An Asynchronous Multithreaded Algorithm for the Maximum Network Flow Problem with Nonblocking Global Relabeling Heuristic

Bo Hong, *Member*, *IEEE*, and Zhengyu He, *Student Member*, *IEEE*

Abstract—In this paper, we present a novel asynchronous multithreaded algorithm for the maximum network flow problem. The algorithm is based on the classical push-relabel algorithm, which is essentially sequential and requires intensive and costly lock usages to parallelize it. The novelty of the algorithm is in the removal of lock and barrier usages, thereby enabling a much more efficient multithreaded implementation. The newly designed push and relabel operations are executed completely asynchronously and each individual process/thread independently decides when to terminate itself. We further propose an asynchronous global relabeling heuristic to speed up the algorithm. We prove that our algorithm finds a maximum flow with $O(|V|^2|E|)$ operations, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. We also prove the correctness of the relabeling heuristic. Extensive experiments show that our algorithm exhibits better scalability and faster execution speed than the lock-based parallel push-relabel algorithm.

Index Terms—Maximum network flow, parallel algorithm, multithreading, lock-free.

1 Introduction

The maximum network flow (max-flow) problem is a fundamental graph theory problem with important applications in many areas. Flow networks can be used to model parts through pipelines, commodities through highways, information through computer networks, etc. They also have applications in other problems, such as computer vision (e.g., [1]).

The max-flow problem is defined as follows: A flow network is a graph $G(V,E)$, where edge $(u,v) \in E$ has capacity c_{uv}. For notational convenience, c_{uv} is set to 0, if $(u,v) \notin E$. 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. $f(u,v) \le c_{uv}$, 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 $|f| = \sum_{u\in V} f(s,u)$, which is the net amount of flow sent from s to t. The max-flow problem searches for a flow with the maximum value.

Both sequential and parallel algorithms have been studied for this problem. Early solutions to the maximum network flow problem are based on the augmenting path method due to Ford and Fulkerson [2], which by itself is pseudopolynomial and was later improved by carefully choosing the order in which augmenting paths are selected (e.g., the $O(|V||E|^2)$ algorithm by Edmonds and Karp [3], and the $O(|V|^2|E|)$ algorithm by Dinic [4]). The concept of preflow was introduced by Karzanov in [5], which leads to an $O(|V|^3)$ algorithm, the execution time was further improved in [6], [7]. Goldberg and Tarjan designed the push-relabel method [8] with $O(|V|^2|E|)$ operations and further improved the complexity bound by using specially designed data structures [9].

In addition to these sequential algorithms, parallel algorithms have also received a lot of attention. For example, the parallel algorithm due to Shiloach and Vishkin [10] runs in $O(|V|^2 \log|V|)$ time using a $|V|$-processor PRAM. Goldberg pointed out that the dynamic-tree-based algorithm in [8] can be implemented on an EREW PRAM, taking $O(|V|^2 \log|V|)$ time and $O(|V|)$ processors. 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 and synchronization overheads. For the related multicommodity flow problem, Awerbuch and Leighton developed approximation algorithms in [12], [13] using the potential method. These approximation algorithms feature local controls and support online execution in distributed networks, which differs from our work that focuses on multicore platforms and searches for the exact solution.

Practical implementations of parallel algorithms have also been investigated intensively. Anderson and Setubal [14] augmented the push-relabel algorithm with a *global relabeling* operation. Bader and Sachdeva [15] designed a parallel implementation using gap relabeling heuristic with considerations in the cache performance of the push-relabel algorithm. Both implementations have demonstrated good execution speed. These parallel implementations, however, share the common feature of using locks to protect every push and relabel operation *in its entirety*, which essentially

- *The authors are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332. E-mail: {bohong, zhengyu.he}@gatech.edu.*

Manuscript received 2 July 2009; revised 31 Dec. 2009; accepted 19 July 2010; published online 23 Aug. 2010.
Recommended for acceptance by O. Beaumont.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2009-07-0298.
Digital Object Identifier no. 10.1109/TPDS.2010.156.

sequentializes any two push/relabel operations whenever a common vertex is involved. Without lock protection, these implementations will fail to find the maximum flow. But locks are known to have expensive overheads [16]. Parallelism in these algorithms is therefore limited by the intensive lock usages, which can lead to performance degradation especially when the number of threads scales up. For applications in computer vision, Graphics Processing Unit (GPU) based parallel max-flow algorithms on GPUs have also been studied. In [17], a GPU-based algorithm is developed for a special type of graphs: 2D grid. The algorithm in [8] was implemented for GPU in [18]. The implementation requires strong synchronization to guarantee correctness and the reported performance is 5-10 times slower than [8].

In this paper, we present a new *asynchronous* parallel max-flow algorithm. The algorithm uses the push-relabel technique proposed in [8] but differs from existing algorithms in that the multiple processes/threads do not need any locks or barriers to protect the push and relabel operations. In addition, each thread also independently determines its own termination without using any locks or barriers. Upon the termination of the last thread, the algorithm finds the maximum flow with $O(|V|^2|E|)$ operations. We further develop a nonblocking global relabeling heuristic to speed up the execution of the algorithm. A preliminary version of the paper appeared in [19].

We implemented our algorithm using the pthread (the IEEE standard that defines an application programming interface for creating and manipulating threads) library and compared its performance with the sequential algorithm in [9] and the lock-based parallel algorithm in [14]. Five graph topologies and various sizes were tested. The results show that our algorithm is faster than the sequential algorithm by up to 89 percent (50 percent on average), and faster than the lock-based algorithm by up to 60 percent (32 percent on average). In addition to the absolute execution time, our algorithm exhibits better scalability than the lock-based algorithm. Experiments show that our algorithm completes significantly more (by up to 69 percent) operations per second than the lock-based parallel algorithm, which further verifies the effectiveness of using asynchronous operations in reducing the synchronization overheads.

