Supplementary File: An Asynchronous Multi-threaded Algorithm for the Maximum Network Flow Problem with Non-blocking Global Relabeling Heuristic

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

Contents of the Supplementary File

Some detailed contents were omitted in the main manuscript due to space limit. These contents are provided in this supplementary file. Section 1 provides the detailed analysis of the interleaved execution of the push and lift operations, which supplements Section 4 of the main manuscript. Section 2 provides details of our asynchronous global relabeling heuristic. Section 3 discusses the details of programming implementation for our algorithm. It covers cache efficiency and the assignment of vertices to threads. Section 4 provides the additional experimental results that were omitted in the main manuscript.

Note that some text and references are copied from the main manuscript to make the supplementary file self-contained.

1 Interleaved Execution of the Push and Lift Operations

Section 4 of the main text proves the correctness and complexity bound of our Algorithm. The proofs are based on the analysis of the interleaved execution of the push and lift operations. Section 4 of the main manuscript only presents an abbreviated analysis due to space limit. The detailed analysis is presented here.

For completeness, the Algorithm and its Program implementation, as well as some explanatory text are copied from the main manuscript.

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

- $Initialize\ h(u)$, $e(u)$, 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 $h(u) > h(v)$.
  $v' \leftarrow \text{argmin}_v[h(v) \mid c_f(u,v) > 0\ and\ h(u) > h(v)]$
  $d \leftarrow \min[e(u), c_f(u,v')]$
  $f(u,v') \leftarrow f(u,v') + d$
  $f(v',u) \leftarrow f(v',u) - d$
  $e(u) \leftarrow e(u) - d$
  $e(v') \leftarrow e(v') + d$
- $Lift(u)$: applies if u is overflowing, and $h(u) \leq h(v)$ for all $(u,v) \in E_f$,
  $h(u) \leftarrow \min\{h(v) | c_f(u,v) > 0\} + 1$

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**

Algorithm 1 can 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.

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 $c_f(u,v)$ [$c_f(u,v) =$

- *B. Hong and Z. He are with the Department of Electrical and Computer Engineering, Georgia Institute of Technology, USA. E-mail: bohong@gatech.edu, zhengyu.he@gatech.edu*

- *Supported by National Science Foundation under award No. CNS-0845583 and in part by an equipment grant from Nvidia Inc.*

$c_{uv} - f(u,v)]$ 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' ← e(u)
4:     h' ← ∞
5:     for all (u, v) ∈ E_f do
6:       h'' ← h(v)
7:       if h'' < h' then
8:         v' ← v
9:         h' ← h''
10:      end if
11:    end for
12:    if h(u) > h' then
13:      d ← min(e', c_f(u, v'))
14:      c_f(u, v') ← c_f(u, v') − d     #executed atomically
15:      c_f(v', u) ← c_f(v', u) + d     #executed atomically
16:      e(u) ← e(u) − d                 #executed atomically
17:      e(v') ← e(v') + d               #executed atomically
18:    else
19:      h(u) ← 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.

Now we examine under what condition special residual edge 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 we discuss below:

Case 1: The execution of $lift(a)$ and $lift(b)$ are interleaved.

Case 1.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 this case. 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). Both (a,b) and (b,a) are still regular residual edges after the two lifts.

Case 1.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. (a,b) remains a regular residual edge after the two lifts.

Case 1.c: Initially, $(b,a) \in E_f$ but $(a,b) \notin E_f$. This is symmetric to Case 1.b. Similarly, we will have $h(b) \leq h(a) + 1$ after the two lift operations, and (b,a) remains a regular residual edge.

Case 1.d: Initially, $(a,b) \notin E_f$ and $(b,a) \notin E_f$. This is a trivial case as the two lift operations do not add (a,b) or (b,a) into E_f.

In summary, interleaved execution of $lift(a)$ and $lift(b)$ does not cause either (a,b) or (b,a) to become special residual edges.

Case 2: The execution of $push(a,b)$ is interleaved with $push(b,c)$ where c is different than a and b. It can be shown easily that this particular trace is always 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 case reduces to the original push-relabel algorithm. Consequently h is trivially maintained as a valid height function and the interleaved execution of $push(a,b)$ and $push(b,c)$ does not cause (a,b) or (b,a) to become special residual edges.

Case 3: The executions of $push(a,b)$ and $lift(b)$ are interleaved. According to Lemma 2 in the main manuscript, this trace is equivalent to either a step-clean or a stage-stepping trace. If it is stage-clean, then the scenario reduces to the original push-relabel algorithm and (a,b) or (b,a) will remain as regular residual edges.

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

Case 3.a: $(b,a) \in E_f$ before the action 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 $(b,a) \in E_f$. Therefore (b,a) remains a regular residual edge.

Case 3.b: $(b,a) \notin E_f$ before the action stage of $push(a,b)$. $push(a,b)$ will add (b,a) into E_f. We have the following two cases to consider:

Case 3.b.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 $(b,a) \notin E_f$. $push(a,b)$ may remove (a,b) from E_f. Note that the removal of (a,b) does not introduce any special residual edges into E_f.

(b,a) will be added into E_f by the action 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 $(b,a) \in E_f$ makes (b,a) a special residual edge in E_f.

When (b,a) becomes a special residual edgel, 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 in E_f means $push(b,a)$ is now applicable. Consequently, $lift(b)$ will become applicable only after (1) $h(a)$ is lifted such that $h(a) > h(b)$, or (2) $push(b,a)$ is applied. In (1), $lift(a)$ restores (b,a) as a regular residual edge. Let us examine if $push(b,a)$ is applied as in (2), how much flow will be sent.

