Parallelising Push-Relabel Maximum Flow Algorithm

Guneet Kaur

07919992

kaurg33@myumanitoba.ca

1 Abstract

Max-flow is a fundamental combinatorial problem. It involves deducing a flow through a flow network with the maximum possible flow rate. It is employed for applications like network resource allocation, scheduling problems, and image segmentation. This text aims to understand the max-flow algorithm and describe a parallel approach for it. In this paper, we study both parallel architectures: *shared memory and message passing interfaces.*. Code is written in C language with OpenMP and MPI programming models. One of the important issues while writing parallel programs is synchronization. Synchronization in MPI is ensured with proper sending and receiving of messages. In OpenMP, there is no such coordination of sending and receiving, but we need to take care of race conditions. Both these programming models offer some advantages and disadvantages which are also discussed in the paper. We put forward our parallel algorithm for push relabel and then evaluate it on random graphs. Towards the end, a theoretical analysis of the algorithm is performed and evaluation metrics are calculated. Later, we discuss the challenges faced along with the scope of future work and limitations.

2 Introduction & Motivation

Maximum flow problem is an important algorithm for many different areas like network resource allocation, scheduling problems, transportation problems, etc. A lot of research has been already carried out previously in coming up with different strategies to implement it parallelly which we will be discussing in the next section of related works. A greedy approach cannot solve the max-flow problem (possibly due to cycles). The primary issue would be the sequential nature of the problem. Hence it might be challenging to optimize even after parallelizing. Therefore, the end goal is to research a suitable parallelism technique for optimizing the running time of the **Goldberg-Tarjan push relabel algorithm**.

3 Related Work

This section offers a brief overview of the past works on parallelizing max-flow algorithms.

Goldberg's algorithm is presumed to be the fastest serial push-relabel algorithm. It considers both global update and gap heuristics. Additionally, there exist two different implementations of push-relabel: highest label based and first in first out (FIFO) order. Baumstark et al. [3] experiments suggests that Goldberg's highest label-based implementation is not much suitable for all the graph families. Due to this, they explore a push relabel in which active vertices (having excess flow associated with them) are in FIFO order. They observed that this implementation is better suitable for most graphs and at the same time better suitable for parallelization. Popic et al. [12] parallelized the push-relabel algorithm along with the global update heuristics. They achieved a speedup of 2 for rmf graphs when run on eight processors. [3] deals with coarse-grained parallelism to achieve better results by rebalancing the work. Baumstark et al. achieved speedups of up to 12 with 40 threads.

Anderson et al. [1] in their work implemented Goldberg's max-flow algorithm on shared-memory multiprocessors. They also focussed on executing global update heuristics concurrently with the main algorithm. In contrast to using a single processor, they speed up the algorithm up to 8.8 with 16 processors. Langguth et al. [11] in their study focussed on parallel push-relabel algorithms for maximum bipartite matching. They tested on several multi-core systems and compared their algorithm with state-of-the-art augmenting path-based algorithms. Bader et al. [2] in their study consider symmetric multiprocessors (SMPs) with large shared memories for implementation of the push-relabel network flow algorithm. Anderson et al. did work on shared-memory multiprocessors. Systems nowadays have deep memory hierarchies. Therefore, they focused on implementing the algorithm on modern shared-memory parallel computers. For sparse graphs, it was 2.1 to 4.3 times faster.

[6] works with CUDA GPU devices for better speedup of the algorithms. They adopt a switching technique between CPU and GPU for better efficiency. And hence, they achieve a speedup of 2 times.

[9] works for coloring-based parallelization techniques. They work in various graphs from DIMACs but focus extensively on sparse networks. The paper claims a speedup of 9 with 16 processors.

4 Push-Relabel Algorithm

In the push relabel algorithm, we consider a directed graph $G = (V, E)$. A vertex is associated with source s and sink t and the goal of the max-flow problem is to maximize the flow from source to sink. Every edge $(u, v) \epsilon E$ has a capacity c associated with it such that its capacity $c(u, v) > 0$.