The rest of the paper is organized as follows: Section 2 presents the model of the target computing platform. The algorithm is described in Section 3, followed by the optimality and complexity analysis in Section 4. Section 5 presents the nonblocking global relabeling heuristic. Experimental results are shown in Section 6. Discussions are provided in Section 7.

2 The Target Multiprocessor Platform

We assume that the target multiprocessor platform consists of multiple processors that access a shared memory. Both Symmetric Multiple Processor (SMP) systems and the recently emerging multicore processors are examples of such platforms. We assume that the platform supports atomic 'read-modify-write' instructions, as most modern parallel architectures do. Atomic 'read-modify-write' instructions allow the platform to sequentialize such instructions automatically without using locks. 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.

3 The Asynchronous Multithreaded Algorithm

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

Given a directed graph $G(V, E)$, function f is called a flow, if it satisfies the three constraints in Section 1. 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 vertex $u \in V$, $e(u)$ is defined as $e(u) = \sum_{w \in V} f(w, u)$, which is the net flow into vertex u. Constraint 3 in the problem statement requires $e(u) = 0$ for $u \in V - \{s, t\}$. But before our algorithm terminates, we may have $e(u) > 0$ for some vertices (which will turn 0 upon termination of the algorithm). We say vertex $u \in V - \{s, t\}$ is *overflowing*, if $e(u) > 0$. When overflowing vertices exist, we call f a preflow. An integer-valued height function $h(u)$ is also defined for every vertex $u \in V$. We say u is higher than v, if $h(u) > h(v)$. We follow the definition in [8] and say h is a valid height function, if $(u, v) \in E_f$ implies $h(u) \le h(v) + 1$. We call $(u, v) \in E_f$ a *regular* residual edge, if $h(u) \le h(v) + 1$. A path in E_f is a regular path, if it only consists of regular residual edges. We call $(u, v) \in E_f$ a *special* residual edge, if $h(u) > h(v) + 1$.

The algorithm is listed below:

Algorithm 1. The Asynchronous Max-flow Algorithm
```
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
```

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

- *Initializeh*(u), $e(u)$, and $f(u, v)$:

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

- $Push(u, v')$: applies if u is overflowing, and $\exists v \in V$ s.t. $(u, v) \in E_f$ and $h(u) > h(v)$.

$$\begin{aligned} &v' \leftarrow \mathrm{argmin}_v[h(v) \mid c_f(u, v) > 0 \text{ and } h(u) > h(v)] \\ &d \leftarrow \min[e(u), c_f(u, v')] \\ &f(u, v') \leftarrow f(u, v') + d \\ &f(v', u) \leftarrow f(v', u) - d \\ &e(u) \leftarrow e(u) - d \\ &e(v') \leftarrow e(v') + d \end{aligned}$$

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

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

The algorithm differs from the original push-relabel algorithm in the following two aspects: 1) the push operation sends flow to the *lowest* neighbor in G_f, and 2) the termination condition examines the value of $e(s) + e(t)$ instead of the existence of overflowing vertices. These modifications allow the algorithm to be executed *asynchronously* by multiple threads, which constitutes the major contribution of this paper.

The asynchronous execution is better explained through the programming implementation of the algorithm as shown in Program 2. Without loss of generality, we assume that for each vertex $u \in V$ there is one thread responsible of executing $push(u, v')$ and $lift(u)$. We will use u to denote both vertex u and the thread responsible for vertex u. Note that the number of available threads is typically smaller than the number of vertices, in which case one thread will be used for multiple vertices. Due to space limit, the vertex-thread allocation problem and other implementation issues are addressed in the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.156.

After the initialization step, thread u executes the code in Program 2, where e', v', h', h'', and d are per-thread private variables and $h(u)$, $e(u)$, and $c_f(u, v)$ [$c_f(u, v) = c_{uv} - f(u, v)$] are shared among all threads.

Program 2. Program Implementation of Algorithm 1

```
 1: while $e(s) + e(t) < 0$ do
 2:    if $e(u) > 0$ then
 3:       $e' \leftarrow e(u)$
 4:       $h' \leftarrow \infty$
 5:       for all $(u, v) \in E_f$ do
 6:          $h'' \leftarrow h(v)$
 7:          if $h'' < h'$ then
 8:             $v' \leftarrow v$
 9:             $h' \leftarrow h''$
10:          end if
11:       end for
12:       if $h(u) > h'$ then
13:          $d \leftarrow \min(e', c_f(u, v'))$
14:          $c_f(u, v') \leftarrow c_f(u, v') - d$     #executed atomically
15:          $c_f(v', u) \leftarrow c_f(v', u) + d$     #executed atomically
16:          $e(u) \leftarrow e(u) - d$                #executed atomically
17:          $e(v') \leftarrow e(v') + d$              #executed atomically
18:       else
19:          $h(u) \leftarrow h' + 1$
20:       end if
21:    end if
22: end while
```

In Program 2, we assume that updates to shared variables $c_f(u, v')$, $c_f(v', u)$, $e(u)$, and $e(v')$ (lines 14-17) are executed atomically by the architecture due to the support of atomic "read-modify-write" instructions. Line 1 decides whether a thread should terminate or not. Next, the thread finds the lowest neighbor v' for vertex u in E_f (lines 5-11). Based on the height of v', the thread executes either $push(u, v')$ (lines 14-17) or $lift(u)$ (line 19).