Let d denote the amount of flow $push(a,b)$ sends from a to b, and 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 the special residual edge (b,a) from E_f.

Case 3.b.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, (b,a) thus becomes a special residual edge. Again, similarly to the previous case, a $push(b,a)$ operation becomes immediately applicable when the trace completes. Thus b will not be lifted unless (1) a is lifted such that $h(a) > h(b)$, which restores (b,a) as a regular residual edge, or (2) $lift(b)$ is applied after $push(b,a)$ is applied. For the second case where $push(b,a)$ is applied, 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 $push(b,a)$, which will remove (b,a) from E_f.

Case 4: The executions of $push(a,b)$ are interleaved with $push(c,a)$. This case is symmetric to Case 2, and also reduces to the original push-relabel algorithm. It can be shown easily that interleaved $push(a,b)$ and $push(c,a)$ does not cause (a,b) or (b,a) to become special residual edges.

Case 5: The executions of $push(a,b)$ and $push(c,b)$ are interleaved where c is different than a and b. It is straightforward to show that this particular trace is equivalent to a stage celan trace where $push(a,b)$ is executed in its entirety before (or after) $push(b,c)$ is executed in its entirety. This reduces to the original push-relabel algorithm and does not cause (a,b) or (b,a) to become special residual edges.

Case 6: The executions of $push(a,b)$ and $push(b,a)$ are interleaved. This is impossible because $push(a,b)$ being applicable implies that $h(a) > h(b)$, and thus $push(b,a)$ is inapplicable.

Case 7: The executions of $push(a,b)$ and $lift(a)$ are interleaved. This is impossible because according to Lemma 4 in the main manuscript, $push(a,b)$ and $lift(a)$ cannot be applicable at the same time.

Case 8: The executions of more than two $push$ and $lift$ operations are interleaved. According to Lemma 3 in the main manuscript, the trace will consists of multiple non-overlapping sub-traces. If all the sub-traces are stage-clean, it reduces to the original push-relabel algorithm. If all the sub-traces are stage-stepping with two operations only, the discussion reduces to Cases 1 - 7. If some of the sub-traces are stage-stepping and have more then two operations, then according to Lemma 4 in the main manuscript (that $push(a,b)$ and $lift(a)$ cannot be applicable to vertex a at the same time), we only have the following two scenarios: (1) $push(x_1,y), push(x_2,y), \cdots, push(x_k,y)$, and $lift(y)$, (2) $push(x_1,y), push(x_2,y), \cdots, push(x_k,y)$, and $push(y,z)$. It can be shown easily that for Scenario 1, the discussion of Case 3 applies to the pair-wise relation between the operations ($push(x_i,y)$ and $lift(y)$); and for Scenario 2, the discussion of Case 2 (or 4) applies to the pair-wise relation between the operations ($push(x_i,y)$ and $push(y,z)$).

In summary, the analysis of the 8 cases above 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.

2 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 towards the sink or the source. Global Relabeling heuristic updates

the heights of the vertices with their shortest distance to the sink. This can be performed by a backward breadth-first search (BFS) from the sink or the source in the residual graph [1]. The Gap Relabeling heuristic due to Cherkassky also improves the practical performance of the push-relabel method (though not as effective as Global Relabeling [1]). 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 [2] and Bader [3] 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. The AGR heuristic is listed in Program 3. Due to the presence of the AGR heuristic, the push and lift operations are also updated as shown in Program 4. In Programs 3 and 4, $color[v]$ (for $v \in V$), *CurrentWave*, and *CurrentLevel* are private variables of the AGR thread. The AGR thread also maintains a private queue for the BFS traversal. $h(v)$, $w(v)$, and $c_f(u, v)$ are shared across all the threads.

With the AGR heuristic, the algorithm dedicates one thread to the execution of the heuristic while other 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.

It was pointed out in [1] that the most accurate height to globally relabel vertex v should be $min(h(v,t), h(v,s)+|V|)$, where $h(v,t)$ and $h(v,s)$ denote the shortest distances from v to the sink and the source respectively. In our global relabeling heuristic, a backwards BFS from the sink is first performed (lines 6-20). If the generated BFS tree does not cover all the vertices in the graph (line 21), the remaining vertices must be disconnected from the sink and their $h(v,t)$ are therefore ∞. Another backwards BFS from the source is then performed to scan the remaining vertices (lines 22-38).

The AGR heuristic also assigns a wave number $w(u)$ to each vertex u. $w(u)$ represents the number of times u has been globally relabeled. The updated push operation requires flow to be pushed within the same wave, or from an older wave to a newer wave (line 14).

Because the global relabeling heuristic and the lift operations are executed by separated threads, they may update the height of the same vertex simultaneously (lines 14 and 31 of Program 3 and line 18 of Program 4). To prevent this from happening, we rely on vertex allocation (Section 3.2) to guarantee that a vertex can be either globally relabelled or lifted, but not both. More precisely, our vertex allocation strategy guarantees that lines 13-14 (and 30-31) of Program 3 and lines 23-24 of Program 4 are mutually exclusive. Note that our algorithm allows the push operations to be arbitrarily interleaved with either the AGR heuristic or the lift operations.

