implementation is designed in such a way that it does not require any locks or barriers and the multiple threads can work independently of each other. Each thread determines its own termination after periodically checking the termination condition, which is common for all threads. This algorithm provides a good speed up and response time compared to serial algorithm for dense and medium density graphs due to its lock-free nature.**

***Keywords—push relabel, maximum flow, multi-threaded, pthreads, asynchronous , lock – free.***

# I. INTRODUCTION

The maximum flow problem is a primary graph theory problem with key applications in many areas. In graph theory, a flow network is a directed graph where every edge is associated with a capacity and a flow. Flow networks can be employed to model information through computer networks, commodities through highways, etc. Other applications include computer vision. One of the asymptotically fastest maximum-flow algorithm to date is push relabel algorithm. The push-relabel methods also efficiently solve other flow problems like minimum-cost flow problem.

The max-flow problem is stated as follows: A flow network is a graph G (V, E), where edge (u, v) has capacity cuv. If an edge (u, v) Є E, cuv is set to 0. A graph G has a source s and a sink t. A flow in a network is the net amount of flow sent from source, s to sink, t. A flow in a network should satisfy the following constraints:

1. 1. Capacity Constraint: f (u, v) ≤ Cuv, for u, v є V
2. 2. Relaxation of flow Conservation: Σ f (v, u) – Σ f (u, v) ≥ 0 for all v є V and u є V-{s}

The maximum-flow problem searches for the flow with the maximum value in the given network. In other words, it finds the maximum possible flow in a given network.

Both serial and parallel algorithms have been examined in this paper. Though there are approximation algorithms to predict an approximate maximum flow, my work focuses on multicore platforms and searches for the exact solution. In this paper, I have implemented the asynchronous multi-threaded maximum-flow algorithm and studied its performance on Intel Xeon architecture and compared its performance with the serial implementation of Push – Relabel algorithm by Karzanov. Although the algorithm uses the push - relabel technique proposed by A.V. Goldberg and R.E. Tarjan [2] , it is unique and faster from the existing algorithm in a way that it is totally free from the use of locks and barriers for synchronization purposes. Each thread can terminate independently of each other when the required condition is satisfied. When the condition is satisfied and all the threads terminate, the flow obtained is the maximum-flow. The maximum number of operations to find the maximum-flow is O (|V|2|E|).

I implemented the algorithm using pthread library and compared its performance with the sequential algorithm in [8]. Graphs with varying densities and sizes were tested. The results shows that the parallel algorithm has a speed of 30 % over the serial algorithm and that the number of threads and the architecture of the machine determines the speed up which is discussed in detail in Section 4.

## A. Organization of paper:

The rest of the paper is organized as follows: Section 2 presents the related work done for the implementation and study of asynchronous push relabel algorithm on Intel Xeon architecture. Section 3 describes push relabel algorithm in general and the various terminologies involved in understanding it. Section 4 presents the pseudo code of asynchronous multi-threaded push relabel algorithm and the details of programming implementation of this algorithm. Section 5 describes the hardware specification of the architecture on which the asynchronous algorithm is tested. The experimental results have been tabulated and plotted in Section 6 followed by the detailed analysis and discussion in Section 7. Conclusion and future work are provided in the Section 8.

# II. RELATED WORK

## A. Journals and Papers Referred

Various papers and journals were referred for understanding the significance of push relabel algorithm and its various forms of implementations during the years. The order of execution time and time complexity has been improved continuously in the consecutive modifications of the algorithm. The complexity bound was further improved by using specially designed data structures.

## B. Background study

Paper 1: The asynchronous multithreaded algorithm [1] is implemented with atomic instruction functionality which replaces the necessity of locks to access the shared variables or data structures. The correctness and complexity bound of the algorithm is proved in paper [1].

Paper 2: The supplementary file [2] of [1] examines the interleaved execution of the Push and Lift operations. Since the stage stepping stages, explained in Section 3, can make an edge special residual edge upon algorithm termination, a corrective measure is examined which is the part of the algorithm itself.

# III. ALGORITHM AND NOTATIONS

## A. Notations used in the algorithm

Some notations used in the pseudo code are explained here.

I consider a directed graph G (V, E) where V is the number of vertices in the graph/network and E is the number of edges. Density of the graph is defined by,

Density, D = E/(V\*(V-1))

The function f of a directed graph G(V, E) is called a flow, if it satisfies the flow constraint and capacity constraint explained in section 1.