While thread u is executing the above code for vertex u, each of the other threads is executing the same code for its own vertex. In our algorithm, the progress at one thread does not need to synchronize with any other threads. Such a property exposes maximum parallelism in the execution of the algorithm. This asynchronous parallel execution is fundamentally different from existing parallel push-relabel algorithms, where both u and v need to be locked for $push(u, v)$ and u needs to be locked for $lift(u)$—without such lock protections these algorithms will fail to find the maximum flow.

4 Correctness and Complexity Bound of the Algorithm

In the original push-relabel algorithm, 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 overflowing. Furthermore, another vertex may push flow to this idling vertex, making it overflow again. The termination of the algorithm, which becomes true only when there do not exist any applicable push or relabel operations at any vertices, requires a global barrier if implemented in a brute-force manner.

The following lemma shows that our termination condition $e(s) + e(t) < 0$ is equivalent to the existence of applicable push and lift operations.

Lemma 1. *Throughout the execution of the Algorithm 1, $e(s) + e(t) = 0$ if and only if there does not exist any overflowing vertices.*

Proof. $f(u, v) = -f(v, u)$ is always maintained throughout the algorithm, which easily leads to $\sum_{u,v \in V} f(u, v) = 0$. Therefore,

$$\begin{aligned} &\sum_{u,v \in V} f(u, v) \\ &= \sum_{u \in V} f(u, s) + \sum_{u \in V} f(u, t) + \sum_{u \in V, v \in V - \{s,t\}} f(u, v) \\ &= e(s) + e(t) + \sum_{u \in V - \{s,t\}} e(u) \\ &= 0. \end{aligned}$$

Because e(u) is always non-negative for $u \in V - \{s, t\}$ during the execution the algorithm (the only operation that reduces $e(u)$ is $push(u, v)$, which always leaves $e(u)$ non-negative), so it is obvious that as long as there exists any overflowing vertices, we always have $e(s) + e(t) < 0$.

Consequently $e(s) + e(t) = 0$ implies that $e(u) = 0$ for all $u \in V - \{s, t\}$. □

Lemma 1 eliminates barrier usages for our algorithm. For any thread u, to determine whether it can terminate or not, it checks the summation of $e(s)$ and $e(t)$ instead of the existence of any overflowing vertices. Even though the two conditions are mathematically equivalent, the latter requires a global barrier at all the threads for a successful detection, while the former can be examined by each thread individually. Note that $e(s)$ and $e(t)$ are modified only by neighbors of s and t. Consequently, multithreaded implementation of the algorithm will find $e(s)$ and $e(t)$ to be read-only by most threads, resulting in little contention when updating $e(s)$ and $e(t)$. For those threads that need to update $e(s)$ and $e(t)$, the update will be executed efficiently through the atomic "read-add-write" instruction supported by the architecture.

4.1 Proof of Correctness

We start with the following observations on the algorithm: even though the asynchronous execution at multiple threads may be interleaved in an arbitrary order, the result of the execution actually reduces to that of just two equivalent orders.

We define the "*consequence*" of a $push(u, v')$ to be the values of $e(u)$, $e(v')$, $c_f(u, v')$, and $c_f(v', u)$ 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 3-13 and lines 14-17. Lines 3-13 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 14-17 update the shared variables accordingly, which we call the *action* stage of the push. Similarly, the trace of a single lift operation can also be split into two stages: lines 3-12 and line 19. Lines 3-12 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 19 updates the vertex height, which is defined as the "*action*" stage of the lift.

Now we present the following predefined traces:

1. a *stage-clean trace* where multiple operations do not have any overlapping in their executions. For example, if a trace contains a push and a lift, and the push completes in its entirety before the lift starts, then this is a stage-clean trace.
2. a *stage-stepping trace* where all the operations execute their preparation stages before any one proceeds with its action stage.

With the above notational preparation, we have:

Lemma 2. *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 2 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. The proof is similar to that for Lemma 2 and omitted here.

Lemma 3. *For any trace of three or more push and/or lift operations, there exists an equivalent trace consisting of a sequence of nonoverlapping traces, each of which is either stage-clean or stage-stepping.*

With Lemmas 2 and 3, we can greatly simplify our discussion by focusing on stage-clean and stage-stepping traces rather than arbitrarily interleaved operations.

Lemma 4. *During the execution of Algorithm 1, if u is an overflowing vertex, then either a push or a $lift$ operation applies to it.*

Proof. If a push operation does not apply to u, we must have $h(u) \leq h(v)$ for all $(u, v) \in E_f$, then $lift(u)$ is applicable. □

Lemma 5. *During the execution of Algorithm 1, vertex height never decreases.*

Proof. Only the lift operation can change the height of a vertex. When vertex u is about to be lifted, we have $h(u) \leq h(v)$ for all vertices v, such that $(u, v) \in E_f$. So $h(u) < min\ h(v) | (u, v) \in E_f + 1$, and the lift operation increases $h(u)$. □

Lemma 6. *During the execution of Algorithm 1, there is no regular path (defined in Section 3) from the source s to the sink t in the residual network G_f.*

Proof. Assuming for the sake of contradiction that there exists a regular path $p = \{v_0, v_1, \ldots, v_k\}$ from s to t, where $v_0 = s$ and $v_k = t$. Without loss of generality, p is a simple path, so $k < |V|$. Because p is a regular path, we have $h(v_i) \leq h(v_{i+1})$ for $i = 0, 1, \ldots, k-1$. Combining these inequalities together, we have $h(s) \leq h(t) + k$. Because $h(t) = 0$, we have $h(s) \leq k < |V|$, which contradicts the fact that $h(s) = |V|$ and is never changed throughout the algorithm. □

Now we examine under what condition special residual edges (defined in Section 3) may appear, and what will happen thereafter. Suppose, we have two vertices a and b in V. After the initialization step and before any push or lift operations, h is a valid height function. Thus, initially (a, b) will be a regular residual edge if $(a, b) \in E_f$, so will (b, a).

