# Parallizing the Push-Relabel Max Flow Algorithm

Victoria Popic
viq@mit.edu

Javier Vélez
velezj@mit.edu

## ABSTRACT

We parallelized Goldberg's push-relabel max flow algorithm and its global update heuristic. We achieve a speedup of 2 for the global update heuristic on wide *rmf* graphs when run on 8 processors. We also implemented a version of the global update heuristic that can run concurrently with the parallel push-relabel code. The best parallel push-relabel algorithm showed a speedup of 2 on longer *rmf* graphs and outperformed Goldberg's *hipr* code when run on 8 processors.

## 1. INTRODUCTION

In this paper we focus on the parallelization of the maximum flow problem, which is a classical combinatorial problem with many important applications (e.g network resource-allocation and scheduling). The sequential algorithms for this problem are usually divided into two groups: augmenting path algorithms and preflow push-relabel algorithms. In this paper, we focus on Goldberg's push-relabel algorithm since it has been shown to be the fastest sequential maximum flow algorithm in practice [1]. It has also been shown that the algorithm performs best when run with a global relabeling heuristic, which is essentially a breadth first search. Therefore, we also focus on parallelizing this heuristic operation.

The paper is organized as follows. In Section 2 we describe Goldberg's push-relabel algorithm and the global relabeling heuristic used to improve its performance. In Section 3 we present the parallelization scheme for the global relabeling heuristic. Section 4 describes a scheme to run the global relabeling concurrently with the push-relabel algorithm. Section 5 introduces several approaches taken to parallelize the push-relabel algorithm itself. In Section 6 we also describe a parallel lock-free algorithm derived from a variation of the push-relabel algorithm by Hong [2]. The lockfree algorithm is of theoretical importance since it only becomes pratical when running on $O(V)$ processors. Finally Section 7 presents the results of our experimental studies.

## 2. PUSH-RELABEL ALGORITHM

We start by defining the maximum flow problem. We are given a flow network, which is a directed graph $G(V,E)$ with $|V| = n$ nodes and $|E| = m$ edges and two special nodes: the source $s$ and the sink $t$. There is a positive capacity value $c(v,w)$ associated with each edge $(v,w)$. The flow on $G$, is a real-valued function $f$ satisfying three constraints: (1) capacity constraint - the flow along each edge does not exceed the capacity of the edge: $f(v,w) <= c(v,w)$, (2) skew symmetry: $f(v,w) = -f(v,w)$, and (3) flow conservation at each node except the source and the sink: sum(f(v,w)) = 0. The value of the flow, $|f|$, is the net flow leaving the source or entering the sink. We want to maximize $|f|$.

The `push-relabel` max flow algorithm has proven to be a very fast and efficient way of computing the maximum flow of a graph [3] [1]. The basic idea is to push as much flow as possible from the source in one step. The algorithm relaxes the flow conservation constraint through the idea of $preflow$, which allows a node $v$ to store $excess\ flow$, $excess(v)$. Nodes with positive $excess\ flow$ are called *active* nodes. Edges along which we can push flow are called *admissible*. The algorithm also associates a *label* with each node $v$, $d(v)$, which represents a lower bound on the distance of this node from the sink. The algorithm runs on the residual network, $G_f$, which consists of all the nodes and only the admissible edges and stops when there are no more active nodes.

The basic algorithm consists of two simple operations: `push` and `relabel`. Listing 1 and 2 show the pseudocode for each of these operations. As the algorithm runs, the active nodes push flow towards nodes one level below them. When a node is unable to push any more flow but has a positive excess, it performs a relabel operation. The discharge operation in listing 3 incorporates this functionality. Two commonly used heuristics for the `push-relabel` are the `global update` and `gap` heuristics. Both of these heuristics have been shown to greatly increase running time performance [1]. In this paper we focus on the global relabeling heuristic, which resets the distance estimates of each node from the sink to their true value.