Given a graph G (V, E) and flow f, the residual *capacity cf (u, v) is given by cuv* – f(u,v), and the residual network of G in*duced by f is Gf (V, Ef), where Ef = (u, v) where u €V and cf* (u,v)>0}. Thus, one of the conditio*n for edge (u, v) to belong to Ef* is that the residual capacity should be greater than 0. In other words, an edge (u,v) belongs to a residual network if,

1. *Residual capacity, cf* (u,v) is greater than zero.
2. Vertices u€V and v€V.

Excess e of a vertex u € V, e(u) is defined as the amount by which the flow in exceeds the flow out. A vertex is said to be overflowing if its excess value is greater than zero i.e., e(u)>0.

When overflowing vertices exist we call f as a pre*-flow. At the end of the algorithm, pre-flow is the maximum flow.* A height function h(u) for every vertex u €V is a valid he*ight function if edge (u, v) € Ef* a regular residual edge, if h(u) ≤ h(v)+1. An edge is called a regular residual edge is h(u) ≤ h(v)+1. If h(u) > h(v)+1, (u, v) is called a special residual edge. According to this asynchronous algorithm, once an edge (b, a) becomes a special residual edge, push(b,a) becomes available immediately and hence upon completion of push(b,a), the special residual edge (b, *a) will be removed from Ef* (Edges of residual graph).

## B. Asynchronous Max-Flow Algorithm:

The Asynchronous Max-flow Algorithm steps:

1. Initialize h(u), e(u) and f (u, v)
2. while e(s) + e (t) < 0 do
3. execute applicable push or lift operations asynchronously
4. end while

## C. Operations in the Algorithm

The initialize, push, and lift operations are defined as follows:

Initialize h (u), e (u) and f (u, v):

* + 1. h (s) ← |V|
    2. e (s) ← |0|
  1. for each u € V – {s}
     1. h (u) ← 0
     2. e (u) ← 0
  2. for each (u,v) € E where u ≠ s and v ≠ s
     1. f (u, v) ← 0
     2. f (v, u) ← 0
  3. for each (s, u) € E
     1. f (s, u) ← Csu
     2. f (u, s) ← - Csu
     3. e (u) ← Csu
     4. e (s) ← e (s) – Csu

Excess of s will be the negative of total capacity of edges incident on s

* Push (u, v’): applies if u is overflowing, and smallest height v € V s.t. (u, v) € Ef and h (u) > h (v).

v’ = argminv [h(v)| cf (u, v) > 0 and h (u) > h (v)]

d ← min [e (u), cf ( u, v’)]

f (u, v’) ← f (u, v’) + d # executed atomically

f (v’, u) ← f (v’, u) – d # executed atomically

e (u) ← e (u) – d # executed atomically

e (v’) ← e (v’) + d # executed atomically

* Lift(u): applies if u is overflowing, and h(u) ≤ h(v) for all (u, v) € Ef,

h(u) ← min{h(v)| cf (u, v) > 0} + 1

The above algorithm is referred from [1].

### 1. Push Operation:

Vertex height determines how flow is pushed; In this algorithm, push operation pushes flow only downhill, that is, from a vertex of higher height to a vertex of lower height. The height of the source is fixed at |V| and the height of the sink is fixed at 0. The heights of the source and sink remains a constant throughout the execution of the algorithm. All other vertex heights start at 0 and increase with time.

### 2. Relabel Operation:

When the heights of the neighboring vertices of the overflowing vertex u, are at the same level as u or below, the height of the vertex u has to be increased to push the excess from the overflowing vertex u to the neighboring vertex. This operation of increasing the height of the vertex u with respective to the heights of its neighbors is called “relabeling”. It is done when the vertex is overflowing i.e., when excess is greater than 0. The height of the vertex is increased to one unit more than the height of the lowest of its neighbors to which it has an unsaturated pipe. After a vertex is relabeled, it has at least one outgoing pipe through which more flow can be pushed.

To make the pre-flow a legal flow, the algorithm sends the excess of the overflowing vertices back to the source by relabeling the heights of vertices to above the fixed height |V| of the source. Program Implementation of asynchronous multithreaded push relabel Algorithm with Vertex exchange:

## D. Modifications in the asynchronous algorithm:

In comparison to the original push relabel algorithm, asynchronous multi-threaded algorithm differ in the following ways.