If a push or lift operation is executed in its entirety without being interleaved with each other, or if the interleaved execution of multiple operations is equivalent to a stage-clean trace, then the scenario is the same as the original push-relabel algorithm. For this scenario, h is trivially maintained as a valid height function and all the residual edges remain as regular residual edges.

When we have stage-stepping traces, the situation is more complicated and the detailed discussion is presented in the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.156. The analysis consists of eight cases. It shows that (b, a) may become a special residual edge when $push(a,b)$ and $lift(b)$ are interleaved and $(b,a) \notin E_f$ before the action stage of $push(a,b)$. Once (b,a) becomes a special residual edge, a $push(b,a)$ immediately becomes available, which, upon completion, will remove (b,a) from E_f. If the algorithm terminates, then such following-up push operations must have already been applied (otherwise the algorithm would not terminate), and therefore, we do not have any special residual edges upon algorithm completion. This leads to the following theorem:

Theorem 1. *If Algorithm 1 terminates, then the preflow f it computes is a maximum flow.*

Proof. By Lemma 1, if the algorithm terminates, there do not exist any overflowing vertices. This implies that all the special residual edges, if any, have disappeared—otherwise according to the above analysis there will be applicable push operations.

It is easy to verify that the preflow f satisfies constraints 1 and 2 of the max-flow problem throughout the execution of Algorithm 1. If the algorithm terminates, then no vertex is overflowing and f also satisfies constraint 3. So f is a valid flow at the termination of Algorithm 1.

We claim that there is no path from s to t in the residual network $G_f(V, E_f)$ upon completion of Algorithm 1. Suppose for the sake of contradiction there is a path $p = v_0, v_1, \ldots, v_k$ from s to t in G_f, where $v_0 = s$ and $v_k = t$. Without loss of generality, this is a simple path, so $k \leq |V| - 1$. Because G_f only contains regular residual edges upon completion of Algorithm 1, we have $h(v_i) \leq h(v_{i+1})$ for $i = 0, 1, \ldots, k-1$. Combining these inequalities together, we have $h(s) = h(v_0) \leq h(v_k) + k \leq h(t) + |V| - 1 = |V| - 1$. But $h(s) = |V|$ initially and never changes.

By the max-flow min-cut theorem [20], when there is no path from s to t in the residual network G_f, f is a maximum flow. □

4.2 Complexity Bound

To show that Algorithm 1 indeed terminates, we show that the algorithm executes at most $O(|V|^2|E|)$ push/lift operations for a given graph $G(V,E)$.

Lemma 7. *Throughout the execution of Algorithm 1, for any vertex $u \in V - \{s,t\}$, if $e(u) > 0$, then there is a simple path p from u to s in the residual graph, and all the edges along path p are regular residual edges.*

Proof. We prove by constructing such a path.

There may be regular and special residual edges in E_f. As shown by the analysis in the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.156, (b,a) becomes a special residual edge only when $push(a,b)$ and $lift(b)$ are interleaved and $(b,a) \notin E_f$ before the action stage of $push(a,b)$. Once (b,a) becomes a special residual edge, a $push(b,a)$ immediately becomes available, which, upon completion, will remove (b,a) from E_f. For each such special residual edge, we can perform the corresponding push operation and eliminate it from the residual graph. We denote the remaining residual edges as E_{fr}. Because the removal of the special residual edges does not add any new residual edges, E_{fr} is a subset of E_f that only consists of regular residual edges. Note that the removal of the special residual does not change any vertex heights.

For an overflowing vertex u, we construct the path from u to s with edges in E_{fr}. Let $U = \{v$: there exists a simple path from u using edges in $E_{fr}\}$, and suppose for the sake of contradiction that $s \notin U$. Let $\overline{U} = V - U$.

For each pair of vertices $w \in \overline{U}$ and $v \in U$, we must have $f(w,v) \leq 0$ because otherwise $f(w,v) > 0$ implies $f(v,w) < 0$, and hence, $c_f(v,w) = c_{vw} - f(v,w) > 0$, which means (v,w) is a residual edge. Hence, there exists a simple path of the form $u \rightsquigarrow v \rightarrow w$. But this contradicts the choice of w.

It can be shown easily that

$$\sum_{x \in U} e(x) = \sum_{y \in \overline{U}, z \in U} f(y,z) \leq 0.$$

However, $e(x)$ never becomes negative, so we must have $e(x) = 0$ for any vertex $x \in U$. In particular, we have $e(u) = 0$. This contradicts the assumption that u is overflowing. Therefore, we must have $s \in U$, which means there exists a simple path from u to s using edges in E_{fr}, which is a subset of E_f that consists of regular residual edges only. □

Lemma 8. *During the execution of Algorithm 1, $h(u) \leq 2|V| - 1$ for all vertices $u \in V$.*

Proof. s and t are never lifted in Algorithm 1. So we always have $h(s) = |V|$ and $h(t) = 0$, both of which are no greater than $2|V| - 1$.