Algorithm 1: The Push-Relabel Maximum Flow Algorithm

```
Data: Directed graph G = (V, E) with capacities C associated with edges
Result: Maximum flow for graph G
Initialize Preflow: Both flows and Heights;
while there exists an active vertex v do
    if no push operation possible then
        do relabel operation;
    else
        do push operation;
    end
end
return flow f
```

Traditionally, *Ford–Fulkerson* algorithm was used for maximum flow problems. This algorithm is based on finding augmenting paths from source to sink in the residual graph and adding the flow along with it. Similar to Ford–Fulkersonz, *Push-Relabel* also works on residual graph. But in contrast to examining the whole graph to find the augmenting path, push relabel works on one vertex at a time. An additional difference is of not maintaining a valid flow. Ford–Fulkerson ensures that at every step, the total incoming and outgoing flow for every vertex except source and sink is 0. On the other hand, Push-Relabel allows vertices to maintain an excess flow, but at the final steps, the difference should be 0 except for source and sink.

5 Sequential Version of Push-Relabel

In Goldberg and Tarjan Push-Relabel algorithm, excess flow is pushed from a higher height to a lower height. If there are no neighbors of lower height, we cannot push a flow. In this case, the vertex's height gets relabelled. Vertices that have an excess flow are called active vertices. Different algorithms exist for processing these active vertices. For instance, active vertices can either be processed in order of highest heights or FIFO queue (First In First Out).
Algorithm 1 shows the sequential code for the max-flow push relabel algorithm. This sequential version of push-relabel has a time complexity of $\mathcal{O}(V^2E)$. Firstly, the serial version of push-relabel is implemented. We note down the time taken T_s.

6 Parallel Version of Push-Relabel

There are various papers for implementing *Push-Relabel* parallelly. Different approaches have different degrees of parallelism and communication to computation ratio. The push-relabel algorithm is iterative. Each iteration takes one vertex from the queue that maintains the active vertices. We implement two different parallel versions of the push-relabel algorithm, one in *OpenMP* and *MPI*.

6.1 MPI Implementation of Push-Relabel

Being confident in MPI, our first approach is with MPI. Considering n vertices in a graph G and p processes. Here, we consider p odd. There will be one manager process $P0$ and the remaining $p-1$ will be worker processes. We will be following a manager-worker paradigm. Applying the PCAM model.

- **Partitioning:** Graph partitioning of n vertices among $p-1$ such that a processor P_i gets a set of vertices V_i such that it is the owner of that particular node in the graph. Here size of the graph n must be divisible by $p-1$. For example, if there is a graph of size $n=4$ and $p=3$ processes are considered. There will be 1 manager process and 2 worker processes. We get the chunk size $= n/p = 2$.

- **Communication:** Communication is ensured among processors using send and receive operations. They can either be synchronous or asynchronous.

- **Agglomeration:** Once the problem gets broken down into smaller tasks and necessary communication is figured out. Next, we ensure that our proposed parallel solution is practical and efficient. Evaluation of tasks and communication patterns concerning performance and cost of implementation is done in this stage.

- **Mapping**. Each processor gets a subset of vertices of size = chunk size calculated in the partitioning stage. Subtasks are assigned dynamically at run time to the processes. This ensures the maximum utilization of processes. In the push-relabel algorithm, a task is defined as performing push-relabel on an active vertex. Manager process

6.2 OpenMP Implementation of Push-Relabel

OpenMP gives us a big advantage over MPI to implement parallel algorithms. It provides a global view to memory that aids in making parallelization of code easier. It provides incremental parallelization. All the data is visible and shared among all the threads, race conditions can occur when there exists WAR (Write after Read) and RAW (Read After Write) dependencies. Therefore, care must be taken to ensure that these race conditions do not happen.

