A Lock-free Multi-threaded Algorithm for the Maximum Flow Problem

Bo Hong
Drexel University, Philadelphia, PA 19104
bohong@coe.drexel.edu

Abstract

The maximum flow problem is an important graph problem with a wide range of applications. In this paper, we present a lock-free multi-threaded algorithm for this problem. The algorithm is based on the push-relabel algorithm proposed by Goldberg. By using re-designed push and relabel operations, we derive our algorithm that finds the maximum flow with $O(|V|^2|E|)$ operations. We demonstrate that as long as a multi-processor architecture supports atomic 'read-update-write' operations, it will be able to execute the multi-threaded algorithm free of any lock usages. The proposed algorithm is expected to significantly improve the efficiency of solving maximum flow problem on parallel/multi-core architectures.

1 Introduction

Given the increasing emphasis on multi-core architectures, the extent to which an application can be multi-threaded to keep the multiple processor cores busy is likely to be one of the greatest constraints on the performance of next generation computing platforms. However, except for the embarrassingly parallel workload where no particular effort is needed to segment the problem into a very large number of independent tasks, multi-threading is often very challenging to achieve efficiency due to the intrinsic data or control dependencies in the applications.

In this paper, we study the maximum network flow problem in the settings of multi-processor platforms. A flow network is a graph $G(V, E)$ where edge $(u, v) \in E$ has capacity c_{uv}. G has source $s \in V$ and sink $t \in V$. A flow in G is a real valued function f defined over $V \times V$ that satisfies the following constraints:

1. $f(u, v) \leq c_{uv}$ for $u, v \in V$
2. $f(v, u) = -f(u, v)$ for $u, v \in V$
3. $\sum_{v \in V} f(v, u) = 0$ for $u \in V - \{s, t\}$

The value of a flow f is defined as $|f| = \sum_{u \in V} f(s, u)$. The maximum network flow problem searches for a flow with the maximum value. The maximum network flow problem is an important graph problem with a wide range of applications. For example, certain placement and routing problems in VLSI design are formulated as maximum flow problems.

In this paper, we present a *lock-free multi-threaded* algorithm for the maximum network flow problem. We demonstrate that as long as a multi-processor architecture supports atomic 'read-update-write' operations, it will be able to execute our algorithm using an arbitrary number of threads (up to the number of network vertices) and the execution is free of any lock usages. To the best of our knowledge, this is a first *lock-free* parallel algorithm for the maximum network flow problem. This algorithm has important practical significance: the performance bottleneck is no longer due to the lack of parallelism in the algorithm. As long as the computer architecture has enough bandwidth to support the algorithm's concurrent accesses to the shared memory, linear speed-up can be expected as the number of processors increases.

The proposed algorithm is based on the push and relabel algorithm by Goldberg [9]. The lock-free property is enabled by the re-designed push and relabel operations. We prove that the proposed algorithm finds maximum flow with $O(|V|^2|E|)$ operations.

The rest of the paper is organized as follows. In section 2, we briefly review algorithms and parallel implementations for maximum network flow problems. Section 3 presents the model of the target multi-processor platform. The algorithm is presented in Section 4, where we prove its optimality in Section 5 and its complexity bound in Section 6. Discussions are provided in Section 7.

2 Related Work

Early solutions to the maximum network flow problem are based on the augmenting path method due to Ford and Fulkerson [6]. Edmonds and Karp [5] demonstrated that pushing flow along the shortest augmenting path has a polynomial running time of $O(|V||E|^2)$. Dinitz [4] suggested searching for augmenting paths in phases and handling all

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

augmenting paths of a given shortest length in one phase, which yields an execution time of $O(|V|^2|E|)$. The concept of preflow was introduced by Karzanov in [13], which leads to a $O(|V|^3)$ algorithm. The execution time of has been further improved by using various techniques such as capacity scaling [7] and dynamic trees [10].

Goldberg etc. designed the push-relabel method [9] that maintains a preflow and a distance labeling, and uses push and relabel operations to update the preflow until a maximum flow is found. The raw algorithm is of $O(|V|^2|E|)$ complexity. By executing the push and relabel operations in a FIFO order, an $O(|V|^3)$ algorithm is achieved in [9]. The running time of the push-relabel method is improved to $O(|V||E|\log(|V|^2|E|))$ in [9] by using dynamic trees. An excellent survey of recent development in maximum network flow problem is presented in [8].

Parallel and distributed algorithms for the maximum flow problem have also received a lot of attention. A first parallel algorithm, due to Shiloach and Vishkin [15], runs in $O(|V|^2\log|V|)$ time using a $|V|-$processor PRAM. Goldberg pointed out that the dynamic tree algorithm in [9] can be implement on an EREW PRAM, taking $O(|V|^2\log|V|)$ time and $O(|V|)$ processors. Parallel algorithms for restricted cases of maximum network flow problems have also been developed. For example, for planar directed graphs, Johnson designed an $O(\log^3|V|)$ algorithm using PRAM with $O(|V|^4)$ processors [12]. PRAM model [11], however, cannot be considered as a physically realizable model because as the number of processors and the size of the global memory scale up, it quickly becomes impossible to ignore the impact of the interconnection.