Consider any vertex $u \in V - \{s,t\}$. $h(u) = 0 \leq 2|V| - 1$ initially. $h(u)$ increases for each time $lift(u)$ is applied. Note that $e(u) > 0$ after each $lift(u)$ (otherwise $lift(u)$ will not be applied). According to Lemma 7, there exists a simple path $p = v_0, v_1, \ldots, v_k$ from u to s, where $v_0 = u$, $v_k = s$, and $k \leq |V| - 1$, and all the edges along p are regular residual edges. We therefore have $v_i \leq v_{i+1} + 1$ for $i = 0, 1, \ldots, k-1$. Combining these inequalities together, we have $h(u) = h(v_0) \leq h(v_k) + k = h(s) + k = 2|V| - 1$. □

From this point onward, the complexity analysis is identical to that in [8]. The upper bound on h can be used to further bound the number of lift and push operations. We omit the detailed analysis and present the final theorem directly:

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

5 Asynchronous Global Relabeling Heuristic

Previous studies suggested two heuristics, Global Relabeling and Gap Relabeling, to improve the practical performance of the push-relabel algorithm. The height h of a vertex helps the algorithm to identify the direction to push the flow toward the sink or the source. Global Relabeling heuristic updates the heights of the vertices with their shortest distance to the sink [9]. The Gap Relabeling heuristic due to Cherkassky also improves the practical performance of the push-relabel method (though not as effective as Global Relabeling [9]). It discovers the overflowing vertices from which the sink is not reachable and then lift these vertices to $|V|$ to avoid unnecessary further operations.

In sequential push-relabel algorithms, Global Relabeling and Gap Relabeling are executed by the same single thread that executes the push and lift operations. Race conditions therefore do not exist. For parallel push-relabel algorithms, the Global Relabeling and Gap Relabeling have been proposed by Anderson [14] and Bader [15], respectively. Both heuristics lock the vertices to avoid race conditions: the global or gap relabeling, push, and lift operations are therefore pair-wise mutually exclusive.

In this paper, we develop a new Asynchronous Global Relabeling (AGR) heuristic to speed up our asynchronous algorithm. The execution of our heuristic can be arbitrarily interleaved with the push operations, which is fundamentally different from the existing parallel relabeling heuristics. With the AGR heuristic, the algorithm dedicates one thread to the execution of the heuristic, while other (multiple) threads simultaneously execute the push and lift operations. To amortize the computational cost, the AGR heuristic is applied periodically after a certain number of push and lift operations.

Due to space limit, details of the AGR heuristic, including both its programming implementation and the proof of correctness, are presented in the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.156.

6 Experimental Results

To evaluate the performance of the proposed asynchronous algorithm, a multithreaded implementation was developed using C and the pthread library. The atomic read-modify-read operation in our algorithm was implemented by using atomic fetch-and-add instructions supported by the experiment computer architectures. The sequential push-relabel algorithm in [9] provides the baseline for performance evaluation. Note that the algorithm in [9] only uses the push-relabel method to find out the value of the maximum flow value and then construct a valid maximum flow using the preflow method, which is of $O(|V||E|)$ complexity. We also compared the performance of our algorithm against multithreaded lock-based push-relabel algorithms described in [14].

The following programs were implemented for the experiments:

1. **amf**: Our multithreaded asynchronous push-relabel algorithm with asynchronous global relabeling. FIFO order is maintained for each local queue. As proved in Section 4.2, the complexity of the algorithm is $O(|V|^2|E|)$.
2. **lmf**: The lock-based implementation of multithreaded push-relabel algorithm with concurrent global relabeling due to Anderson [14]. Each thread processes its vertices in a FIFO order. This complexity is $O(|V|^2|E|)$.
3. **q_prf**: The sequential push-relabel algorithm due to Goldberg [9], with global relabeling. Vertices are processed in FIFO order. The complexity of this algorithm is $O(|V|^3)$ [9].
4. **hi_pr**: The sequential push-relabel algorithm with global relabeling and gap relabeling. In this implementation, vertices are processed in the descending order of their heights, which leads to a complexity of $O(|V|^2\sqrt{|E|})$ [21]. This is currently the fastest sequential implementation of push-relabel algorithm that we are aware of.

We conducted experiments on both Intel and AMD platforms. The Intel platform has four Six-Core Xeon E7450 processors running at 2.4 GHz with 64 GB DDR2 800 MHz memory. The AMD platform has four 64 bit Quad-Core Opteron 8358 processors running at 2.4 GHz with 64 GB DDR2 667 Hz memory. The operating system (Redhat Enterprise 5) ran Linux kernel version 2.6.18. The gcc version 4.1.2 was used to generate the executables.

Five types of graphs were used in the experiments. These graphs were used in the first DIMACS Implementation Challenge [22]. Featuring a variety of topologies and sizes, these graphs were used by many existing max-flow algorithms to test their performance (e.g., [9], [14], [15]).