Program 3 The Asynchronous Global Relabeling Heuristic

```
1:  CurrentWave ← CurrentWave + 1
2:  for all vertex v ∈ V do
3:     color[v] ← 0
4:  end for
5:  Enqueue the sink
6:  CurrentLevel ← 0
7:  while queue ≠ ∅ do
8:     Dequeue a vertex u
9:     CurrentLevel ← CurrentLevel + 1
10:    for all vertex v|(v,u) ∈ E_f do
11:       if color[v] = 0 then
12:          color[v] ← 1
13:          if h(v) < CurrentLevel then
14:             h(v) ← CurrentLevel
15:          end if
16:          w(v) ← CurrentWave
17:          Enqueue vertex v
18:       end if
19:    end for
20: end while
21: if Not all the vertices are colored then
22:    Enqueue the source
23:    CurrentLevel ← |V|
24:    while queue ≠ ∅ do
25:       Dequeue a vertex u
26:       CurrentLevel ← CurrentLevel + 1
27:       for all vertex v|(v,u) ∈ E_f do
28:          if color[v] = 0 then
29:             color[v] ← 1
30:             if h(v) < CurrentLevel then
31:                h(v) ← CurrentLevel
32:             end if
33:             w(v) ← CurrentWave
34:             Enqueue vertex v
35:          end if
36:       end for
37:    end while
38: end if
```

2.1 Correctness of the AGR heuristic

We prove that the AGR heuristic (with the updated push and lift operations in Program 4) computes a maximum flow for any given graph $G(V, E)$ with $O(|V|^2|E|)$ push and lift operations.

We first examine the interleaved execution of the AGR heuristic and the push operations. Similar to Program 2, each push and lift operation in Program 4 is performed in two stages. For the push operation, lines 2-15 constitute the preparation stage, determining whether a push is applicable, and the amount of flow to be pushed if it is applicable; lines 16-19 constitute the action stage that actually performs the push. For the lift operation, the preparation stage consists of

Program 4 Updated push and lift due to the AGR Heuristic

```
1: while e(s) + e(t) < 0 do
2:   if e(u) > 0 then
3:     e' ← e(u)
4:     h' ← ∞
5:     for all (u, v) ∈ E_f do
6:       h'' ← h(v)
7:       if h'' < h' then
8:         v' ← v
9:         h' ← h''
10:      end if
11:    end for
12:    if h(u) > h' then
13:      d ← min(e', c_f(u, v'))
14:      if w(u) ≤ w(v) then
15:        if h(u) > h(v) then
16:          c_f(u, v') ← c_f(u, v') − d          #atomic
17:          c_f(v', u) ← c_f(v', u) + d          #atomic
18:          e(u) ← e(u) − d                      #atomic
19:          e(v') ← e(v') + d                    #atomic
20:        end if
21:      end if
22:    else
23:      if h(u) < h' + 1 then
24:        h(u) ← h' + 1
25:      end if
26:    end if
27:  end if
28: end while
```

lines 2-12, the action stage consists of lines 23-25. The AGR heuristic also has two stages:

1) the preparation stage: lines 9-11 (26-28) determines whether $h(v)$ needs to be updated and the new value of $h(v)$ if it needs to be updated.
2) the action stage: lines 13-16 (30-33) updates $h(v)$ to the new value, and $w(v)$ to the current wave number.

Because our algorithm does not lock the vertices, the two stages of the AGR heuristic may be arbitrarily interleaved with the push operations. We can extend the definition of traces to include the AGR heuristic, and show that all traces (of interleaved AGR heuristic, push, and lift) are equivalent to either stage-clean or stage-stepping traces. The formal statement and the proof of the lemma are similar to that of Lemmas 2 and 3 in the main manuscript and omitted here.

Similar to Algorithm 1, the interleaved execution of the AGR heuristic and the push operations may lead to special residual edges. Let us examine under what condition such interleaved execution may cause special residual edges, and what will happen thereafter. We use $async\_global\_relabel(u)$ to denote the global relabeling operation for vertex u.

If the execution of $async\_global\_relabel(u)$ is not interleaved with any push operations, the correctness of the AGR heuristic is the same as that in [2] (which locks the vertices to prevent interleaved executions).

Next we consider the following two interleaving scenarios:

- A stage-stepping trace consists of $push(u, v)$ and $async\_global\_relabel(v)$.

  $push(u, v)$ may add (v, u) into E_f. We claim the newly added (v, u) will be a regular residual edge.

  According to line 14 of Program 4, we have $w(u) \leq w(v)$ for $push(u, v)$ to be applicable. Actually, we must have $w(u) = w(v)$ because $w(u) < w(v)$ would imply that v has already been relabeled and thus we should relabel u instead.

  The action of $async\_global\_relabel(v)$ increases $w(v)$ by 1, which results in $w(u) \neq w(v)$ and temporarily removes the constraint that $h(v) \leq h(u) + 1$.

  When u is relabeled later, $w(u)$ will be increased to the same value of $w(v)$, and will thus bring back the constraint that $h(v) \leq h(u) + 1$. This constraint is trivially satisfied: because u is relabeled later, according to the BFS order of the AGR heuristic, we will have $h(u) \geq h(v)$ after u is relabeled, which satisfies $h(v) \leq h(v) + 1$, and (v, u) is thus a regular residual edge.

- A stage-stepping trace consists of $push(u, v)$ and $async\_global\_relabel(u)$.