In order to implement the parallel program, we consider a FIFO queue, data structure queue is considered, and contention is required among workers for data structure. We mainly do two changes in our serial code for parallel program to work effectively. As we are using a global queue, care must be taken to ensure that while accessing the queue, only one of the worker is acessing the queue. This means that it should be in the form of atomic operations. In order to ensure atomicity, we use locks. Additionally, while pushing excess flow we ensure synchronization among workers.

Algorithm 2: Parallel Formulation of The Push-Relabel Maximum Flow Algorithm

```
Data: Directed graph G = (V, E) with capacities C associated with edges
Result: Maximum flow for graph G
Initially only master thread is running. It initializes height($u$), excess_flow($u$),
 flow($u, v$) and height($s$) = $|V|$ ;
for each $u \epsilon V - s$ do
    height(u) = 0;
end
for each $u \epsilon V$ do
    excess_flow(u) = 0;
end
for each (u, v) $\epsilon$ E do
    flow($u, v$) = 0;
end
while there exists an active vertex v schedule using a scheduling strategy among
 threads do
    if no push operation possible by the thread then
        do thread performs the relabel operation;
    else
        do push performed by the thread operation;
    end
end
return flow f
```

7 Performance Analysis

According to Grama et al. [5], the time taken by a parallel algorithm to solve a problem depends on various factors like problem size, number of processors, underlying machine characteristics, etc. The performance of a parallel code relies on a parallel system (combination of parallel algorithm and parallel architecture) rather than just relying only on the parallel algorithm. Our goal to get good performance is to maximize the computation/communication ratio while maintaining sufficient parallelism.

The theoretical time complexity of this algorithm is $\mathcal{O}(V^2E)$. This time complexity is calculated by counting the number of operations required to perform relabel operations or push operations instead of calculating execution time.

For evaluation purposes, we consider speedup and efficiency as our metrics. For speedups, we consider both speedup and relative speedup. **Relative speedup** is obtained from equation 2, where T_1 is the time taken for program execution by 1 processor and T_p is the time taken for program execution by p processors. Relative speedup is not an effective measure because some overheads may even come into the picture for 1 processor which is the same even when execution is performed on p processors. We denote T_s as the time is taken to execute the algorithm on a sequential computer. **Absolute Speedup** is calculated using the best sequential time and time taken by p processors. We define $process - parallel\ product$ as the cost of a parallel program and is defined as the total work done by all the processors.

$$C_p = n \ * \ T_p \tag{1}$$

$$Speedup\_absolute = T_s/T_p \tag{2}$$

$$Speedup\_relative = T_1/T_p \tag{3}$$

The **Efficiency** metric is defined as the ratio of speed up to that of the number of processors. In addition to efficiency, we also take into account **Isoefficiency metric** provided by Grama et al. [5]. It takes into account both the parallel program (size of the problem) and parallel architecture (number of processors) and aims to maintain the efficiency at a fixed value.

$$Efficiency = Speedup\_absolute/n \ = \ T_s/C_p \tag{4}$$

For our problem of performing push-relabel, we define our problem size as = the number of operations required to complete the program. Taking into account all the operations: relabel, saturating pushes, and non-saturating pushes, we known that atmost V^2E push or relabel operations can take place [4]. Let's assume, for simplicity that in our case, both push and relabel takes the same amount of time $= t_1$. Therefore, for sequential computer, $T_s = V^2E \ * \ t_1$.

For parallel program, using p processors(threads), we schedule the active vertices among processors(threads). For parallel implementation, T_p is calculated as:

$$T_p = T_{comm} + T_{comp} + T_{sync} + T_{wait} \tag{5}$$

In OpenMP, each thread assuming gets equal number of tasks from the active queue, computation time

T_{comp} = (total number of operations/number of processors)* $t_1 = (V^2E/p) * t_1$.