Practical implementations of parallel algorithms have also been studied. Anderson and Setubal [1] augmented the push-relabel algorithm with a *global relabeling* operation, which, applied periodically, updates the distance labels to be the exact distance to the sink. Experimental results demonstrate good speed-ups on parallel computers. Bader etc. [2] designed a parallel algorithm using gap relabeling heuristic with considerations of the cache performance, also demonstrating good performance. These parallel implementations share the common feature of using locks to protect every push and relabel operation *in its entirety*, which limits the parallelism of implementation and will lead to contention and performance degradation when the number of processors scales up.

Our algorithm differs from the PRAM based study in that our algorithm is readily implementable on modern computer architectures (Implementing a PRAM algorithm requiring $O(|V|)$ processors would be very challenging when the input graph has a large number of vertices, let alone the cost of the interconnect which PRAM ignores). Compared with existing parallel algorithms that are practically implementable, our novelty is in the removal of lock usages, thus greatly exposing parallelism. (For example, existing methods need to lock the two vertices of an edge during a push, which prohibits any other operations to be applied to the vertices, even if they are applicable.)

3 The Target Multi-Processor Platform

The target multi-processor platform consists of multiple processor that access a shared memory. We assume that the architecture supports sequential consistency and atomic 'read-modify-write' instructions, as most modern parallel architectures do.

A system provides sequential consistency if every node (processor cores in a multi-core architecture) of the system sees the memory accesses in the same order, although the order may be different from the order as defined by real time (as observed by hypothetical external observer or global clock) of issuing the operations [14].

Atomic 'read-modify-write' instructions allows the architecture to sequentialize such instructions automatically. For example, suppose $x \leftarrow x + d_1$ and $x \leftarrow x + d_2$ are executed by two processors simultaneously, the architecture will atomically complete one instruction after another, thus the final value of x will be the accumulation of d_1 and d_2.

4 The Lock-free Multi-threaded Algorithm

Before presenting the algorithm and its programming implementation, we first briefly re-state some notations for network flow problems.

Given a direct graph $G(V, E)$, function f is called a flow if it satisfies the three constraints above. Given $G(V, E)$ and flow f, the *residual capacity* $c_f(u, v)$ is given by $c_{uv} - f(u, v)$, and the *residual network* of G induced by f is $G_f(V, E_f)$, where $E_f = \{(u, v)|u \in V, v \in V, c_f(u, v) > 0\}$. Thus $(u, v) \in E_f \Leftrightarrow c_f(u, v) > 0$.

For each node $u \in G$, $e(u)$ is defined as $e(u) = \sum_{w \in V} f(w, u)$, which is the net flow into node u. Constraint 3 in the problem statement requires $e(u) = 0$ for $u \in V - \{s, t\}$. But the intermediate result before an algorithm terminates may have non-zero $e(u)$'s. We say vertex $u \in V - \{s, t\}$ is *overflowing* if $e(u) > 0$. An integer valued height function $h(u)$ is also defined for every node $u \in V$. We say u is higher than v if $h(u) > h(v)$.

The algorithm is listed below:

1. **Initialize** $h(u)$, $e(u)$, and $f(u, v)$
2. While there exist applicable **push** or **lift** operations execute the applicable operations *asynchronously*

where the operations of **initialize**, **push**, and **lift** are defined as follows:

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

$h(s) \leftarrow |V|$
for each $u \in V - \{s\}$
    $h(u) \leftarrow 0$
for each $(u, v) \in E$
    $f(u, v) \leftarrow 0$
    $f(v, u) \leftarrow 0$
for each $(s, u) \in E$
    $f(s, u) \leftarrow c_{su}$
    $f(u, s) \leftarrow -f(s, u)$
    $e(u) \leftarrow c_{su}$

- $Push(u, \hat{v})$: applies if u is overflowing, and $\exists v \in V$ s.t. $(u, v) \in E_f$ and $h(u) > h(v)$,

$\hat{v} \leftarrow \text{argmin}_v[h(v) \mid c_f(u, v) > 0 \text{ and } h(u) > h(v)]$
$d \leftarrow \min(e(u), c_f(u, \hat{v}))$
$f(u, \hat{v}) \leftarrow f(u, \hat{v}) + d$
$f(\hat{v}, u) \leftarrow f(\hat{v}, u) - d$
$e(u) \leftarrow e(u) - d$
$e(\hat{v}) \leftarrow e(\hat{v}) + d$

- $Lift(u)$: applies if u is overflowing, and $h(u) \leq h(v)$ for all $(u, v) \in E_f$,

$$h(u) \quad \leftarrow \min\{h(v) | c_f(u, v) > 0\} + 1$$

The push operation in this algorithm pushes to the lowest neighbor, which is the major modification to the original push relabel algorithm in [9] (which pushes to a neighbor whose height is lower by 1). This new choice of destination for pushes is essential for the correctness of the lock-free algorithm, as will be shown in the next two sections.