The fastest serial `push-relabel` in practice seems to be Goldberg's `hipr` code [1], which uses both `global update` and `gap` heuristics. The `hipr` algorithm works as described above. It keeps a lower-bound estimate of the distance label of each node and pushes flow from nodes with excess to lower-distanced nodes through admissible arcs. It stores all the nodes with the same distance label in a linked-list struc-

```
PUSH$(u,v)$
    // Applicability: $u$ is active, $c_f(u,v) > 0$ and $d(u) < d(v)$
    $\delta = min(e(u), c_f(u,v))$
    $f(u,v) \leftarrow f(u,v) + \delta$
    $f(v,u) \leftarrow f(v,u) - \delta$
    $excess(u) = excess(u) - \delta$
    $excess(v) = excess(v) + \delta$
```

**Figure 1:** PUSH operation

```
RELABEL$(u)$
    // Applicability: $u$ is active,
    // $\forall w \in V, if c_f(u,w) > 0 \Rightarrow d(u) \leq d(w)$
    $d(u) \leftarrow min(d(w) | (u,w) \in E_f) + 1$
```

**Figure 2:** RELABEL operation

```
DISCHARGE$(u)$
    // Applicability: $u$ is active
    while $excess(u) \neq 0$
        PUSH or RELABEL $(u)$
```

**Figure 3:** DISCHARGE operation

ture called *bucket*. There are $n$ buckets in total for each possible distance from the sync. Periodically, the `global update` is run to reset the distance estimates to their real values. It has been shown [1] that the performance improves significantly if the nodes with higher distances (further from the sink) are discharged first (versus a FIFO order). Therefore, the algorithm always pushes from the farthest nodes towards the sink by keeping track of the currently highest distance label of an active node, known as *aMax*. See [1] for further details and a proof of why the algorithm converges to the correct maximum flow. The basic `push-relabel` has a running time of $O(V^2E)$ [1]. This can be improved to the `hipr` running time of $O(V^2\sqrt{E})$ by following the highest-distance node discharge ordering [1], and to $O(VE\log(\frac{V^2}{E}))$ by using dynamic tree structures [1].

It has been shown by profiling the sequential execution [4] that the `global update`can take as much as 40% of the total work. Figure 4 shows the amount of time the `hipr` algorithm spends on the heuristic on several graph sizes; we can see that at least $\frac{1}{3}$ of the time is spent running the `global update`. This suggest that increasing the performance of the `global update` step can significantly improve the performance of the overall `push-relabel` algorithm.

## 3. PARALLEL GLOBAL-UPDATE

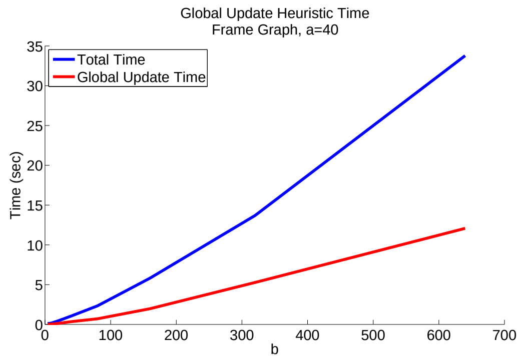

**Figure 4:** Global Heuristics Times Versus Total Running Time for `hipr`

The `global update` is in escense a backwards breadth first search over the current residual graph, $G_f$. We can parallelize this breadth first search by using the `Bag` and the `Pennant` data structures introduced by Leiserson and Shardl, which ensure a fast parallel iteration over the set of stored elements [5]. A `Bag` is a dynamic unordered set of elements implemented using the auxiliary `Pennant` data structure, which is a tree of $2^k$ nodes, where $k$ is a nonnegative integer. The `Pennant` structure allows us to efficiently union and split the elements of a `Bag`.

We implemented our parallel `global update` for *hipr* by keeping track of two `Bag`s for layers $i$ and $i+1$ of the breadth first search. We process all the nodes in layer $i$ in parallel by scanning all their neighbors in the residual graph - all nodes in layer $i+1$ which have admissible edges to the nodes in layer $i$ - and store these neighbors in the layer $i+1$ `Bag`, setting their distance from the sink. Once we are done with layer $i$ `Bag`, we swap the two `Bag`s and continue until we reach the source. We start with the `sink` as the only element in the layer 0 `Bag`. We use a `cilk reducer` on the $i+1$ `Bag` to correctly serialize the insertions of elements into the `Bag`. While a standard breadth first search can be done in parallel this way, the `global update` heuristic actually edits the value of the node in such a way that we no longer have a benign race when visiting nodes ( we do not want duplicate nodes in our `Bag` ). We solve this issue by acquiring a fine-grain lock on a per-node basis to ensure that we only visit each node once. In order for the *buckets* of *hipr* to contain the linked list of appropriate nodes at the end of the `global update`, we maintain the node pointers when adding elements to a `Bag` and merging `Bag`s.