1. **Acyclic-dense graphs**: These are complete directed acyclic-dense graphs: each vertex is connected to every other vertex. We tested graphs of 2,000 and 4,000 vertices.
2. **Genrmf-long graphs**: These graphs consist of l_1 square grids of vertices (frames) each having $l_2 \times l_2$ vertices. The source is at a corner of the first frame, and the sink is at the opposite corner of the last frame. Each vertex is connected to its grid neighbors within the frame and to one vertex randomly chosen from the next frame. We tested the graphs of $l_1 = 256, l_2 = 32$ (262,144 vertices and 1,276,928 edges) and $l_1 = 512, l_2 = 32$ (524,288 vertices and 2,554,880 edges).
3. **Genrmf-wide graphs**: The topology is the same as genrmf-long graphs except for the values of l_1 and l_2. Frames are bigger in Genrmf-wide graphs than in Genrmf-long graphs. We tested the graphs of $l_1 = 64, l_2 = 64$ (262,144 vertices and 1,290,240 edges) and $l_1 = 128, l_2 = 64$ (524,288 vertices and 2,584,576 edges).
4. **Washington-RLG-long graphs**: These graphs are rectangular grids of vertices with w rows and l columns. Every vertex in a row has three edges connecting to random vertices in the next row. The source and the sink are external to the grid, the source has edges to all vertices in the top row, and all vertices in the bottom row have edges to the sink. We tested the graphs of $w = 256, l = 512$ (131,074 vertices and 392,960 edges) and $w = 256, l = 768$ (196,610 vertices and 589,568 edges).
5. **Washington-RLG-wide graphs**: Same as Washington-RLG-long graphs except for the values of w and l. Each row in the Washington-RLG-wide graphs are wider. We tested the graphs of $w = 512, l = 256$

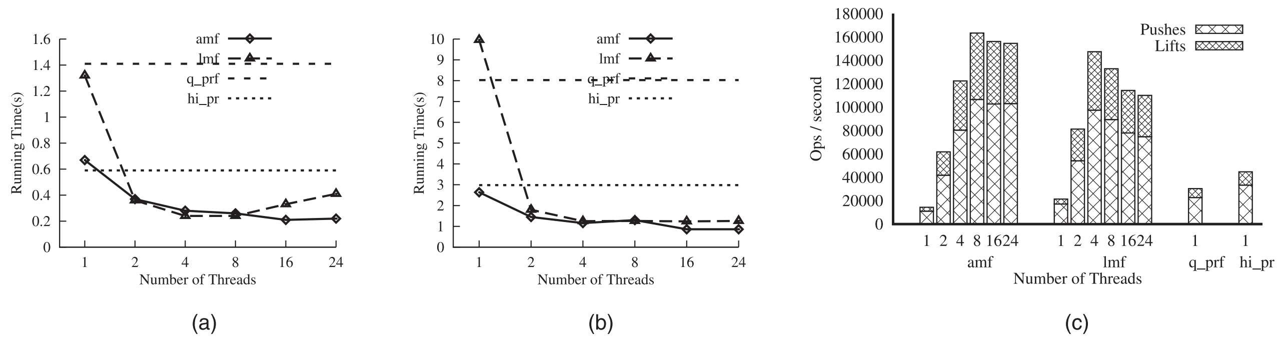

Fig. 1. Experimental results on acyclic-dense graphs. (a) 2,000 nodes (Intel). (b) 4,000 nodes (Intel). (c) Operation efficiency (Intel).

(131,074 vertices and 392,704 edges) and $w = 768, l = 256$ (196,610 vertices and 589,056 edges).

We compared the performance of the four algorithms on the five input graph topologies. For each type of graphs, 50 instances were generated using different seeds for the pseudorandom number generator. For each algorithm, each instance was executed three times. The execution result reported for each algorithm was averaged over the 150 runs.

For each set of experiments, we report the execution time of the four algorithm. However, execution time by itself does not describe all the aspects of a parallel push-relabel algorithm. The number of operations depends on the vertex ordering scheme as well as the input graphs. For example, we observed that on acyclic-dense graphs, **hi_pr** executed more operations than our **amf** algorithm but less operations than **q_prf**. Furthermore, due to concurrent executions at multiple threads, parallel algorithm (both **amf** and **lmf**) cannot keep strict FIFO orders. Parallel algorithms therefore may execute more operations than the sequential algorithms. For the above reasons, to demonstrate the benefit of asynchronous executions, we also evaluated the four algorithms in terms of their operation efficiency, which was calculated as the total number of push and lift operations executed by all threads in one unit of time.

Due to space limit, here we only show experimental results of Acyclic-dense, Genrmf-long, and Washington-RLG-long graphs on the Intel system. The complete set of experiments is presented in the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.156.

Fig. 1 shows the experiment results on acyclic-dense graphs. For such graphs, the **q_prf** algorithm and the **hi_pr** algorithm have effectively the same complexity bound $[O(|V|^3) \simeq O(|V|^2\sqrt{|E|})]$, and both are lower than that of the **amf** and **lmf** algorithms $[O(|V|^2|E|)]$. But when multiple threads were used, the parallel algorithms were able to outperform the sequential algorithms on both Intel and AMD platforms (except for a few cases with the **lmf** algorithm). This demonstrates that parallel algorithm works well for such input graphs.

Fig. 1 also shows that our **amf** algorithm outperforms the **lmf** algorithm in terms of both execution time and operation efficiency. Fig. 1c shows that our **amf** algorithm executed 21 percent more operations per second than the **lmf** algorithm. This demonstrated the advantage of our algorithm in executing the individual push/lift operations asynchronously.

Furthermore, we also observed that as the number of threads increases to larger than eight, the execution time of the **lmf** algorithm increased for graphs with 2,000 vertices (from 0.34 to 0.74 s on the AMD system (figure presented in the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.156) with 16 threads, and from 0.24 to 0.33 s on the Intel system with 16 threads). We conjecture this is due to the deteriorated lock contentions caused by the large number of threads. On the other hand, our algorithm outperformed the **q_prf** and **hi_pr** algorithms whenever more than two threads were used, and the execution time reduced each time when we increased the number of threads. When the graphs have 4,000 vertices, the threads were less likely to compete for the same vertices, so we did not observe the increase in the execution time for **lmf**. But we expect to observe similar behavior when we further increase the number of threads (as future processors may support). This demonstrated that, by avoiding lock usages, our **amf** algorithm demands less system resources and is therefore able to support more threads on a given hardware platform before losing scalability.