The algorithm can be easily multi-threaded by assigning each thread T_i a distinct subset of of the vertices V_i (*s.t.* $V_i \cap V_j = \emptyset$ if $i \neq j$, and $\cup_i\{V_i\} = V$). The initialization step is performed by the main thread before spawning all the multiple threads. After the initialization step, each thread T_i checks whether any push or lift operations can be applied to any of the vertices in V_i, and executes the applicable operations if there exist any.

When implementing the algorithm on a real computer, it is reasonable to have the same number of threads as the processor cores. Thread assignment can be either static or dynamic. Additionally, it is desirable to have balanced load across the threads, letting each thread execute (close-to) the same number of operations. Load balance is determined by the assignment of vertices to the threads (and of course also by the topology of the input graph). As the concentration (and the novelty) of this paper is the lock-free property of the presented algorithm, we leave the optimal vertex assignment problem for future studies as itself is another open research problem.

Variables	Written by thread(s)	Read by thread(s)
$h(u)$	u	u, and w s.t. $(w, u) \in E_f$
$e(u)$	u, or w s.t. $(w, u) \in E_f$	u
$c_f(u, v)$	u or v	u and v
$e', \hat{v}, \hat{h}, h', d$	per thread private variables	

Table 1. Variable access characteristics

Without loss of generality, we assume that for each vertex $u \in V$ there is one thread responsible for executing $push(u, \hat{v})$ and $lift(u)$. In the following analysis, we will use u to denote both vertex u and the thread responsible for vertex u, which can be easily clarified given the context.

The algorithm leads to the following lock-free programming implementation where e', $\hat{v}$, $\hat{h}$, and h' are per-thread private variables and $h(u)$, $e(u)$, and $c_f(u, v)$ ($u \in V$, $(u, v) \in E_f$) are shared among all threads. The sharing characteristics of the variables are listed in Table 1. For programming convenience, the implementation maintains $c_f(u, v)$ rather than $f(u, v)$. The constraint $f(u, v) \leq c_{uv}$ in the problem statement translates to $c_f(u, v) \geq 0$. Upon termination of the algorithm, the flow $f(u, v)$ along each edge $(u, v) \in E$ can be derived easily from $c_f(u, v)$ since $c_f(u, v) = c_{uv} - f(u, v)$.

Initially, only the master thread is running.

1. The master thread initializes $h(u)$, $e(u)$, and $c_f(u, v)$
    $h(s) \leftarrow |V|$
    for each $u \in V - \{s\}$
        $h(u) \leftarrow 0$
    for each $(u, v) \in E$
        $c_f(u, v) \leftarrow c_{uv}$
        $c_f(v, u) \leftarrow c_{vu}$
    for each $(s, u) \in E$
        $c_f(s, u) \leftarrow 0$
        $c_f(u, s) \leftarrow c_{us} + c_{su}$
        $e(u) \leftarrow c_{su}$
2. The master thread creates one thread for each vertex $u \in V - \{s, t\}$, and then terminates itself.
3. Each of the newly created thread u executes lines 4-22:
4. while $e(u) > 0$
5. do
6. $e' = e(u)$
7. $\hat{v} \leftarrow null$
8. $\hat{h} \leftarrow \infty$
9.   for each $(u, v) \in E_f$ /* i.e. $c_f(u, v) > 0$ */
10. $h' \leftarrow h(v)$
11.   if $h' < h(\hat{v})$, then
12. $\hat{v} \leftarrow v$
13. $\hat{h} \leftarrow h'$
14.   end for   /* $\hat{v}$ is u's lowest neighbor in E_f */

15. if $h(u) > \hat{h}$, then /* $push(u, \hat{v})$ is applicable */
16. $\quad d \leftarrow \min(e', c_f(u, \hat{v}))$
17. $\quad c_f(u, \hat{v}) \leftarrow c_f(u, \hat{v}) - d$
18. $\quad c_f(\hat{v}, u) \leftarrow c_f(\hat{v}, u) + d$
19. $\quad e(u) \leftarrow e(u) - d$
20. $\quad e(\hat{v}) \leftarrow e(\hat{v}) + d$
21. else /* $lift(u)$ is applicable */
22. $\quad h(u) \leftarrow \hat{h} + 1$

The sequential consistency property of the architecture guarantees that each thread executes its own lines 4-22 in the order specified above. Updates to shared variables $c_f(u, \hat{v})$, $c_f(\hat{v}, u)$, $e(u)$, and $e(\hat{v})$ (lines 17-20), due to the support of atomic 'read-modify-write' instructions, are executed atomically by the architecture. Other than the two execution characteristics provided by the architecture, we do not impose any order in which executions from multiple threads can or should be interleaved, as it will be left for the sequential consistency property of the architecture to decide.

Shared variable updates in $push(u, v)$ are all in the form of $x \leftarrow x + \delta$ so they can be executed correctly by the architecture without any lock protection. Note that $h(u)$ is updated by and only by thread u during a $lift(u)$ operation. Thus even though $h(u)$ is shared (multiple threads may read its value), $h(u)$ does not need lock protection because only the single thread u needs to update it. When another thread reads $h(u)$ while it is being updated by thread u, the reader thread will get the value of $h(u)$ either before or after the update. Our algorithm does not require a strict order as to what value must be obtained by the reader thread.