For communication time, unlike sending and receiving flows, heights of vertices in MPI which takes a lot of time, OpenMP gives us the advantage of saving communication time of sends and receives as entire data is already shared among processors. But, race conditions need to be taken care of using locks. Because locks themselves are memory locations and access to these memory locations counts as communication time T_{comm}. Locks can adversely degrade the parallel performance, therefore needs to be carefully implemented. Table 1 shows the time taken by 1 processor in OpenMP using with and without locks for the graph of size 200. We use locks in our OpenMP version at two places:

1. Enqueue (Addition) and Dequeue (Deletion) of a vertex from the global queue are protected using a lock. This can be done using 1 lock.

2. In push operations, we lock nodes before pushing the flow. Each push operation needed two locks.

Using Locks	0.137651
No Locks	0.031394

Table 1: Runtime comparison with and without locks (1 Processor) V = 200

Let t_c be the average contention time due to the locking operation. Since there can be atmost V^2E/p operations per processor and atmost two locks in each operation. A lock is acquired by 1 processor, causing a wait for other processors. This gives us the communication time

$T_{comm} = p * t_c$.

Thus, the total parallel execution time

$T_p = (V^2E/p) * t_1 + p * t_c$.

Though relative speedup is not an effective measure, we consider that metric here for convenience. The speedup S and efficiency E for this algorithm is given by

$$S = \frac{T_1}{T_p} = \frac{V^2E * t_1}{\frac{V^2E}{p} * t_1 + p * t_c} \tag{6}$$

$$E = \frac{S}{p} = \frac{V^2E * t_1}{V^2E * t_1 + p^2 * t_c} \tag{7}$$

7.1 Evaluation: Times and Speedups

For evaluation purposes, we generate the random graphs in C language, giving number of vertices and edges as input to the script. We perform our evaluations for $N = 50, 100, 200, 300, 400, 500$ & 600 vertices in the graph Figure 1, 2 shows for graph with 200 and 400 vertices. It can be clearly seen from these graphs that we donot get the linear speedup with increase in the number of processors, instead speedups starts to saturate. This result is in consistent with the Amdahl's law which states that the efficieny drops with increase in number of processors.

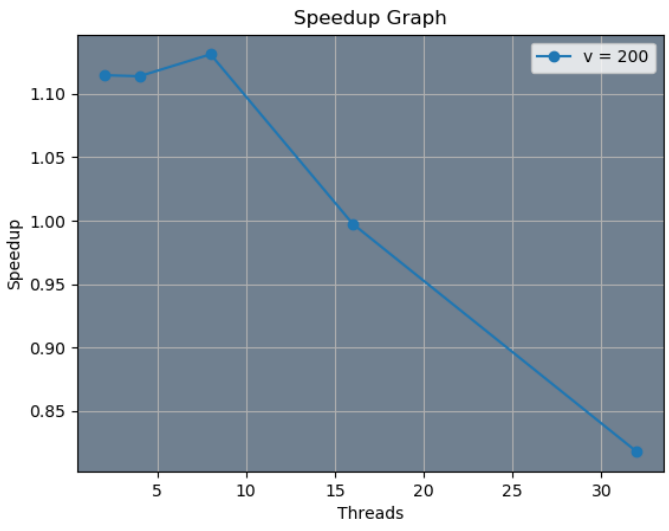

Figure 1: speedup for v = 200

Figure 3, 4 shows that increase in the problem size for fixed number of processors yields a good efficiency (speedup). This is because computation per thread increases and computation to communication ratio is maximized.