1. Push operation pushes flow to the lowest neighbor in the residual graph, Gf where as in the original push relabel algorithm flow can be pushed to any lower neighbor.
2. The termination of the algorithm depends on the condition e (s) + e (t) = 0 rather than the absence of overflowing vertices.

These modifications allow the algorithm to be executed asynchronously by multiple threads. The algorithm is designed in such a way that each thread can execute independent of each other and there are no synchronization barrier required at any point. Each thread can terminate on its own upon a condition. This helps in obtaining a very high performance improvement and speedup compared to sequential algorithm.

## E. Correctness and Complexity bound of the algorithm

### 1. Correctness:

Since the threads are completely asynchronous in its working, there can be different ways of interleaving of push and relabel operations of vertices by threads. The order in which instructions from the threads are executed in real time due to thread interleaving is called a trace.

A trace has two stages: preparation stage and action stage. Preparation stage checks if the push or relabel operation is possible and action stage does the push or relabel operation i.e., updates the shared variables.

There are two types of traces:

1. A stage-clean trace: where several operations are independent of each other i.e., there are no overlapping in their executions. For eg., stage clean trace of a push and lift operation means that push or lift has completed in its entirety before the other one starts.

2. A stage-stepping trace: where every operation finishes its preparation stage and are ready to proceed with the action stage.

Only time when there could be an inconsistency in the availability of edges in the residual graph is when there is a stage-stepping trace due to push (u, v) and lift (v) and (b, a) € E before the action stage of push (u, v). This special case of interleaving might make edge (b, a) to become a special residual edge. But according to the algorithm since e (v) > 0, (v, u) € Ef and h (v) > h (u) + 1 after the trace, push (v, u) becomes immediately available and hence will remove push (v, u) from residual graph, Ef.

### 2. Complexity:

According to the “paper”, given the graph G (V, E) with source s and sink t, the algorithm finds maximum flow with at most O(|V|2|E|) push/lift operations.

# IV. PROGRAMMING IMPLEMENTATION OF THE ALGORITHM

## A. Data Structures Used:

Vectors and queues are the primary data structures used for dynamically allocating memory. Vector of vector helps in adjacency list implementation of FIFO push relabel maximum flow algorithm. During initialization, the neighbors of Source vertex s are sent the maximum of its capacity and the edges thus saturated initially. Hence all the neighboring vertices of s are overflowing and hence the overflowing vertices are given to the threads equally. Every thread has a local queue in which the overflowing vertices are stored and processed from. New overflowing vertices obtained due to the push operation on any of the vertices in the local queue are pushed back to the same local queue. This supports cache affinity.

Since memory allocation of the vertices and edges is important for the cache performance, the variables associated with every vertex (height, excess) are in a single C structure and hence allocated continuously in memory for improved locality. Due to the adjacency list implementation of edges, edges leaving the same vertex are allocated contiguously in memory. Since the push operation in the algorithm has to search for the lowest neighbor during its preparation stage, this type of memory allocation for edges improves cache efficiency.

## B. Initializing pre-flow:

The source node sends as much flow as possible downhill from the source to the sink. The amount source sends to its neighbors is exactly equal to the capacity of the outgoing pipe. Hence all the neighboring nodes of s are overflowing after initialization and their excess value is equal to the capacity of the edge connecting s to the neighbor node.

## C. Atomic Fetch-And-Add instructions:

The updates to shared variables c (u, v) and e(u) have to be done atomically by all threads so that no interleaving happens in between the update. For this purpose atomic fetch-and-add instructions are used in the push operation to update the heights and excess of nodes u and v of edge (u, v). A copy constructor is used to declare the excess vector and capacity variable atomic.

The gcc 4.4 enables support for atomic instructions C++0x by adding the command-line parameter -std=c++0x to the g++ command line.

## D. Graph inputs:

A list of weighted edges are generated using a graph generation algorithm found at [7]. The type of graph, number of nodes and edges and maximum capacity of the edges in the graph are given as input. The edge list is written to a text file and it is read from the main file. After which the graph is generated in a vector of vector data structure in an adjacency list structure.

## *E. Graph generation in the Program:*

Graph generation is done with the vector of vector data structure and edge data structure having three inputs: from, to and capacity. Each entry in the vector of G is an edge added according to the adjacency list implementation. The graph generation algorithm referred is given in [7].