Now we have shown that the algorithm indeed can be implemented without using any locks. Next we will prove that despite the seemingly uncontrolled and unpredictable execution order, the algorithm still solves the maximum flow problem. In fact, letting the threads advance without lock-based synchronization is the essence of our lock-free multi-threaded algorithm. We shall first prove that the algorithm finds the maximum flow, if it terminates. We shall next prove that the algorithm indeed terminates.

5 Correctness Proof

For notational convenience, we use both $f(u, v)$ and $c_f(u, v)$ for discussions in this section, although the actual programming implementation only uses $c_f(u, v)$.

We start with the following observations on the algorithm.

Lemma 1. *During the execution of the algorithm, for any $u \in V$, $h(u)$ never decreases during the execution of the algorithm.*

Lemma 2. *During the execution of the algorithm, for any overflowing vertex, either a lift or a push operation can be applied.*

The proof of Lemmas 1 and 2 is the same as that in [9].

Our next observation is that even though the execution at multiple threads can be interleaved arbitrarily, it actually reduces to only two equivalent scenarios. This reduction allows us to continue the proof by referring to the operations, instead of to its programming implementation line by line.

We define the '*consequence*' of a $push(u, \hat{v})$ to be the values of $e(u)$, $e(\hat{v})$, $c_f(u, \hat{v})$, and $c_f(\hat{v}, u)$ after the push, the '*consequence*' of a $lift(u)$ to be the value of $h(u)$ after the lift. We also define the '*trace*' of the interleaved execution of multiple threads to be the order in which instructions from the threads are executed in real time. We say two traces are *equivalent* if they have the same consequences.

The trace of a single push operation can be split into two stages: lines 6-16 and lines 17-20. Lines 6-16 test whether a push is applicable, and if applicable, how much flow needs to be pushed to which neighbor. We call this the '*preparation*' stage of the push. Lines 17-20 updates the shared variables accordingly, which we call the *fulfillment* stage of the push. Similarly, the trace of a single lift operation can also be split into two stages: lines 6-15, and line 22. Lines 6-15 test whether a lift is applicable, and if applicable, what should be the new height of the vertex. This is the '*preparation*' stage of the lift. Line 22 updates the vertex height, which is defined as the '*fulfillment*' stage of the lift.

Now we present the following pre-defined traces, each involving two push and/or lift operations:

1. the *stage-clean trace* where multiple operations do not have any overlapping in their executions. In the example of two operations, it can be illustrated as follows: $P1 \rightarrow F1 \rightarrow P2 \rightarrow F2$. The $P1$ notation denotes the preparation stage of operation 1. $F1$ denotes the fulfillment stage of operation 1. $P2$ and $F2$ are defined similarly for operation 2. The $\rightarrow$ notation represents precedence in real time order.

2. the *stage-stepping trace* where all the operations execute their preparation stages before any one proceeds with its fulfillment stage. In the example of three operations, we may have the following stage-stepping traces: $P1 \rightarrow P2 \rightarrow P3 \rightarrow F1 \rightarrow F2 \rightarrow F3$ or $P1 \rightarrow P2 \rightarrow P3 \rightarrow F1 \rightarrow F3 \rightarrow F2$ (and four more possibilities depending on which operation finishes its fulfillment stage earlier).

With the above notational preparation, we have the following lemma:

Lemma 3. *Any trace of two push and/or lift operations is equivalent to either a stage-clean trace or a stage-stepping trace.*

The proof of Lemma 3 is straightforward. We simply need to enumerate all the possible pairs of operations that might be interleaved and derive an equivalent trace (either stage-clean or stage-stepping) for each such pair. The detailed proof is omitted here.

It is easy to show that traces with more operations can also be reduced similarly as stated in the next lemma.

Lemma 4. *For any trace of three or more push and/or lift operations, there exists an equivalent trace consisting of a sequence of non-overlapping traces, each of which is either stage-clean or stage-stepping.*

The proof is similar to that for Lemma 3. We need to examine various scenarios that the operations might be interleaved. The detailed proof is omitted here.

With Lemmas 3 and Lemma 4, we can greatly simplify our discussion by confining to stage-clean and stage-stepping traces rather than arbitrarily interleaved operations. We have the next important property of the algorithm.

Lemma 5. *If the algorithm terminates, then* $h(u) \leq h(v)+1$ *for any edge* $(u, v) \in E_f$.

Proof: We show that throughout the execution of the algorithm, $(u, v) \in E_f$ implies $h(u) \leq h(v) + 1$ except for one occasion where $h(u) > h(v) + 1$ may occur. However, we show this specific occasion is *transient* in that (u, v) will be removed from E_f by a $push(u, v)$ operation, thus removing the requirement on $h(u)$ and $h(v)$. Therefore, if the algorithm terminates (i.e. when no push or lift can be applied), we must have $h(u) \leq h(v) + 1$ for any $(u, v) \in E_f$.