## 4. CONCURRENT GLOBAL-UPDATE

The `global update` needs to run periodically during the push-relabel algorithm. If the push-relabel algorithm is parallelized, then all the processors would need to be suspended in order to run the `global update`. Anderson [4] has presented a correct method for running the `global update` *concurrently* with a parallel implementation of the push-relabel algorithm. This method ensures that the valid labeling condition of Goldberg's algorithm is met. The valid labeling condition states that given any vertex $v$, it must be the case that $d(v) \leq d(w)+1$ for all the edges $(v,w)$ in the residual graph $G_f$ and a violation of this condition might result in incorrect results. They associate a *wave* number with each node, which stores the number of times the node was globally relabeled. It is only allowed to push flow between nodes with the same wave number and the two nodes must be both locked when the push occurs. It is also important to make sure that in any distance label update the node's distance label is *never decreased* . The relabel operation needs to lock the node being relabeled, similarly the `global update` also needs to lock each node it reaches. There are two global variables to store the current `global update` wave number and the current level in the breadth first search tree. Further information and correctness proofs can be found in [4].

We have implemented Anderson's concurrent `global update`. This heuristic runs periodically after at least $n$ discharges occur.

## 5. PARALLEL PUSH-RELABEL

The idea behind parallelizing push-relabel is to find a way to discharge active nodes in parallel. Discharging nodes in parallel (as well as running the concurrent `global update`) introduces many races. In order to avoid them, locks need to be used in several sections of the code. For example, when flow is pushed between two nodes, both nodes are locked in order to update the flow excess and residual capacity values and prevent a concurrent relabeling of one of the nodes. The node being relabeled needs to be locked throughout the entire operation in order to update its distance label and prevent flow from being pushed to it creating an admissible arc at a lower level. Adding newly activated nodes to the *buckets*, also requires a locking mechanism or an atomic operation. Finally, we also need to ensure that the updates to the global variables (such as *aMax*) are performed correctly (a lock is currently used in all the implementations to achieve this).

The following subsections describe several different algorithms used to parallelize push-relabel.

### 5.1 Discharge-Chain

The sequential code walks through the buckets from highest to lowest level, discharging the active nodes. We would like these discharges to run in parallel. In order to avoid excessive Cilk overhead, we would like to provide sufficient work for each processor; however, one discharge operation is a very small amount of work. The amount of nodes in each separate bucket can also be very small; therefore, traversing each bucket in parallel and then synching is also not a good option.

One way of providing more work to each processor is to spawn a discharge and let the processor proceed discharging its newly activated node with the highest distance label, if one exists. We call this modified discharge operation *discharge − chain*. The processor should only proceed with discharging this node, if the distance label of this node is higher than the current *aMax* value (i.e. the current highest distance of an active node), in order to guarantee an approximate highest-level first traversal. Listing 5 shows the discharge loop.

```
MAIN$_{discharge-chain}$()
    while $ActiveNodeSet \neq \emptyset$
        $u \leftarrow max_{d(v)}\{v | v \in ActiveNodeSet\}$
        cilk_spawn DISCHARGE-CHAIN($u$)
```

**Figure 5:** Main Loop for DISCHARGE-CHAIN Algorithm

```
MAIN$_{coarsened-discharge}$()
    while $ActiveNodeSet \neq \emptyset$
        // Grab the top $T$ active nodes
        $a \leftarrow buckets[top\ T\ elements]$
        cilk_for $u \in a$
            DISCHARGE($u$)
```

**Figure 6:** Main Loop for COARSENED-DISCHARGE Algorithm

Each bucket is a linked list of nodes with pointers to the first and last nodes. In order to speed up the transfer of nodes activated by a worker into the buckets, each processor keeps a *local* linked list of newly activated nodes and appends this list, when it's done, to the appropriate bucket by a simple node pointer manipulation. Since a discharge for a node $v$ can only activate nodes one distance label below $v$ $(d(v)-1)$, we are guaranteed to accumulate nodes in the local list for only one bucket (as long as this list is transfered back before any relabeling).

Two versions of this algorithm have been implemented. One that syncs the processors periodically to run the global relabel update and a second version which runs the global update concurrently. Running the concurrent global update seems to improve performance significantly.

### 5.2 Coarsened-Discharge

The second parallelization approach first gathers a batch of active nodes to discharge into an array starting from the highest-level bucket and then runs a cilk_for loop over these nodes (see Listing 6). The number of nodes to gather, $T$, can be varied to improve performance.