  $push(u, v)$ may add (v, u) into E_f, which may or may not be a special residual edge. We have two sub-cases:

  Case 1: $w(u) < w(v)$ before the action of $push(u, v)$. This implies that v has already been relabeled and $async\_global\_relabel(u)$ will increase $w(u)$ such that $w(u) = w(v)$, which will bring back the constraint that $h(v) \leq h(u) + 1$. Due to the BFS order that the AGR relabels the vertices, u will not be relabeled lower than v because u is relabeled later than v. We therefore will have $h(u) \geq h(v)$ when $async\_global\_relabel(v)$ completes, which trivially satisfies $h(v) \leq h(u) + 1$, and thus (v, u) is a regular residual edge in this case.

  Case 2: $w(u) = w(v)$ before the action of $push(u, v)$. $w(u) = w(v)$ implies that neither u or v has been relabeled, which means u will be relabeled earlier than v. The action of $async\_global\_relabel(u)$ will lead to $w(u) > w(v)$, and temporarily remove the height constraint between $h(u)$ and $h(v)$. But the constraint will re-appear after $async\_global\_relabel(v)$, as it will increase $w(v)$ such that $w(u)$ is equal to $w(v)$ again. Therefore we need to examine the status of the vertices after $async\_global\_relabel(v)$.

  Case 2.a: $(v, u) \in E_f$ before the action of $push(u, v)$. The existence of (v, u) will cause the AGR heuristic to add v as a neighbor of u and subsequently relabel $h(v)$ to $h(u) + 1$. Therefore we have $h(v) \leq h(u) + 1$ after $async\_global\_relabel(v)$ completes, and (v, u) is a regular residual edge.

  Case 2.b: If $(v, u) \notin E_f$ before the action of $push(u, v)$. In this case, we must also have $f(v, u) = c_{vu}$ before the push. Otherwise $f(v, u) < c_{vu}$ implies $c_f(v, u) = c_{vu} - f(v, u) > 0$ and subsequently $(v, u) \in E_f$, which contradicts the assumption that $(v, u) \notin E_f$.

  Case 2.b.i The action of $push(u, v)$ completes before the preparation of $async\_global\_relabel(v)$ starts. $push(u, v)$ will add (v, u) into E_f, which will cause the AGR heuristic to add v as a neighbor of u and subsequently relabel $h(v)$ to $h(u) + 1$. Therefore we have $h(v) \leq h(u) + 1$ for $(v, u) \in E_f$ after

$async\_global\_relabel(v)$ completes, and (v, u) is a regular residual edge in this case.

Case 2.b.ii The action of $push(u, v)$ completes after the preparation of $async\_global\_relabel(v)$ completes. (v, u) does not exist in E_f during the preparation of $async\_global\_relabel(v)$, therefore v is not considered as a neighbor of u and the relabeling may result in $h(v) > h(u) + 1$, which will make (v, u) a special residual edge once $push(u, v)$ completes (and thus adds (v, u) into E_f).

When (v, u) becomes a special residual edge, at the same time, u becomes the lowest neighbor of v in E_f having the same wave number (other neighbors of v either have been relabeled to $h(v) - 1$, or have not been relabeled yet and thus having smaller wave numbers), thus a new $push(v, u)$ operation is now applicable. Next we examine how much flow $push(v, u)$ will send.

Let d' denote the amount of flow $push(v, u)$ will send from v to u. According to the algorithm, $d' = \min\{c_f(v, u), e(v)\}$. $f(v, u) = c_{vu}$ before $push(u, v)$ implies $c_f(v, u) = d$ thereafter (where d is the amount of flow that $push(u, v)$ sends from u to v). $push(u, v)$ will increase $e(v)$ by d. Note that $e(v) \geq 0$ before $push(u, v)$, we will have $e(v) \geq d$ after the push. Consequently, $d' = \min\{c_f(v, u), e(v)\} = \min\{d, e(b)\} = d$.

$d' = d$ means we will have $f(v, u) = c_{vu}$ upon completion of $push(v, u)$. Thus the special residual edge (v, u) will be removed from E_f.

The above analysis shows that a special sequence of interleaved push and global relabel operations can cause special residual edges. Once (v, u) becomes a special residual edge under this situation, a $push(v, u)$ immediately becomes available, which, upon completion, will remove (v, u) from E_f. Note that The AGR heuristic trivially terminates after it sweeps through all the vertices, by which time all the vertices will have the same wave number.

Because there will not be any overflowing vertices when the algorithm terminates, this implies that all the special residual edges caused by the interleaved push and global relabeling operations will be disappear - otherwise according to the above analysis there will be appliable push operations. We therefore have the following Theorem:

Theorem 1. *With the AGR heuristic and the updated push and lift operations, if Algorithm 1 terminates, then the pre-flow f it computes is a maximum flow.*

The proof is simimar to that of Theorem 1 in the main manuscript and hence omitted here.

Further more, the above analysis leads to the following lemma:

Lemma 1. *Upon completion of the AGR heuristic, 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 Sketch: The above analysis shows that the special residual edges caused by interleaved push and global relabel operations can be removed from the residual graph E_f without changing the height of any vertices. Further more, the remaining residual graph, which is a subset of the E_f that consists of regular residual edges only, must contain a simple path from u to s. The detailed proof of this lemma is similar to that of Lemma 7 of the main manuscript and omitted here. Note that we require the AGR heuristic and the lift operations to be mutually exclusive, so we do not need to consider the situation where push, lift, and global relabeling operations are interleaved. □

Lemma 1 leads to the following lemma:

Lemma 2. *Upon completion of the AGR heuristic, $h(u) \leq 2|V| - 1$ for all vertices $u \in V$.*

The proof of Lemma 2 is similar to that of Lemma 8 of the main manuscript and omitted here. This upper bound on vertex height further bounds the total number of push and lift operations to $O(|V|^2|E|)$, which means that Algorithm 1, when augmented with the AGR heuristic, still has the same complexity bound as the original Algorithm 1. This is stated in the following theorem:

Theorem 2. *With the AGR heuristic and the updated push and lift operations, Algorithm 1 finds a maximum flow with $O(|V|^2|E|)$ push and lift opeations for any given graph $G(V, E)$ with source s and sink t.*

The proof of Theorem 2 is similar to Theorem 2 of the main manuscript: the upper bound on h is used to limit the number of push and lift operations. Details of the proof is omitted here.