The proof is by induction on the push and lift operations, with consideration in the interleaved execution of the operations.

Initially, all the nodes have height of 0 except s. The only edges (u, v) that satisfy $h(u) > h(v) + 1$ are those for which $u = s$, and those edges are saturated in the initialization step so they are not in the residual network E_f. So we have $h(u) \leq h(v) + 1$ for $(u, v) \in E_f$ right after the initialization.

Now consider the execution of push and lift operations. We have the following scenarios:

1. A $lift(a)$ operation is executed in its entirety without being interleaved with any other operations. For the residual edge (a, b) that leaves a, the lift operation guarantees $h(a) \leq h(b) + 1$ afterward. For the residual edge (c, a) that enters a, $h(c) \leq h(a) + 1$ before the lift implies $h(c) \leq h(a) + 1$ afterward since $h(a)$ never decreases according to Lemma 1.

2. A $push(a, b)$ operation is executed in its entirety without being interleaved with any other operations. This operation may add (b, a) to E_f or may remove (a, b) from E_f. In the former case, we have $h(a) > h(b)$ (otherwise $push(a, b)$ cannot be applied). Thus we have $h(b) \leq h(a) + 1$ for the new residual edge (b, a). In the latter case, the removal of (a, b) from E_f removes the requirement that $h(a) \leq h(b) + 1$.

3. The executions of $lift(a)$ and $lift(b)$ are interleaved. As indicated in Lemma 3, a trace of two lift operations is equivalent to either a stage-clean or a stage-stepping trace. A stage-clean trace reduces to scenario 1 discussed above. For a stage-stepping trace, we may have the following four sub-scenarios:

   (a) Initially, $(a, b) \in E_f$ and $(b, a) \in E_f$. In this case, we must have $h(a) = h(b)$ because otherwise we either have $h(a) > h(b)$ or $h(b) > h(a)$, then either $push(a, b)$ or $push(b, a)$ can be applied, which contradicts the assumption of the scenario. For $lift(a)$ to be applicable, we must have $h(c) \geq h(a)$ for all $(a, c) \in E_f$, then $h(a) = h(b)$ implies $h(b) = \min\{h(c)|(a, c) \in E_f\}$ because $(a, b) \in E_f$. So $\min\{h(c)|(a, c) \in E_f\} + 1 = h(b) + 1 = h(a) + 1$ and consequently $lift(a)$ will update $h(a) \leftarrow h(a) + 1$. Similarly, $lift(b)$ will update $h(b) \leftarrow h(b) + 1$. So after the two lift operations, we still have $h(a) = h(b)$. Thus $h(a) \leq h(b) + 1$ is maintained for residual edge (a, b) and $h(b) \leq h(a) + 1$ is maintained for residual edge (b, a).

   (b) Initially, $(a, b) \in E_f$ but $(b, a) \notin E_f$. In this case, an applicable $lift(a)$ implies $h(a) \leq h(b)$ before the lift because otherwise we need to apply $push(a, b)$ instead. $lift(a)$ updates $h(a) \leftarrow \min\{h(c)|(a, c) \in E_f\} + 1$. Since $(a, b) \in E_f$, $h(b)$ will be polled to compute the $\min$, so the lifted $h(a)$ will be lower than $h(b) + 1$. As $h(b)$ is further increased by $lift(b)$, we must have $h(a) \leq h(b) + 1$ after the two lift operations.

   (c) Initially, $(b, a) \in E_f$ but $(a, b) \notin E_f$. This is symmetric to sub-scenario (c). Similarly, we will have $h(b) \leq h(a) + 1$ after the two lift operations.

   (d) Initially, $(a, b) \notin E_f$ and $(b, a) \notin E_f$. Due to the lack of residual edges between a and b, this is a trivial sub-scenario because the update of $h(a)$ and $h(b)$ are not constrained by each other.

4. The execution of $push(a, b)$ is interleaved with $push(b, c)$. It can be shown easily (as a special case of Lemma 3) that this particular trace is equivalent to a stage-clean trace where $push(a, b)$ is executed in its entirety before (or after) $push(b, c)$ is executed in its entirety. Then this scenario reduces to scenario 2 and the same analysis applies. We have $h(u) \leq h(b) + 1$ for $(a, b) \in E_f$ before and after the two operations.

5. The executions of $push(a,b)$ and $lift(b)$ are interleaved. According to Lemma 3, this trace is equivalent to either a step-clean or a stage-stepping trace. If it is stage-clean, then this reduces to scenarios 1 and 2 discussed above.

   If this is equivalent to a stage-stepping trace, we have the following two sub-scenarios to consider. Note we must have $(a,b) \in E_f$ for $push(a,b)$ to be applicable.