This algorithm does not need to explicitly sync the nodes for the global update heuristic, which is run after the *cilk_for* loop, if appropriate.

### 5.3 Local Queues

What we want is to provide a batch of active nodes to each processor instead of just running one discharge. Anderson and Setubal [4] and later Bader [6] have proposed parallelizing push-relabel using the idea of a threadpool: each processor sends tasks to and receive tasks from a global workpile. The size of the tasks assigned to the processors depends on the work available and needs to be adjusted dynamically. We have implemented this algorithm.

In this scheme, each processor has two *local* queues: an $in-queue$ and an $out-queue$. A processor discharges active nodes from its in-queue and places any newly activated nodes in its out-queue. When a processor runs out of work, it grabs a new batch of nodes from the global queue and stores it in the in-queue. When the out-queue reaches a certain capacity, all its nodes are transfered to the global queue. For load-balancing, the number of vertices transferred in and out needs to vary throughout the execution of the program depending on how many processors are $idle$. A processor is considered idle if its in-queue is empty and the global queue and its out-queue are empty.

Since the processing of vertices must occur highest-label first rather than in FIFO order, similarly to Bader[2], we divided the global queue and the local in-queue into *buckets* corresponding to the distance to the sink. When transferring nodes from the global queue to the local queue, we transfer the nodes from the highest buckets first. Similarly, the nodes in the highest buckets of the in-queue are discharged first. This ensures that the nodes are processed approximately in the highest-label first order.

The current implementation of this algorithm does not run the global heuristic concurrently; therefore, each thread needs to be interrupted and synched with the global queue in order to run the global heuristic. However, a parallel of the global heuristic can be run.

## 6. PARALLEL LOCKFREE

In contrast to the `hipr` algorithm, we also implemented the lockfree `push-lift` algorithm in [2]. In the original publishing of the algorithm it was stated that locks were still needed in order to verify the termination condition. We improved the algorithm by creating a completely lockfree parallel maximum flow algorithm using the `push` and `lift` operation defined in [2]. Termination is verified by a careful bookkeeping of when a node losses all excess or gains excess after being excess free. This bookkeeping is done using `atomic fetch-and-add` operations in order to check when no nodes have any excess flow – in which case the algorithm has terminated with the maximum flow stored as excess in the `sink`. We keep a current correct count of the number of nodes with excess at any point in time. Each concurrent process can check this value and decide to stop when it reaches 0.

It is worth noting that the lockfree `push-lift` algorithm did not utilize any heuristics when deciding which nodes to `push`. In fact, all `push` and `lift` operations per node can be done in parallel. The serial running time of the `push-lift` is $O(V^2E)$ which is slower than `hipr` by a factor of $O(\sqrt{E})$. In section 6.1 we detail how we improved the orders of `push` and `lift`

```
PR-NODE-LOOP(u)
    // This is run in parallel for each node $u \notin \{sink, source\}$
    while $excess(u) > 0$
        for $v \in \{(u,v) \in G \ \& \ c_f(u,v) > 0\}$
            if $d(v) = d(u) + 1$
                $\text{PUSH}_{lockfree}(u,v)$

        if $excess(u) > 0$
            $lifted = \text{UPLIFT}(u)$
            if $lifted = \text{FALSE}$
                return
        else
            // $u$ no longer active
            return
```

**Figure 7:** Node Loop For Lockfree `push-relabel` Algorithm

```
$\text{MAIN}_{pr-lockfree}()$
    while $activenodecount > 0$
        cilk_for $i \in V$
            $\text{PR-NODE-LOOP}_{lockfree}(v_i)$
    $flow = excess(sink)$
```

**Figure 8:** Lockfree `push-relabel` Algorithm

operations in order to reduce the number of nodes idling as well as try to always `push` flow from nodes farthest from `sink` towards the `sink`.

Starting with the `push-lift` algorithm, we modified the operations to mimic those of the `hipr` algorithm when run without the `global update` and `gap` heuristics. Listing 8 shows the code for our lockfree `push-relabel` algorithm. This algorithm can be run entirely in parallel, giving us a $O(V)$ potential parallelism when run on $O(V)$.

### 6.1 Order Heuristics

Given that the `global update` and `gap` are an important part of the performance of the `push-relabel` max flow algorithms, we investigated how to order `push` and `relabel` operations in our lockfree `push-relabel` algorithm. We created a set of thread local `strata` for each of our concurrent threads. A `strata` stores a partially ordered set of nodes within it. The `strata` allows us to determine which nodes have distances from the sink in the top half of all the distances in the local `strata`. Figure 11 represents the structure of our `strata`.