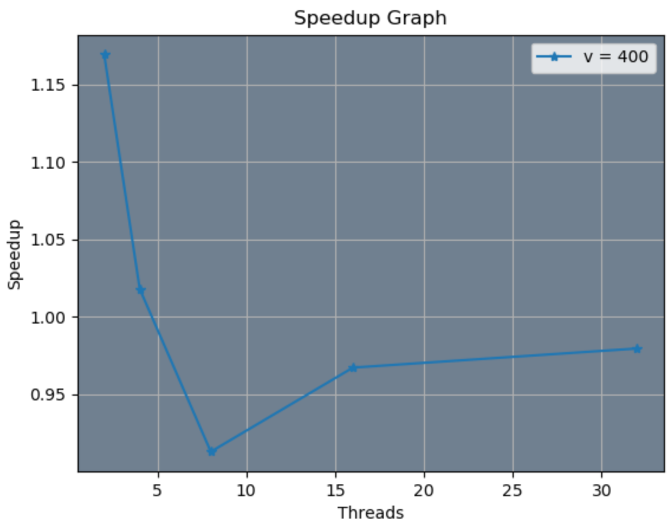

Figure 2: speedup for v = 400

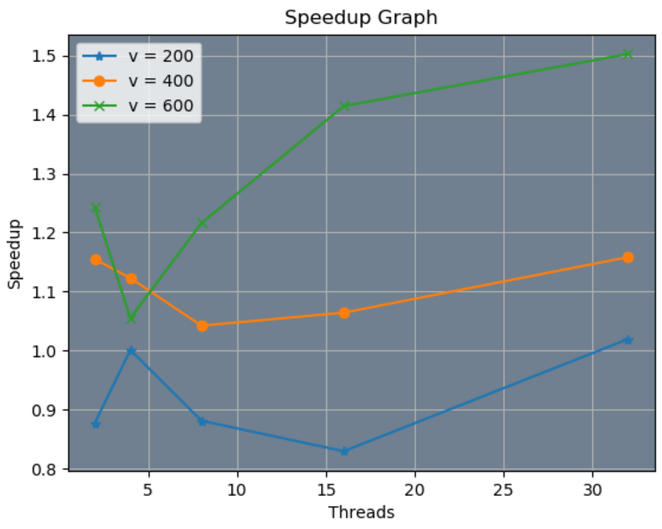

Figure 3: speedup comparison

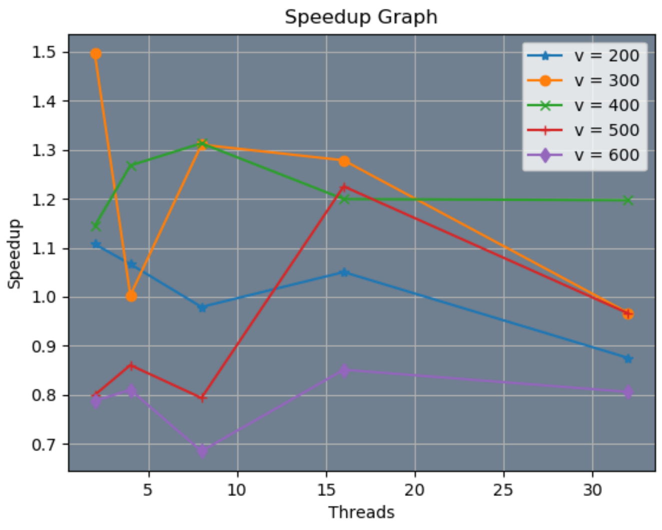

Figure 4: speedup comparison

threads / vertices	**200**	**300**	**500**
1	0.247443	0.542015	0.676664
2	0.223445	0.362301	0.681802
4	0.231987	0.413684	0.628826
8	0.25268	0.540322	0.44175
16	0.265564	0.560416	0.560171
32	0.28268	0.58622	0.681203

8 Challenges

The biggest challenge that we encountered was forming a parallelization strategy in MPI. Due to so much communication using sends and receives within processors, we found it hard to come up with a good parallelization strategy that can maximize the computation to communication ratio. Furthermore, we initially planned on a hybrid implementation of push-relabel. But could not complete due to challenges in MP. Additionally, to care for atomicity, we had to use locks (mutex-based synchronizations). Coming with the correct parallel algorithm for asynchronous lock-free implementation as in [7] posed another challenge.