Fig. 2 shows the results for the Genrmf-long graphs. Our **amf** algorithm scaled well and outperformed **q_prf** when more than four threads were used. When the number of threads reached 16, **amf** even outperformed **hi_pr** which has a lower complexity bound, especially when $\sqrt{|E|} \ll |V|$ for these Genrmf graphs. For Genrmf-long graphs, **amf** achieved an operation efficiency of 1.2×10^7 ops/s, which is 1.66 times higher than **hi_pr**'s 4.4×10^6 ops/s. For Genrmf-wide graphs (figure presented in the supplementary file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.156), **amf** achieved an operation efficiency of 7.9×10^6 ops/s, while **hi_pr** only achieved 4.2×10^6 ops/s.

From Fig. 2, we can also observe that our **amf** algorithm scales well, while the lock-based **lmf** algorithm saw increased execution time when the number of threads exceeded 16. The **lmf** algorithm also exhibited significantly lower operation efficiency (up to 59 percent lower than our **amf** algorithm). These results demonstrated the effectiveness of avoiding lock usages in our algorithm design.

On the Washington-RLG graphs, **amf** demonstrated similar scalability and absolute speed up as on the Genrmf

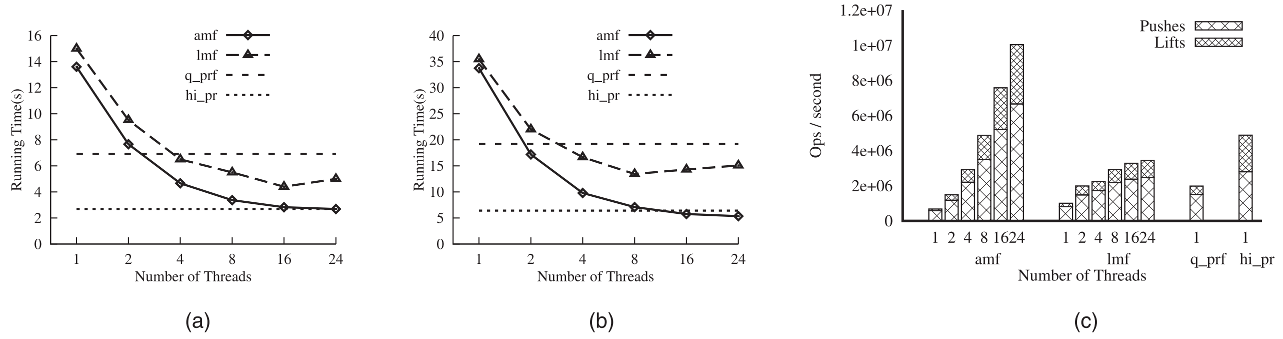

Fig. 2. Experimental results on Genrmf-long graphs. (a) 262,144 nodes (Intel). (b) 524,288 nodes (Intel). (c) Operation efficiency (Intel).

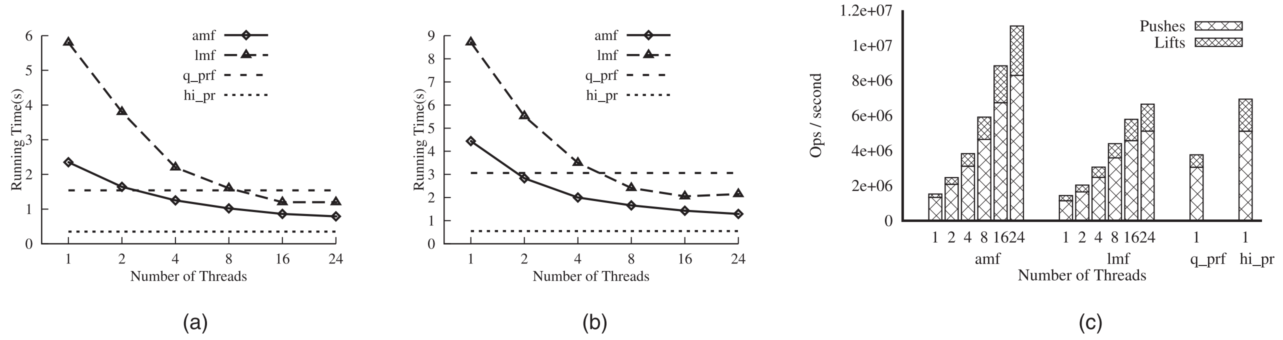

Fig. 3. Experimental results on Washington-RLG-long graphs. (a) 131,074 nodes (Intel). (b) 196,608 nodes (Intel). (c) Operation efficiency (Intel).

graphs (Figs. 3). When we had more than 16 threads, **amf** was three times faster than **q_prf** and even approached the execution time of **hi_pr**, which has a lower complexity bound.

In summary, the experiments show that unlike the lock-based **lmf** algorithm that often failed to scale when the number of threads exceeded certain threshold, our **amf** algorithm scales well as the number of threads increases up to the maximum number supported by the test platforms. Our algorithm also demonstrated absolute speed up over the well-known sequential push-relabel algorithms for certain graphs, in spite of the fact that our algorithm has a higher complexity bound in terms of the total number of operations.