   (a) $(b,a) \in E_f$ before the fulfillment stage of $push(a,b)$. In this sub-scenario, $push(a,b)$ may remove (a,b) from E_f and hence remove the requirement that $h(a) \leq h(b)+1$. If $push(a,b)$ does not remove (a,b) from E_f, then $h(a) \leq h(b)+1$ before the push (induction assumption) implies $h(a) \leq h(b)+1$ thereafter. The operation $lift(b)$ increases $h(b)$ to $\min\{h(w)|(b,w) \in E_f + 1\}$, which implies $h(b) \leq h(a)+1$ after the lift since $(b,a) \in E_f$.

   (b) $(b,a) \notin E_f$ before the fulfillment stage of $push(a,b)$. $push(a,b)$ will add (b,a) into E_f. This is the specific scenario where $h(a)$ may become larger than $h(b)+1$ for residual edge (a,b), as mentioned in the beginning of the proof. We have the following two cases to consider:

      i. $(b,a) \in E$. In this case, we must also have $f(b,a) = c_{ba}$ before the push. Otherwise $f(b,a) \leq c_{ba}$ then we can still push some flow from b to a, which means $(b,a) \in E_f$ - but this contradicts the assumption that $(b,a) \notin E_f$. Let d denote the amount of flow $push(a,b)$ sends from a to b.
      $push(a,b)$ may remove (a,b) from E_f. The removal of (a,b) from E_f removes the requirement that $h(a) \leq h(b)+1$.
      (b,a) will be added into E_f by the fulfillment stage of $push(a,b)$. Note that $lift(b)$ calculates the new height of $h(b)$ during its preparation stage, during which $(b,a) \notin E_f$. So $h(a)$ will not be polled by the preparation stage of $lift(b)$ (i.e. $h(a)$ will not be included when computing $\min\{h(w)|(b,w) \in E_f\}+1$ for $lift(b)$). Consequently, we may have $h(b) > h(a)+1$ after $lift(b)$ updates $h(b)$. In the mean time, we have $(b,a) \in E_f$ by the end of this trace. The combination of $h(b) > h(a)+1$ and $(b,a) \in E_f$ violates the requirement that $h(u) \leq h(v)+1$ for $(u,v) \in E_f$.
      This violation is only transient. We have $e(b) > 0$, $(b,a) \in E_f$, and $h(b) > h(a)+1$ after the trace. $h(b) > h(a)+1$ implies a was lower than all of b's neighbors in E_f before the trace (otherwise $h(b)$ would be increased to lower than $h(a)+1$). a being b's lowest neighbor means $push(b,a)$ is now applicable. Next we will examine how much flow $push(b,a)$ will send.
      Let d' denote the amount of flow that $push(b,a)$ will send from b to a. According to the algorithm, $d' = \min\{c_f(b,a), e(b)\}$. $f(b,a) = c_{ba}$ before $push(a,b)$ implies $c_f(b,a) = d$ thereafter. In the mean time, $e(b)$ will be increased by d since $push(a,b)$ just sent d amount of flow to vertex b. Note that $e(b) > 0$ before $push(a,b)$ (otherwise $lift(b)$ will not be applicable), so we have $e(b) > d$ and consequently $d' = \min\{c_f(b,a), e(b)\} = \min\{d, e(b)\} = d$.
      $d' = d$ means we will have $f(b,a) = c_{ba}$ upon completion of $push(b,a)$, which removes (b,a) from E_f and hence removes the requirement that $h(b) \leq h(a)+1$. In summary, if the algorithm terminates, then $push(b,a)$ must have already completed, we will have $h(u) \leq h(v)+1$ for $(u,v) \in E_f$.

      ii. $(b,a) \notin E$. We must have $f(a,b) = 0$ because otherwise $f(a,b) > 0$ leads to $c_f(b,a) = c_{vu} - f(b,a) = 0 + f(b,a) > 0$, which means $(b,a) \in E_f$ and contradicts the assumption that $(b,a) \notin E_f$.
      Similar to the previous $(b,a) \in E$ case, we may have $h(b) > h(a)+1$ when the trace finishes. Because $push(a,b)$ will add (b,a) into E_f, we will violate the requirement that $h(u) \leq h(v)+1$ for $(u,v) \in E_f$. Again, similarly to the previous case, this violation is only transient. A $push(b,a)$ operation becomes immediately applicable when the trace completes. And because $f(a,b) = 0$ before $push(a,b)$, the same amount of flow sent to b by $push(a,b)$ will be returned to a by $push(b,a)$, thus removing (b,a) from E_f and hence the requirement that $h(b) \leq h(a)+1$. If the algorithm terminates, then $push(b,a)$ must have already been executed, then we will have $h(u) \leq h(v)+1$ for $(u,v) \in E_f$.

6. The execution of $push(a,b)$ and $lift(a)$ are interleaved. This can never happen because according to Lemma 2 either a lift or a push can be applied to an overflowing vertex, but not both.

7. The execution of $push(a,b)$ and $push(b,a)$ are interleaved. This cannot happen either. $push(a,b)$ is applied when $h(a) > h(b)$. $push(b,a)$ is applied when $h(b) > h(a)$. The two conditions conflict.

8. The execution of more than two operations are interleaved. Because the discussion is similar to the above and the conclusion is the same, details are omitted here. □

The next lemma gives an important property of the algorithm.