Figure 12 shows the main algorithm loop which allows us to efficiently iterate over all the nodes at the top of all local `strata`. We create a **noderange** set which maps the top halves of all `strata` into a compact range of integers by simply ordering our `strata` and enumerating the location of nodes which are in *top* or within the top *num-active-layers* layers. This **noderange** represents the top distnaced active nodes com-

```
PUSH$_{lockfree}(u,v)$
    // Applicability: $excess(u) > 0$
    $\delta = min(excess(u), c_f(u,v))$
    FETCH-AND-SUBTRACT$_{atomic}(f(u,v) - \delta)$
    FETCH-AND-ADD$_{atomic}(f(v,u) + \delta)$
    $e_u$ = SUBTRACT-AND-FETCH$_{atomic}(excess(u) - \delta)$
    $e_{v_{old}}$ = FETCH-AND-ADD$_{atomic}(excess(v) + \delta)$
    if $e_{v_{old}} = 0 \ \& \ \delta > 0 \ \& \ u \notin \{source, sink\}$
        // $v$ gained positive exces sand became active
        FETCH-AND-ADD$_{atomic}(active\text{-}node\text{-}count + 1)$
        LOCAL-ADD-TO-STRATA-OUTSET( v )

    if $e_u = 0 \ \& \ \delta > 0$
        // $u$ just became inactive
        FETCH-AND-SUBTRACT$_{atomic}(active\text{-}node\text{-}count - 1)$
```

**Figure 9:** Lockfree `push` Operation

```
UPLIFT$_{lockfree}(u)$
    // Applicability: $excess(u) > 0 \ \& \ \forall(u,v)|c_f(u,v) > 0,$
    // $d(u) \geq d(v)$
    // First, find min distance of admissible arc neighbors
    $h \leftarrow min\{d(v)|(u,v) \in G \ \& \ c_f(u,v) > 0\}$
    if $\{v|(u,v) \in G \ \& \ c_f(u,v) > 0\} = \emptyset$
        return$false$
    else
        $d(u) \leftarrow h + 1$
        return$true$
```

**Figure 10:** Lockfree "`relabel`" Operation

pactly, and we can now iterate over these nodes in parallel using a `cilk_for` statement.

It is important to notice that our algorithm now iterates in phases, where each phase the top half of active nodes are discharged ( repeated calls of `push` ) and any new active nodes are added to the *local* `strata` *outset*. After a phase is complete, the layers of all the `strata` are cleared ( not including the *outset* ) and then re-filled from elements within the *outset* of each `strata`.

## 7. RESULTS

We tested our algorithm using the following two types of graphs.

• *Rmf graphs* : these graphs are parametrized by two number $(a, b)$. The graphs consists of $b$ square grids (frames) of $a^2$ nodes each connected to each other in a sequence, such that, each node in the frame is connected to one unique node in the next layer. The source is in a corner of the first frame and the sink is in a corner of the last frame. These graphs were used

```
STRATA:
    // number of layers in array
    long num-layers
    // the distance of lowest layer no including bottom
    long lowest-layer
    // max distance of any node in outset
    long max-distance
    // min distance of any node in outset
    long min-distance
    std::vector<node*> top
    std::vector<node*>* layers
    std::vector<node*> bottom
    // num elements in layers not including outset
    long num-elements
    // nodes which are locally in this strata
    // but are not in layers yet
    std::vector<node*> outset
    // num layers which will be operated in parallel
    // not including top
    long num-active-layers
    // bookeeping variable with the last
    // noderange mapping to this strata
    long node-range-last
```

**Figure 11:** The `strata` data structure

```
MAIN$_{strata}()$
    node-range ← COMPUTE-NODE-RANGE-FROM-STRATA()
    while $activenodecount > 0$
        cilk_for $nr\text{-}id \in node\text{-}range$
            node $u \leftarrow$ GET-NODE-FROM-NODE-RANGE($nr$)
            PR-NODE-LOOP($u$)

        CLEAR-ALL-STRATA-LAYERS()
        FILL-INDIVIDUAL-STRATA-LAYER-FROM-OUTSET()
        node-range ← COMPUTE-NODE-RANGE-FROM-STRATA()

    $flow \leftarrow excess(sink)$
```