3 Programming Implementation of the Algorithm

To validate the efficiency of the proposed asynchronous algorithm, we implemented the algorithm using C and the pthread library. In this section, we discuss the major programming techniques used to accelerate the implementation.

3.1 Cache Efficiency

The memory allocation of the vertices and edges is important for the cache performance and hence the practical execution speed of our implementation. In our implementation, the variables associated with every vertex (height, excess flow, wave number, etc.) are packed in a C struct and allocated continuously in the memory for improved locality. The edge locality, however, is more complicated.

Historically, there have been two edge allocation schemes. In the first scheme, it is observed that during the execution of the algorithm, an edge $(u, v) \in E$ in the original graph may induce two edges (u, v) and (v, u) in the residual graph E_f. Flow may exist along both edges. In particular, The action stage of $push(u, v)$ needs to increase the flow along one edge, and decrease the flow along the other. Since the pair of edges are always updated together, this scheme packs the two edges contiguously in the memory (e.g. into a single C struct). This scheme was proposed in [4] and also used in [3]. The other

scheme sorts the edges leaving the same vertex and allocates these edges contiguously in the memory. In this scheme, (u, v) and (v, u) most likely cannot reside in the memory side by side. Pointers are thus used to expedite accesses to such edges that are updated in pairs. This scheme is used in the code published by [1].

We experimented both schemes in our implementation. The second scheme, which allocates outgoing edges continuously, slightly outperforms the first one. We conjecture the reason is that each push operation in our algorithm needs to search for the lowest neighbor during the preparation stage, while existing algorithms only need to find any neighbor that is lower by 1. The search procedure requires our algorithm to read all the outgoing edges of a vertex. The second edge allocation scheme therefore improves the locality for the preparation stage of pushes at the cost of affecting the action stage. Because the preparation stages in general need to access more edges than the action stage, the second scheme explores the trade-offs better. The scheme also improves the execution of the lift operation: the preparation stage of a lift operation needs to access all the outgoing edges of a vertex and will benefit from the improved data locality. Based on these reasons, we adopt the second scheme of edge layout in our implementation.

3.2 Load Balancing

Our algorithm can be implemented with an arbitrary number of threads. To achieve meaningful acceleration on a multicore or a multi-processor system, we need to limit the number of threads to the number of hardware contexts that the system supports. Under this practical limit, each thread needs to be in charge of multiple vertices. Load balancing is therefore crucial to the efficiency of our algorithm. The key to load balancing is the allocation of overflowing vertices to keep all the threads busy.

Previous programming implementations use a global queue to maintain the overflowing vertices. When a thread finishes processing the overflowing vertices that were obtained from the global queue, all the newly generated overflowing vertices (by push operations) will be added to the global queue, from which other threads can request a subset of such vertices to work on. The process continues until none of the vertex overflows. Because the global queue is accessed by all the threads and thus needs to be protected (locked) for concurrent accesses, it is likely to become a performance bottleneck especially as the number of threads increases.

We hereby introduce our strategy of using distributed queues to balance the load. Every thread maintains a local queue for overflowing vertices. Since these are local queues, no lock is required to protect their accesses. Every thread processes its local vertex queue in FIFO order. In our scheme, each thread processes the vertices in the local queue and insert all the newly generated overflowing vertices back to the local queue. Communication between threads only occurs when a thread empties its local queue (and hence needs to request overflowing vertices from other threads).

When a thread T1 empties its local queue, it will search and find another thread T2 that has extra overflowing vertices. T1 will send a request by atomically setting a flag $Exchange$ of T2 if it is not set. After successfully setting T2's $Exchange$, T1 will lock the its own $Exchange$. With the two flags, other threads will not attempt to send any requests to T1 or T2 until the vertex exchange between T1 and T2 completes. This $Exchange$ flag is similar to a lock but it contains the sender thread's ID. T2 will notice (after finishing the current push or lift operation) that T1 has requested overflowing vertices. T2 will split its local queue and give a certain number of vertices to T1. T2 will then reset the $Exchange$ flags for T1 and itself, thereby completing the vertex exchange.

The above load balance scheme has the advantage of allowing a thread to locally execute as much operations as possible before exchanging vertices with other threads. In addition, each exchange involves only two threads so that other threads are not affected during the process. Additionally, the elimination of the global queue also reduces the memory footprint of the program.

To prevent a vertex from being lifted and globally relabeled by two threads simultaneously, we need to protect the update of the vertex height (lines 13-15/30-32 of Program 3 and lines 23-25 of Program 4). In our implementation, we do not need to stall any threads in case of such a conflict. For the lift operations, we let them skip the vertex that is currently being relabeled and process the remaining vertices in the local queue. For the global relabel operations, we let them skip the vertex that is currently being lifted and process the following vertices at the same BFS level. The skipped vertex will be reexamined later: (1) our algorithm iterates through the vertices until it terminates so a skipped push operation will be reexamined, and (2) for the global relabeling operation, because it is BFS, it needs to complete vertices at one level before moving to the next level. So if a vertex is skipped, it will be revisited before the heuristics moves to the next level of vertices. Our implementation is therefore non-blocking. We use the atomic compare-and-swap instruction to determine whether we should continue with the relabel/lift operation or skip to the next vertex.