# V. HARDWARE SPECIFICATION

Intel Xeon quad core architecture supports atomic operations. Since the performance of the algorithm should be studied on a multicore Symmetric Multiple Processor system, Intel Xeon has been chosen. It has three levels of cache and 12 MB L3 cache is shared between the cores. The Intel platform has four core X5672 processors running at 3.2 GHz with 12 MB L3 shared cache.

# VI. RESULTS

The following algorithms were used for experiments:

1. Asynchronous multi-threaded push relabel algorithm (in short “amf”) was implemented. Local queue of each thread follows FIFO ordering.
2. Serial push relabel algorithm was tested with various graph inputs.

The Intel Xeon quad core architecture supports 8 threads. Results are based upon response time and speed up when compared with serial implementation.

I compared the performance of two algorithms on three types of acyclic graphs. Sparse graph whose density is less than 0.05, a random directed graph with density value in between 0.3 to 0.7 and a dense graph with density value as close as 0.98. Execution result reported for each algorithm was averaged over 15 runs. Response time does not describe every aspect of a parallel push-relabel algorithm. Number of push, relabel operations needed, depends on density of input graph, weights associated with each edge and also the vertex ordering scheme. Though the complexity bound, number of push relabel operations executed by sequential algorithm is lesser compared to parallel algorithm, due to the asynchronous and multi-threaded nature of the algorithm, the amf (asynchronous multi-threaded algorithm) is able to outperform the serial in general and it is consistent when the graph is a dense graph as shown in Table 1. Larger and denser the graph becomes, asynchronous outperforms the serial algorithm with a speedup of 2.3 as shown in Table 5. This is due to the fact that the parallel threads in amf execute asynchronously meaning there are synchronization needed between threads. Each thread knows when to terminate and hence the threads are fully independent. All the threads work on the same graph and they have shared variables like height h (u), excess e (u) and capacity of edge (u, v). Hence the update to the shared variables have to be protected which is achieved through atomicity the architecture supports. The atomic instruction fetch-and-add helped in achieving the atomic update of shared variables.

Table 1. Response time of amf algorithm for a dense graph of D = 0.98 and N = 100

|  |  |
| --- | --- |
| No Of threads | Response time |
| 2 | 0.403806 |
| 4 | 0.128093 |
| 8 | 0.196928 |
| 16 | 4.803705 |

Table 2. Response time of amf algorithm for a dense graph of number of nodes, N=500 and varying densities

|  |  |  |  |
| --- | --- | --- | --- |
| N | D = 0.1 | D = 0.3 | D = 0.8 |
| 2 | 0.045508 | 10.2156 | 36.587 |
| 4 | 2.255065 | 6.2158 | 22.015 |
| 8 | 0.882910 | 4.69728 | 15.858 |
| 16 | 7.112848 | 17.4229 | 40.2142 |

Table2 shows that the execution time of 16 threads is the maximum. This is because during initialization there might be more number of threads than the available overflowing vertices. This might leave few threads with no task and also the cost of context switching between the threads becomes a overhead for small number of graphs. But as the size of graph, the number of vertices as well as number of edges grows, the performance tends to improve i.e., reduced execution time. The comparison between a parallel implementation of push relabel algorithm and asynchronous implementation would show a higher speed up as there will be a lot of locks used in the parallel implementation.

Table 3. Response time of (asynchronous) amf algorithm with varying Maximum Capacity of edges (Max C) for N = 1000 and D = 0.3,

|  |  |  |
| --- | --- | --- |
| No. Of Threads | Response Time (Max C = 150) | Response Time (Max C = 450) |
| 4 | 29.458 | 0.45 |
| 8 | 25.423 | 0.42431 |

Table 3 shows that the response time is very much dependent on the type of graph and also with the capacity of the edges. The amf tends to perform well when the capacity of edges is larger because the number of push and relabel operations are reduced.

Table 4. Response time of amf algorithm for sparse graph for N = 2000, D = 0.0005

|  |  |
| --- | --- |
| No. Of Threads | Response time |
| 2 | 149.15565 |
| 4 | 109.41425 |
| 8 | 60.5148 |

Table 4 shows the response time of amf algorithm for sparse graph. The amf algorithm performs badly when the input graph type is sparse, irrespective of the number of edges. A worst case measurement of sparse graph response time is tabulated here. Number of nodes, N is considered as 2000 and D = 0.0005 which means the graph has the number of edges equivalent to the nodes. Many number of threads will be idle during sparse graph processing.