**Figure 12:** The main function to utilize the `strata` structure

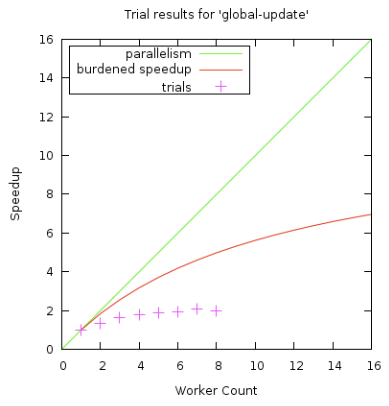

**Figure 13:** Parallel global-update heuristic on $rmf(a = 100, b = 100)$

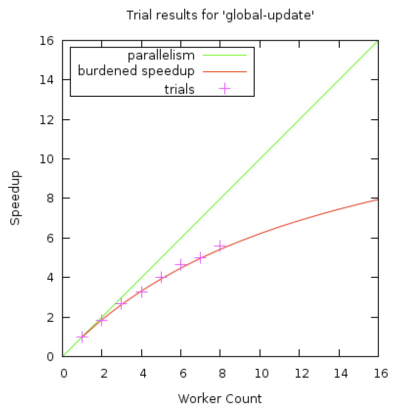

**Figure 14:** Parallel global-update heuristic on $rmf(a = 100, b = 100)$ with extra work

in the original `hipr` algorithm and are described in [**?**]. We differentiate between wide rmf graphs, $rmfw$, and long rmf graphs, $rmfl$.

• *Tree graphs* : these graphs are parametrized by three numbers $(L, d, m)$. They consist of a $d$-degree tree of $L$ layers. There are $m$ middle line graphs connected from the last layer of the tree. Lastly, the line graphs are "collected" back into a single node ( symetric to the tree ). The source is at one end of the tree and the `sink` at the other.

## 7.1 Global Relabeling Heuristic

We achieve a speedup of 2 for the parallel global update heuristic on wider $rmf$ graphs when run on 8 processors. However, Cilkview reports a higher parallelism on these graphs. For example, when run on an rmf graph with 1,000,000 nodes and 4,950,000 edges (a = 100, b = 100), the reported parallelism is 36.34 and the burdened parallelism is 14.14, while the speedup is only 2 on 8 processors. Figure 13 generated by Cilkview shows the performance of the parallel global-update on this graph. Given the burdened parallelism,

| copy # | update running time (s) |
|---|---|
| 1 | 16.351 |
| 2 | 13.556 |
| 3 | 18.031 |
| 4 | 18.963 |
| 5 | 18.166 |
| 6 | 17.479 |
| 7 | 17.607 |
| 8 | 11.704 |

**Table 1:** Sequential global-update run on 8 independent copies

we should expect a speedup of 4 on 8 processors. In order to determine whether the lower speedup could be caused by a memory bandwidth problem, we ran the following two tests. The first test consisted of running 8 independent copies of the sequential code at the same time. When only one version of the sequential code is running, the parallel global-update takes a total of 7.848 s. Table 1 shows the amount of time that each of the 8 independent copies took when ran at the same time. Comparing to the time taken by only one copy of the code running alone, we see that the time roughly doubles when the copies are run simultaneously. This factor 2 slowdown might account for the factor of 2 decrease in speedup. To confirm that, the second test was to insert extra trivial work (with no effect on the computation) per node of a given level. Figure 14 shows that when extra work is added, the speedup increases to 4 and the burdened parallelism now becomes the limiting factor. The reason why we see a low burdened parallelism might be the use of reducers and the fact that the number of nodes at a given level of the residual graph can become small after a certain stage of the push-relabel algorithm, causing a high Cilk overhead.

Since the global-update heuristic accounts for about $\frac{1}{3}$ of the total time of the *hipr* algorithm, getting a speedup of 2 for the heuristic should improve the total running time by a factor of $\frac{6}{5}$.

## 7.2 Parallel Push-Relabel

Table 2 summarizes the results for the described variations of the parallel push-relabel. The best results were seen for the discharge-chain algorithm when run with the concurrent global-update heuristic on $rmfl$ graphs. When run on 8 workers, this algorithm outperforms *hipr* on the $rmfl$ graph presented in the table. The performance of each algorithm is analyzed in more details below.