4 Additional Experimental Results

This section presents the additional experimental results that were omitted in the main manuscript due to space limit.

4.1 Impact of Global Relabeling

We observed that when the AGR heuristic is applied, the execution time is greatly reduced for all the input graphs, by as much as 500 times. The frequency of applying the AGR heuristic also affects the execution time. The impact on the Genrmf-long ($l_1 = 256, l_2 = 32$) graphs is shown in Figure 1 (we observed the same trend on other graphs and the results are omitted here). Figure 1 shows that, when the AGR heuristic is applied more frequently (from every $4|V|$ push/lift operations to $2|V|$, $|V|$, and $|V|/2$), the execution time reduces, though the improvement is marginal beyond $2|V|$. In the experiments, we also observed (not shown in Figure 1) increases in the execution time when the frequency of global relabeling was further increased. In summary, the experiments suggested that

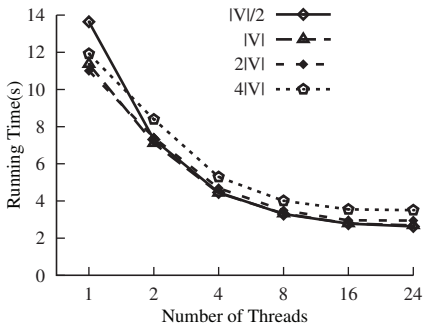


Fig. 1. The impact of the frequency of global relabelling

applying the AGR heuristic after every $|V|$ push/lift operations is a reasonable choice. In the rest of the experiments, we fixed the frequency to $|V|$.

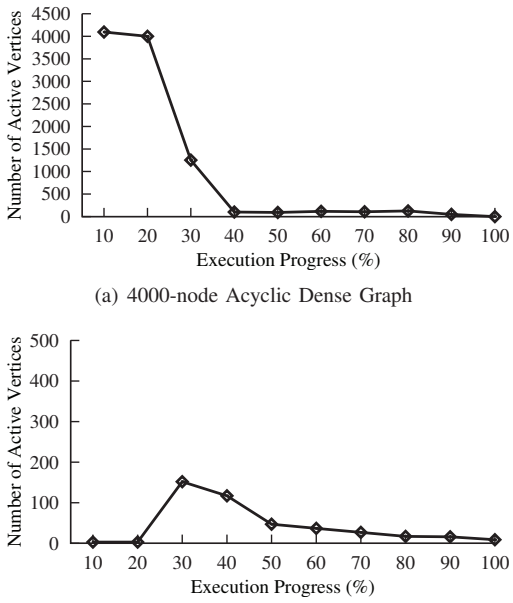
4.2 Comparison Against Existing Algorithms

Figure 2 shows the experimental results of Acyclic-dense graphs on the AMD platform. Figure 3 shows the experimental results of Genrmf-long graphs on the AMD platform. Figure 4 shows the experimental results on Genrmf-wide graphs. Figure 5 shows the experimental results of Washington-RLG-long graphs on the AMD platform. Figure 6 shows the experimental results of the Washington-RLG-wide graphs. These figures were omitted in the main manuscript due to space limit.

4.3 Detailed Comparison Against hi_pr

Comparing with **hi_pr**, our algorithm demonstrated different characteristics for dense (Figure 2) and sparse (Figures 3, 4, 5, 6) graphs. For dense graphs, **amf** (with multiple threads) outperforms **hi_pr** significantly despite the higher complexity bound in terms of total number of operations. For sparse graphs, **amf** is slower in absolute speeds until 8 - 16 threads were used (though good scalability were observed).

There can be multiple factors affecting the execution speed of actual implementations of algorithms. To name a few examples, multicore platforms are subjected to the impact of cache coherency protocols where shared accesses cause expensive overheads and therefore affecting the achievable scalability, the cache performance is further affected by the choice of data structures (our **amf** implementations used arrays to store vertex information which tends to be more cache friendly than the linked list structures used by **hi_pr**). However, we conjecture the primary reason for this seemingly 'inconsistent' behavior is due to the input graphs. In particular, parallelization efficiency of **afm** is affected by the number of vertices that overflow at the same time (which sets an upper bound on the number of parallel threads that can do useful work simultaneously). This is demonstrated in Figure 7. For the dense acyclic dense graph, the number of concurrently overflowing vertices is orders of magnitudes higher than what was available with the sparse Genrmf-long graph. This number directly translates to the number of available push and lift operations that can keep the threads busy. Consequently, our **amf** algorithm was able to utilize the threads more efficiently for dense graphs. To



(a) 4000-node Acyclic Dense Graph

(b) 4000-node Genrmf-long Graph

Fig. 7. Availability of overflowing vertices during the course of execution, dense v.s. sparse graphs.

take advantage of the fact that **amf** and **hi_pr** are suitable for different input graphs, it would be desirable to develop an adaptive algorithm that dynamically switches between **amf** and **hi_pr** for different input. This would require priori knowledge of the input graphs and is beyond the scope of this paper. We will leave this topic for future research.