Lemma 6. *If the algorithm terminates, then there is no path from s to t in the residual graph G_f when the algorithm* *terminates. Here f is the flow function calculated by the algorithm.*

Proof: Assume for the sake of contradiction that there is a path from s to t in G_f when the algorithm terminates. Without loss of generality, suppose this is a simple path consisting of $s \rightarrow u_1 \rightarrow ... \rightarrow u_k \rightarrow t$ where $k \leq |V| - 2$.

Each edge along the path is in E_f, then according to Lemma 5, we have $h(s) \leq h(u_1) + 1$, $h(u_1) \leq h(u_2) + 1$, ... $h(u_k) \leq h(t) + 1$. Combining these inequalities together, we have $h(s) \leq h(t) + |V| - 1$. But this contradicts the fact that $h(s) = |V|$ and $h(t) = 0$ are never changed throughout the algorithm. □

The next theorem shows that if the algorithm terminates, it finds the maximum flow.

Theorem 1. *Given graph G, if the algorithm terminates, then the calculated function f is a maximum flow for G.*

Proof: if the algorithm terminates, then we must have $e(u) = 0$ for $u \in V - \{s, t\}$ because otherwise according to Lemma 2, either a push or a lift is applicable at u, then the algorithm has not terminated yet. $e(u) = 0$ for $u \in V - \{s, t\}$ makes the calculated f a feasible solution to the maximum flow problem as all three constraints have been satisfied.

Lemma 6 says that there is no path from s to t in G_f. According to the maximum-flow minimum-cut theorem [3], f must be a maximum flow in G. □

6 Complexity Bound of the Algorithm

In this section, we show that the algorithm indeed terminates: it executes at most $O(|V|^2|E|)$ push/lift operations for a given graph $G(V, E)$. Note that the complexity is analyzed in the number of push and lift operations rather than in the execution time. This is because the algorithm is executed by multiple threads simultaneously. The time complexity depends on multiple factors including the number of threads and the assignment of vertices to the threads. The total number operations is therefore a more concrete measure of the complexity of the algorithm.

We first set a bound on the height of the vertices, which is then used to bound the number of lift and push operations.

Lemma 7. *During the execution of the algorithm, for any vertex u s.t. $e(u) > 0$, there exists a path from u to s in the residual graph G_f.*

Proof: Assume for the sake of contradiction that there exists a vertex u such that $e(u) > 0$ but there is no path from u to s in G_f. Let $U = \{v$: there exists a simple path from u to v in $G_f\}$ and $\overline{U} = V - U$.

Consider an edge (v, w) where $v \in U$ and $w \in \overline{U}$. We must have $f(v, w) \leq 0$ because otherwise $c_f(v, w) = c_{vw} - f(v, w) > 0$ implies $(v, w) \in E_f$, then w can be reached by u, contradicting the selection of w.

It is fairly easy to show that $\sum_{v \in U} e(v) = \sum_{x \in \overline{U}, y \in U} f(x, y)$. Since every such $f(x, y) \leq 0$, we must have $\sum_{v \in U} e(v) \leq 0$. On the other hand, during the execution of the algorithm, $e(v)$ never goes negative for any $v \in V$. So we must have $e(v) = 0$ for every $v \in U$, including $e(u)$, but this contradicts the assumption that $e(u) > 0$. □

Lemma 8. *Given graph G, source vertex s, and sink vertex t, then during the execution of the algorithm, if $e(u) > 0$, then there exists a path $u_1 \rightarrow u_2 ... \rightarrow u_k$ in the residual graph from u to s ($u_1 = u$, $u_k = s$) and $h(u_i) \leq h(u_{i+1}) + 1$ for $i = 1, ..., k - 1$.*

Proof:

If $e(u) > 0$, according to Lemma 7, if $e(u) > 0$, then there exits a path from u to s in the residual graph. Let the path be $v_1 \rightarrow v_2 ... \rightarrow v_m$ where $v_1 = u$ and $v_m = s$.

Note that we may not have $h(v_i) \leq h(v_{i+1} + 1)$ for $i = 1, ..., j - 1$. Without loss of generality, assume (v, w) is the first edge along the path that exhibits $h(v) > h(w) + 1$.

As we have discussed in the proof for Lemma 5, the coexistence of $h(v) > h(w) + 1$ and $(v, w) \in E_f$ can only be the result of the interleaved execution of $push(v, w)$ and $lift(v)$, and the existence of residual edge (v, w) is only transient. A $push(v, w)$ becomes immediately applicable, which, upon completion, will remove (v, w) from E_f. Additionally, we will still have $e(v) > 0$ after $push(v, w)$ because $push(v, w)$ will not deplete all the excessive flow $e(v)$ at vertex v (refer to the proof of Lemma 5 for details). The removal of (v, w) from E_f and the fact that we still have $e(v) > 0$ after the removal indicates the existence of a path $v, w', ..., s$ from v to s whose first edge (v, w') satisfies $h(v) \leq h(w') + 1$.