### *7.2.1 Discharge-Chain*

Two versions of the discharge-chain algorithm were evaluated: one that syncs all the threads to perform a synchronous global-update and one that runs the global-update concurrently. Running Cilkview on the first version reports sufficient parallelism (33.56) but a very low burdened parallelism (mostly around 1), which yields no speedup and indicates that there is a significant overhead in performing the spawn and sync operations and possibly some lock contention, see Figure 15. All threads have to be suspended and synched for the synchronous global update, which slows down the parallel code significantly and creates more Cilk overhead. However, with this synchronization step removed using the concurrent heuristic, the code performs much better.

The concurrent global-update improves performance significantly and we can actually observe a speedup of roughly 2 on 8 processors. Figure 16 shows the running time of this algorithm on *rmfl* and *rmfw* graphs. We found that the performance is better on *rmfl* graphs. Since we need at least 2 workers to run the algorithm (one worker needs to be dedicated to running the concurrent heuristic), we could not obtain Cilkview results.

### 7.2.2 *Coarsened-Discharge*

The size of the batch of accumulated nodes has been varied to improve performance and was set to 64 for the reported runs. This is because for several instances of rmf graphs it was found that on average there are only about 50-60 active nodes inside *all* of the buckets when the *cilk_for* loop is run; therefore, increasing the number of nodes to spawn on has no effect. Running *cilk_for* on such a small number of nodes creates a significant overhead as can be seen in Figure 17 generated by Cilkview. Although the parallelism value of 48.29 reported by Cilkview for the *cilk_for* portion of the main loop is sufficiently high, the burdened parallelism is very low, 0.87, and could be caused by the small number of active nodes in the buckets, as well as some locking overhead.

### 7.2.3 *Local Queues*

We currently have a working implementation of the described local queues algorithm; however, it is highly not optimized. There is currently no dynamic adjustment of the parameters controlling the size of the tasks transferred between the global queue and the local queues, which can create significant overhead and result in poor load-balancing. Similarly, the transfer of the nodes from the local queues to the global queue and vice versa can be significantly improved by changing the representation of the queue data structure.

We have observed a wide range of results when running this algorithm on different graphs and with different settings for the size of the tasks. For the test graphs presented in Table 2, we saw no speedup when run on multiple processors. This results can be due to lock overhead and lack of task size adjustment. However, this approach does look promising, since it actually can provide a reasonable amount of work to each processor, if optimized for implementation and design.

## 7.3 Lockfree `push-relabel`

The `hipr` algorithm outperformed all of our lockfree implementations, including those which tried to order the inputs such that the top half of active nodes were discharged in parallel. We believe the reason for this improved performance is that the `hipr` algorithm correctly takes into account discharge dependencies when calculating the discharge order. The lockfree algorithms essentially take a snapshot of the discharge

| | algorithm | sequential | parallel | speedup |
|---|---|---|---|---|
| rmfl | discharge-chain | 126.63 | 108.98 | 1.16 |
| | discharge-chain-conc | 131.8 | 54.07 | 2.44 |
| | coarse-discharge | 85.83 | 116.79 | 1.16 |
| | local-queues | 176.85 | 166.31 | 1.064 |
| rmfw | discharge-chain | 94.44 | 86.11 | 1.1 |
| | discharge-chain-conc | 116.35 | 65.57 | 1.77 |
| | coarse-discharge | 102.24 | 133.3 | 0.77 |
| | local-queues | 186.8 | 202.51 | 0.92 |

**Table 2:** Running times (in seconds) of the parallel push-relabel algorithms. Parallel times were obtained on 8 workers. The *rmfl* graph has parameters a = 50 and b = 1000 and a total of 2500000 nodes and 12297500 edges; the *rmfw* graph has parameters a = 200 and b = 50 and a total of 2000000 nodes and 9920000 edges. The *hipr* algorithm runs in 88.77 s on *rmfl* and 126.66 s on *rmfw*