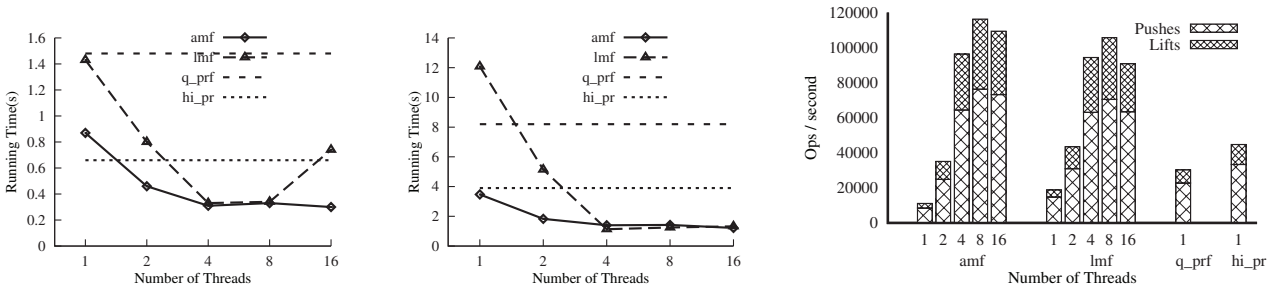
REFERENCES

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

[2] 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*. New York, NY, USA: ACM, 1992, pp. 168–177.

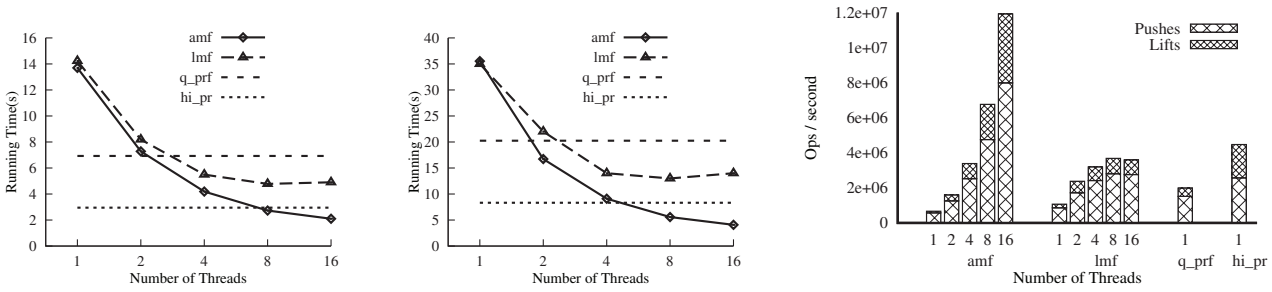
[3] D. Bader and V. Sachdeva, "A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic," in *PDCS '05: Proceedings of the 18th ISCA International Conference on Parallel and Distributed Computing Systems*, 2005.

[4] D. D. K. Sleator, "An o(nm log n) algorithm for maximum network flow," Ph.D. dissertation, Stanford, CA, USA, 1981.



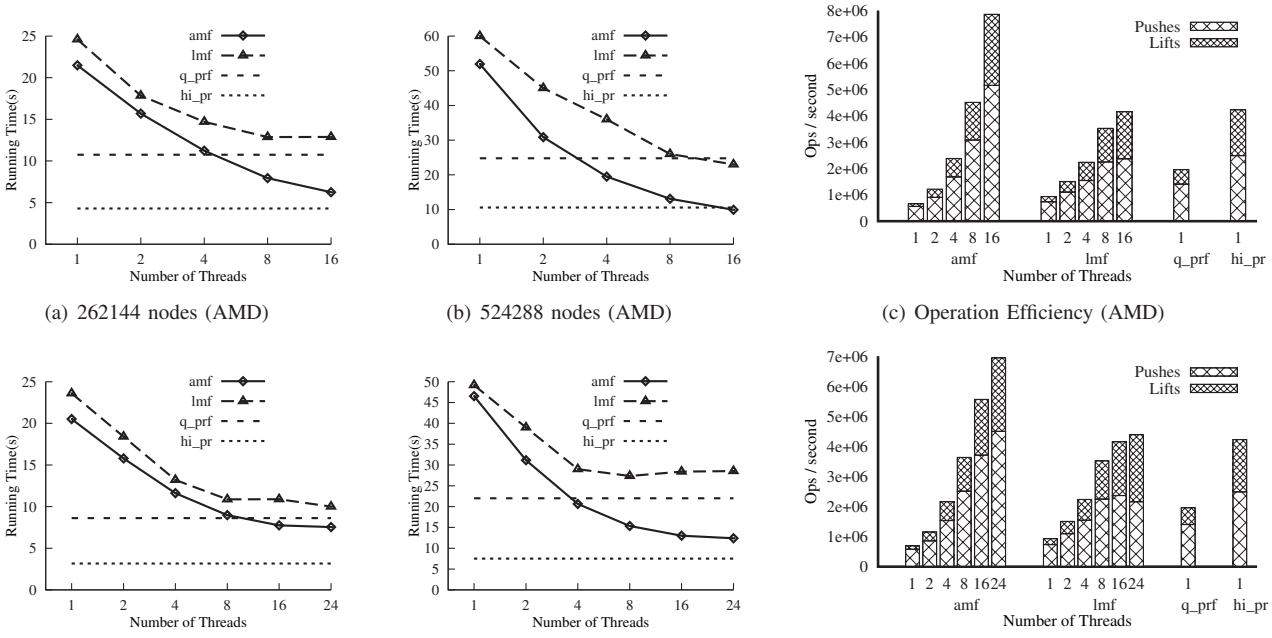
(a) 2000 nodes (AMD) (b) 4000 nodes (AMD) (c) Operation Efficiency (AMD)

Fig. 2. Experimental results of Acyclic-Dense graphs on the AMD platform.