7 Discussion and Conclusion

In this paper, we presented an $O(|V|^2|E|)$ asynchronous multithreaded push-relabel algorithm for the max-flow problem. The algorithm features lock-free push and relabel operations. We further developed an asynchronous global relabeling heuristic to speed up our algorithm. Experimental results demonstrated the effectiveness of the algorithm. We expect the algorithm to be particularly useful for multicore systems.

We further experimented with a relaxed push operation, in which $push(u, v)$ pushes to any lower neighbor v when $h(u) \leq |V|$, and only pushes to the lowest neighbor when $h(u) > |V|$. The relaxation intends to "allow" flow to be freely moved among vertices reachable from the sink (when $h(u) \leq |V|$), and "revokes" the freedom only when flow may be incorrectly returned to the source (when $h(u) > |V|$). The relaxation improved the execution speed by $\sim$50%—mainly due to the reduced workload of the preparation stages of push operations (searching for any lower neighbor is simpler than searching for the lowest neighbor). Correctness of the relaxation was experimentally verified and we plan to investigate the mathematical proof of the correctness of this relaxation.

Acknowledgments

This research work was supported by the US National Science Foundation (NSF) under award no. CNS-0845583 and in part by an equipment grant from Nvidia Inc.

References

[1] Y. Boykov and V. Kolmogorov, "An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 26, no. 9, pp. 1124-1137, Sept. 2004.

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

[3] J. Edmonds and R.M. Karp, "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems," *J. ACM,* vol. 19, no. 2, pp. 248-264, 1972.

[4] E. Dinic, "Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation," *Soviet Math. Doklady,* vol. 11, pp. 1277-1280, 1970.

[5] A.V. Karzanov, "Determining the Maximal Flow in a Network by the Method of Preflows," *Soviet Math. Doklady,* vol. 15, pp. 434-437, 1974.

[6] H.N. Gabow, "Scaling Algorithms for Network Problems," *J. Computer and System Sciences,* vol. 31, no. 2, pp. 148-168, 1985.

[7] A.V. Goldberg and R.E. Tarjan, "Finding Minimum-Cost Circulations by Successive Approximation," *Math. Operations Research,* vol. 15, no. 3, pp. 430-466, 1990.

[8] 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.

[9] A.V. Goldberg, "Recent Developments in Maximum Flow Algorithms (Invited Lecture)," *Proc. Sixth Scandinavian Workshop Algorithm Theory (SWAT '98),* pp. 1-10, 1998.

[10] Y. Shiloach and U. Vishkin, "An O(n^2 log n) Parallel Max-Flow Algorithm," *J. Algorithms,* vol. 3, no. 2, pp. 128-146, 1982.
[11] J. JáJá, *An Introduction to Parallel Algorithms.* Addison Wesley Longman Publishing Co., Inc., 1992.
[12] B. Awerbuch and T. Leighton, "A Simple Local-Control Approximation Algorithm for Multicommodity Flow," *Proc. 34th Ann. Symp. Foundations of Computer Science,* pp. 459-468, 1993.
[13] B. Awerbuch and T. Leighton, "Improved Approximation Algorithms for the Multi-Commodity Flow Problem and Local Competitive Routing in Dynamic Networks," *Proc. 26th Ann. ACM Symp. Theory of Computing,* pp. 487-496, 1994.
[14] 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.
[15] 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 Systems (PDCS '05),* 2005.
[16] D. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, 1998.
[17] V. Vineet and P. Narayanan, "Cuda Cuts: Fast Graph Cuts on the GPU," *Proc. IEEE CS Conf. Computer Vision and Pattern Recognition Workshops (CVPRW '08),* pp. 1-8, 2008.
[18] N. Dixit, R. Keriven, and N. Paragios, "GPU-Cuts and Adaptive Object Extraction," Technical Report 05-07, Laboratoire Centre Enseignement Recherche Traitement Information Systèmes (CERTIS), Mar. 2005.
[19] B. Hong, "A Lock-Free Multi-Threaded Algorithm for the Maximum Flow Problem," *Proc. Workshop Multithreaded Architectures and Applications (MTAAP '08), Held in Conjunction with IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS '08),* 2008.
[20] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms,* second ed. MIT Press, 2001.
[21] J. Cheriyan and K. Mehlhorn, "An Analysis of the Highest-Level Selection Rule in the Preflow-Push Max-Flow Algorithm," *Information Processing Letters,* vol. 69, pp. 69-239, 1998.
[22] D.S. Johnson and C.C. McGeoch, *Network Flows and Matching: First DIMACS Implementation Challenge.* Am. Math. Soc., 1993.

Bo Hong received the bachelor and master's degrees from Tsinghua University, China, in 1997 and 2000, respectively, and the PhD degree in computer engineering from the University of Southern California in 2005. He is currently an assistant professor in the School of Electrical and Computer Engineering at the Georgia Institute of Technology. He worked as an assistant professor in the Electrical and Computer Engineering Department at Drexel University from 2005 to 2008 before he joined the Georgia Institute of Technology. His research interests include high-performance computing, multicore computer architecture, parallel and multithreaded algorithms, and distributed computing. He has published extensively in the above areas. He is a member of the IEEE and the IEEE Computer Society.

Zhengyu He received the BEng and MEng degrees from the Beijing Institute of Technology, and currently he is working toward the PhD degree at the School of Electrical and Computer Engineering, Georgia Institute of Technology. His current research interests include parallel and distributed processing, high-performance computer architectures, and concurrency control mechanisms. He is a student member of the IEEE.