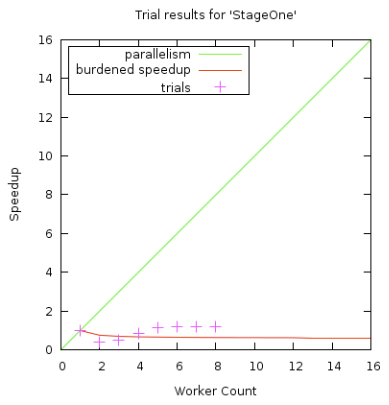

**Figure 15:** Parallel push-relabel using discharge-chain on $rmf(a = 50, b = 1000)$

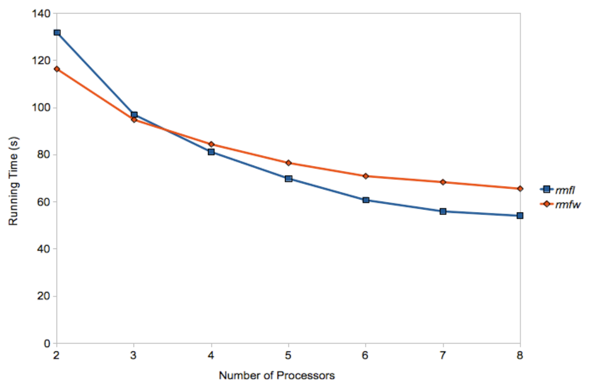

**Figure 16:** Parallel push-relabel using discharge-chain on $rmf(a = 50, b = 1000)$ with concurrent global-update

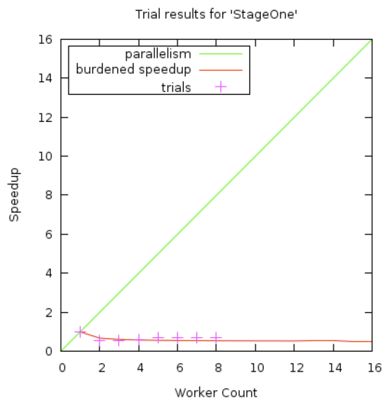

**Figure 17:** Parallel push-relabel using coarsened-discharge on $rmf(a=50, b=1000)$

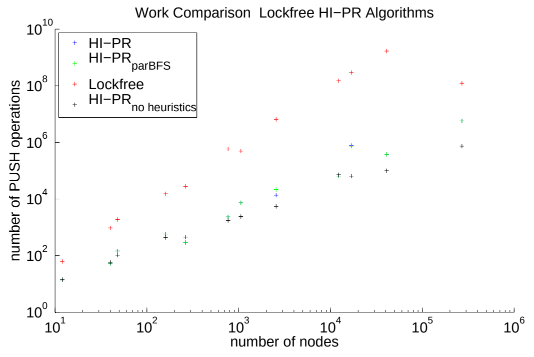

**Figure 18:** Amount of `push` operations done by lockfree and `hipr` algorithms

order and do not update this ordering while discharging all the nodes within the top half of the active set. This coarsened ordering may degrade performance because of bad discharge node interactions. Furthermore, none of our lockfree algorithm utilize the `global update` or `gap` heuristics. Figure 18 shows that the amount of `push` operations performed by our lockfree algorithm is much more than work done by `hipr` ( even without the `global update` or `gap` heuristics ). It is worth noting that we improved upon the lockfree algorithm by Hong [2] and implemented a fully lockfree `push-relabel` alogrithm. The algorithm is of theoretical interest since it can take full advantage of $O(V)$ processors because each node can be processed completely in parallel and without lock contention. To our knowledge this is the first fully lockfree implementation of a `push-relabel` algorithm which runs in $O(V^2E)$ time [2].

## 8. FURTHER WORK

All the presented algorithms need to be further optimized. Especially the local-queue algorithm requires more tuning. The concurrent global update needs to be incorporated into all the existing algorithms for comparison. More input graph variety needs to be generated, in order to establish which graphs lend themselves to a better parallelization. It would be interesting to explore alternatives to locking and algorithms other than *hipr* to parallelize.

## 9. REFERENCES

[1] B. V. Cherkassky and A. V. Goldberg, “On implementing push-relabel method for the maximum flow problem,” Stanford University, Stanford, CA, USA, Tech. Rep., 1994.

[2] B. Hong, “A lock-free multi-threaded algorithm for the maximum flow problem,” in *IPDPS*. IEEE, 2008, pp. 1–8.

[3] A. V. Goldberg and R. E. Tarjan, “A new approach to the maximum flow problem,” in *STOC ’86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1986, pp. 136–146.

[4] R. Anderson and J. Setubal, “On the parallel implementation of goldberg’s maximum flow algorithm,” in *4th Annual Symbosium Parallel Algorithms and Architectures (SPAA-92 )*, San Diego, CA, July 1992, pp. 168–177.

[5] C. E. Leiserson and T. B. Schardl, “A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers),” 2010.

[6] 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 *The 18th ISCA International Conference on Parallel and Distributed Computing Systems (PDCS 2005)*, September 12-14, 2005.