(a) 262144 nodes (AMD) (b) 524288 nodes (AMD) (c) Operation Efficiency (AMD)

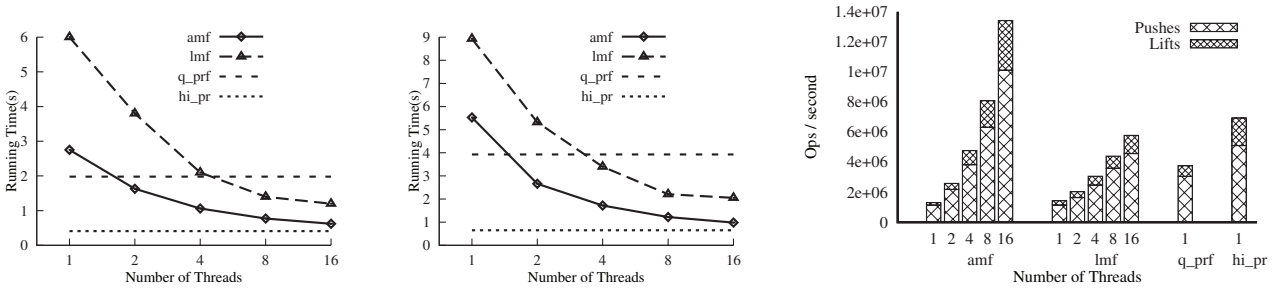
Fig. 3. Experimental results of Genrmf-long graphs on the AMD platform.



(a) 262144 nodes (AMD) (b) 524288 nodes (AMD) (c) Operation Efficiency (AMD)

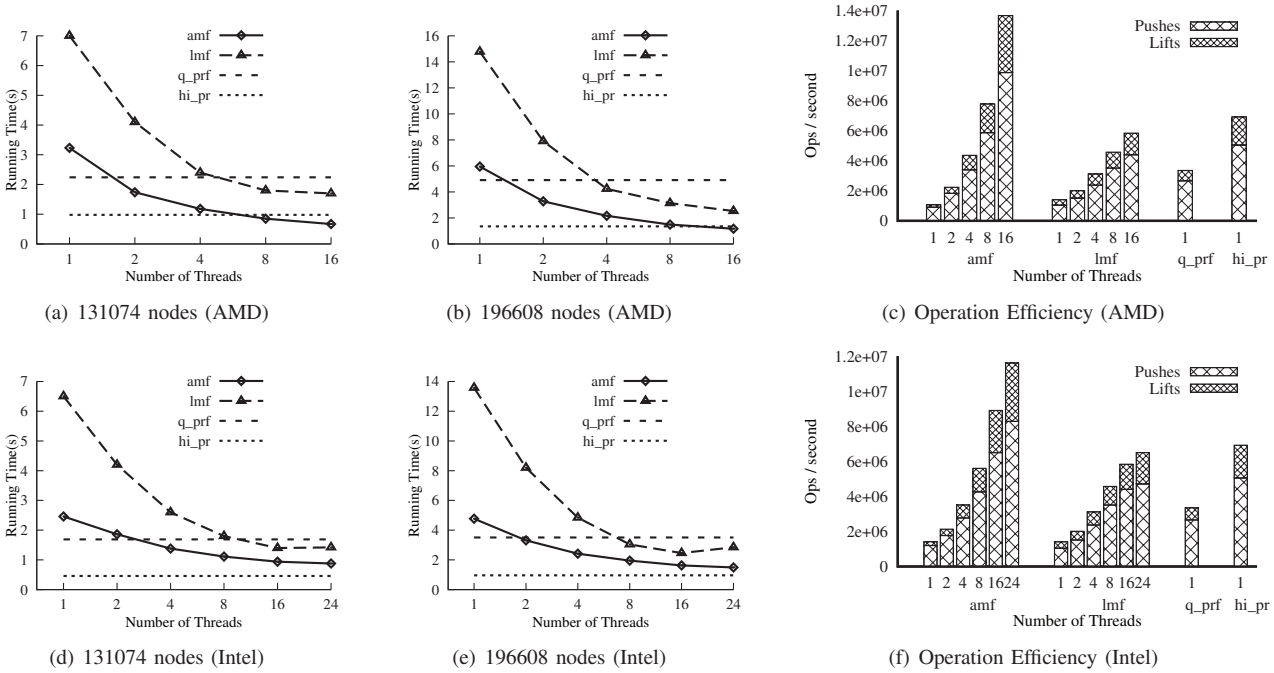
(d) 262144 nodes (Intel) (e) 524288 nodes (Intel) (f) Operation Efficiency (Intel)

Fig. 4. Experimental results of Genrmf-wide graphs.



(a) 131074 nodes (AMD) (b) 196608 nodes (AMD) (c) Operation Efficiency (AMD)

Fig. 5. Experimental results of Washington-RLG-long graphs on the AMD platform.



(a) 131074 nodes (AMD) (b) 196608 nodes (AMD) (c) Operation Efficiency (AMD)

(d) 131074 nodes (Intel) (e) 196608 nodes (Intel) (f) Operation Efficiency (Intel)

Fig. 6. Experimental results of Washington-RLG-wide graphs.