Table 5. Comparison between amf algorithm and serial algorithm for N = 500

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | Rserial | Rasync | Speedup |
| 2 | 0.01846 | 0.04550 | 0.40584 |
| 4 | 2.43882 | 2.25506 | 1.08148 |
| 8 | 1.23059 | 0.88291 | 1.39379 |
| 16 | 1.19856 | 7.11284 | 0.16850 |

Table5 shows that the speedup is maximum when the number of threads is 8. This attributes to the fact that meaningful acceleration on a multicore system is possible when the number of threads is equal to the number of hardware contexts the Intel architecture supports. Intel Xeon X5672 supports 8 threads.

Fig. 1 Response time variations due to varying graph densities.

In Fig. 1, we see that response time of thread 8 is the lowest for various densities. The response time of thread 16 is always higher due to the cost of context switching.

Table 6: Comparison between amf algorithm and serial algorithm for

N = 1000, D = 0.3

|  |  |  |  |
| --- | --- | --- | --- |
| Threads | Rserial | Rasync | Speedup |
| 2 | 22.39 | 34.24 | 0.65 |
| 4 | 43.9249 | 52.02 | 0.84 |
| 8 | 138.025 | 67.70 | 2.03 |
| 16 | 0.09226 | 1.38 | 0.06685 |

Table 6 shows that parallelization efficiency of afm (asynchronous multi-threaded algorithm) is affected by the number of vertices that overflow at a given point of time. Speedup due to synchronous algorithm is not always higher than the serial because asynchronous algorithm has more number of push and relabel operations performed than the serial algorithm. More number of pushes and relabeling operations are due to asynchronous and independent nature of threads. Speedup is below 1 when number of threads, T = 2 is due to the fact that with two threads and very large number of overflowing vertices at a time, the serial is faster due to its simplicity and lesser number of operations. When T = 16, there are more number of context switches and very large number of independent push and relabel operations.

# VII. FURTHER RESEARCH

Global Relabeling and Gap Relabeling heuristics are expected to increase the speed up multifold, which I would take up as a part of my future research paper.

# VIII.CONCLUSION

Push relabel algorithm is considered one of the fastest algorithms among the maximum flow algorithms. Push relabel technique when modified to run in a parallel, asynchronous way has a much better response time and is scalable across multi core platforms.

##### ACKNOWLEDGMENT

I would like to thank my Professor Dr. Bharat Joshi in giving great insights into the concepts and for guiding me in the project throughout. I would also like to extend my gratitude to my committee members Dr. Conrad and Dr. Nasipuri for their valuable time and feedback.

##### REFERENCES

1. Bo Hong and Zhengyu He, “An Asynchronous Multithreaded Algorithm for the Maximum Network Flow Problem with Nonblocking Global Relabeling Heuristic,” IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 22, NO. 6, JUNE 2011.
2. A.V. Goldberg and R.E. Tarjan, “A New Approach to the Maximum Flow Problem,” Proc. 18th Ann. ACM Symp. Theory of Computing (STOC ’86), pp. 136-146, 1986.
3. A.V. Goldberg, “Recent Developments in Maximum Flow Algorithms (Invited Lecture),” Proc. Sixth Scandinavian Workshop Algorithm Theory (SWAT ’98), pp. 1-10, 1998.
4. D. Bader and V. Sachdeva, “A Cache-Aware Parallel Implementation of the Push-Relabel Network Flow Algorithm and Experimental Evaluation of the Gap Relabeling Heuristic,” Proc. 18th ISCA Int’l Conf. Parallel and Distributed Computing System (PDCS ’05), 2005.
5. R.J. Anderson and J.C. Setubal, “On the Parallel Implementation of Goldberg’s Maximum Flow Algorithm,” Proc. Fourth Ann. ACM Symp. Parallel Algorithms and Architectures (SPAA ’92), pp. 168-177, 1992.
6. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and Clifford Stein, Introduction to Algorithms, third ed. MIT Press, 2001.
7. ​Acyclic directed graph generation is referred from the below mentioned link, http://condor.depaul.edu/rjohnson/source/graph\_ge.c
8. Serial implementation of the algorithm is referred from the following link http://web.stanford.edu/~liszt90/acm/notebook.html#file3