9 Conclusion

In this paper, a parallel implementation of Goldberg's algorithm on a shared memory multiprocessor is presented. In our approach, there existed contention for the global queue, which slowed down our algorithm. We generated random graphs in the C language of different sizes and used them for evaluating our algorithm. In addition, we put forward theoretical analysis for speedup and efficiency. Furthermore, our results were consistent with results presented by Grama et al. [5] that for a fixed problem size, with an increase in the number of processors, speedup starts to saturate (Amdahl's Law). We try to make our parallel system scalable by keeping efficiency constant by increasing the size of the problem and the number of processors simultaneously.

10 Future Work & Limitations

At last, we discuss the possible future works and limitations of our current approach.We could not ensure the correctness of the implementation of parallel push-relabel in MPI. In message-passing interfaces, since the entire graph was divided among the processors, a change in height and excess flows of one node by a processor needs to be updated for processors holding the nodes as per the location of nodes. It was observed that with the current algorithm, huge overheads were there due to communication times which ultimately resulted in negative speedups. OpenMP on other hand had a limitation of using locks which are expensive operations and degrade the parallel performance. In the future, we plan on doing a more theoretical analysis of what could have caused this slowdown in addition to locks.

For evaluation purposes, initially, the plan was to work on net-flow benchmarks provided by the DIMACS challenge [8], partition them using PARMETIS [10] and then do the performance analysis. We were unable to do so considering the time constraint and scope of the project. In forseable future , we plan on working on the hybrid implementation of the push-relabel algorithm using both OpenMP and MPI programming models. This will allow us to take advantage of both OpenMP and MPI by reducing the communication time and gaining an advantage from shared memory systems. Additionally, we plan on exploring and using the proposed DIMACS evaluation benchmarks along with the ParMetis library. Furthermore, we can use the concept of transactional memory [13]. Currently, synchronization is ensured using locks, but as studied above and can be seen from the results, lock-based synchronization can lead to deadlocks if not written properly. They make fine-grained synchronization difficult. Using transactional memory, instead of performing push/relabel operations using locks, we simply place them in a transaction and hardware takes care of it. A software transactional memory (STM): part of McRT, an experimental Multi-Core RunTime is described in [14]

References

[1] Richard J Anderson and Joao C Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 168–177, 1992.

[2] David A Bader and Vipin Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. Technical report, Georgia Institute of Technology, 2006.

[3] Niklas Baumstark, Guy Blelloch, and Julian Shun. Efficient implementation of a synchronous parallel push-relabel algorithm. In *Algorithms-ESA 2015*, pages 106–117. Springer, 2015.

[4] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[5] Ananth Y Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(3):12–21, 1993.

[6] Zhengyu He and Bo Hong. Dynamically tuned push-relabel algorithm for the maximum flow problem on cpu-gpu-hybrid platforms. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–10. IEEE, 2010.

[7] 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*, 22(6):1025–1033, 2010.

[8] David S Johnson, Catherine C McGeoch, et al. *Network flows and matching: first DIMACS implementation challenge*, volume 12. American Mathematical Soc., 1993.

[9] Gökçehan Kara and Can Özturan. Algorithm 1002: Graph coloring based parallel push-relabel algorithm for the maximum flow problem. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–28, 2019.

[10] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis. *Parallel graph partitioning and sparse matrix ordering library. Version*, 2, 2003.

[11] Johannes Langguth, Ariful Azad, Mahantesh Halappanavar, and Fredrik Manne. On parallel push–relabel based algorithms for bipartite maximum matching. *Parallel Computing*, 40(7):289–308, 2014.

[12] Victoria Popic and Javier Vélez. Parallizing the push-relabel max flow algorithm. 2010.

[13] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, 2006.

[14] Alexander Shekhovtsov and Václav Hlaváč. A distributed mincut/maxflow algorithm combining path augmentation and push-relabel. *International journal of computer vision*, 104(3):315–342, 2013.