Repeating the process, we are able to construct a path $u_1 \rightarrow u_2 ... \rightarrow u_k$ in E_f where $u_1 = u$, $u_k = s$, and $h(u_i) \leq h(u_{i+1}) + 1$ for $i = 1, ..., k - 1$.

□

With Lemma 8, we can show that the height of the vertices are bounded.

Lemma 9. *Given graph $G(V, E)$, source vertex s and sink vertex t, then during the execution of the algorithm, we always have $h(u) \leq 2|V| - 1$ for $u \in V$.*

Proof: After initialization, we have $h(s) = |V|$ and $h(t) = 0$, and these two are never updated by the algorithm.

The height of a vertex u is lifted only when $e(u) > 0$. If $e(u) > 0$, then according to Lemma 8 we have a path from u to s in the residual path. Let $u_1 \rightarrow u_2 ... \rightarrow u_k$ denote the path. (So $u = u_1$, $u_k = s$.) Without loss of generality, this is a simple path so $k \leq |V|$. According to Lemma 8, we have $h(u_1) \leq h(u_2) + 1$, ... $h(u_{k-1}) \leq$

$h(u_k) + 1$. Combine these inequalities together, we have $h(u) = h(u_1) \leq h(u_k) + |V| - 1 = h(s) + |V| - 1 = 2|V| - 1$. □

Now the height of the vertices has been bounded, we can derive the following lemmas whose proof are similar to that in [9] and thus omitted here.

Lemma 10. *Given graph $G(V, E)$ with source vertex s and sink vertex t, then during the execution of the algorithm, the total number of lift operations is less than $2|V|^2 - |V|$.*

Lemma 11. *Given graph $G(V, E)$ with source vertex s and sink vertex t, then during the execution of the algorithm, the number of saturating pushes is less than $(2|V| - 1)|E|$.*

Lemma 12. *Given graph $G(V, E)$ with source vertex s and sink vertex t, then during the execution of the algorithm, the number of non-saturating pushes is less than $4|V|^2|E|$.*

And the following theorem is derived immediately from Lemmas 10, 11, and 12.

Theorem 2. *Given graph $G(V, E)$ with source vertex s and sink vertex t, the algorithm executes $O(|V|^2|E|)$ push and lift operations.*

7 Discussion

In this paper, we presented a lock-free multi-threaded algorithm for the maximum network flow problem. The algorithm finds the maximum flow in $O(|V|^2|E|)$ time. This algorithm should not be considered as a conclusion for the lock-free solution to the target problem. Further improvement should be investigated in the following directions:

The termination of the algorithm has been proved theoretically. But it may be difficult to design a practical implementation to detect the termination without using locks. As shown in the Theorem 2, the algorithm terminates when no further push or relabel operations can be applied. However, the absence of applicable push or relabel operations at an individual vertex does not imply the termination, because other vertices may be active. Further more, another vertex may push flow to this idling vertex, making it active again. The termination of the algorithm, which becomes true only when we do not have any applicable push or relabel operations at any vertices, needs to be detected with the help of a global barrier. Barriers are implemented using locks, however. To derive a completely lock-free algorithm, further study is needed for the efficient detection of algorithm termination.

The complexity bound of $O(|V|^2|E|)$ needs to be (and we believe it can be) improved. The $O(|V|^2|E|)$ running time (in terms of the number of operations) is the same as the original sequential push-relabel algorithms. Previous studies have shown that the running can be greatly reduced by advanced data structures such as dynamic trees, or by techniques such as global relabeling. It is a challenging problem to improve the complexity of the algorithm and still keep it lock-free.

References

[1] R. J. Anderson and a. C. S. Jo˙ On the parallel implementation of goldberg's maximum flow algorithm. In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 168–177, New York, NY, USA, 1992. ACM.

[2] 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. In *PDCS '05: Proceedings of the 18th ISCA International Conference on Parallel and Distributed Computing Systems*, 2005.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms, 2nd Edition*. MIT Press, 2001.

[4] E. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.

[5] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *J. ACM*, 19(2):248–264, 1972.

[6] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[7] H. N. Gabow. Scaling algorithms for network problems. *J. Comput. Syst. Sci.*, 31(2):148–168, 1985.

[8] A. V. Goldberg. Recent developments in maximum flow algorithms (invited lecture). In *SWAT '98: Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, pages 1–10, London, UK, 1998. Springer-Verlag.

[9] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 136–146, New York, NY, USA, 1986. ACM.

[10] A. V. Goldberg and R. E. Tarjan. Finding minimum-cost circulations by successive approximation. *Math. Oper. Res.*, 15(3):430–466, 1990.

[11] J. JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[12] D. B. Johnson. Parallel algorithms for minimum cuts and maximum flows in planar networks. *J. ACM*, 34(4):950–967, 1987.

[13] A. V. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Mathematics Doklady*, 15:434–437, 1974.

[14] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, 1997.

[15] Y. Shiloach and U. Vishkin. An o(n^2 log n) parallel maxflow algorithm. *J. Algorithms*, 3(2):128–146, 1982.