Efficient Implementation of a Synchronous Parallel Push-Relabel Algorithm

Niklas Baumstark[1], Guy Blelloch[2], and Julian Shun[2]

[1] Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
niklas.baumstark@gmail.com
[2] Computer Science Department, Carnegie Mellon University, USA
{guyb,jshun}@cs.cmu.edu

Abstract. Motivated by the observation that FIFO-based push-relabel algorithms are able to outperform highest label-based variants on modern, large maximum flow problem instances, we introduce an efficient implementation of the algorithm that uses coarse-grained parallelism to avoid the problems of existing parallel approaches. We demonstrate good relative and absolute speedups of our algorithm on a set of large graph instances taken from real-world applications. On a modern 40-core machine, our parallel implementation outperforms existing sequential implementations by up to a factor of 12 and other parallel implementations by factors of up to 3.

1 Introduction

The problem of computing the maximum flow in a network plays an important role in many areas of research such as resource scheduling, global optimization and computer vision. It also arises as a subproblem of other optimization tasks like graph partitioning. There exist near-linear approximate algorithms for the problem [20], but exact solutions can in practice be found even for very large instances using modern algorithms. It is only natural to ask how we can exploit readily available multi-processor systems to further reduce the computation time. While a large fraction of the prior work has focused on distributed and parallel implementations of the algorithms commonly used in computer vision, fewer publications are dedicated to finding parallel algorithms that solve the problem for other graph families.

To assess the practicality of existing algorithms, we collected a number of benchmark instances. Some of them are taken from a common benchmark suite for maximum flow and others we selected specifically to represent various applications of maximum flow. Our experiments suggest that Goldberg's *hi_pr* program (a highest label-based push-relabel implementation) which is often used for comparison in previous publications is not optimal for most of the graphs that we studied. Instead, push-relabel algorithms processing active vertices in first-in-first-out (FIFO) order seems to be better suited to these graphs, and at the same time happen to be amenable for parallelization. We proceeded to design and implement our own shared memory-based parallel algorithm for the

maximum flow problem, inspired by an old algorithm and optimized for modern shared-memory platforms. In contrast to previous parallel implementations we try to keep the usage of atomic CPU instructions to a minimum. We achieve this by employing coarse-grained synchronization to rebalance the work and by using a parallel version of global relabeling instead of running it concurrently with the rest of the algorithm.

We are able to demonstrate good speedups on the graphs in our benchmark suite, both compared to the best sequential competitors, where we achieve speedups of up to 12 with 40 threads, and to the most recent parallel solver, which we often outperform by a factor of three or more with 40 threads.

2   Preliminaries and Related Work

We consider a directed graph G with vertices V, together with a designated source s and sink t, where $s \neq t \in V$ as well as a capacity function $c : V \times V \to \mathbb{R}_{\geq 0}$. The set of edges is $E = \{(v, w) \in V \times V \mid c(v, w) > 0\}$. We define $n = |V|$ and $m = |E|$. A *flow* in the graph is a function $f : E \to \mathbb{R}$ that is bounded from above by the capacity function and respects the *flow conservation* and *asymmetry* constraints

$$\forall w \in V : \qquad \sum_{(v,w) \in E, v \neq w} f(v, w) = \sum_{(w,x) \in E, w \neq x} f(w, x) \qquad (1)$$

$$\forall v, w \in V : \qquad f(v, w) = -f(w, v) \qquad (2)$$

We define the *residual graph* G_f with regard to a specific flow f using the residual weight function $c_f(v, w) = c(v, w) - f(v, w)$. The set of residual edges is just $E_f = \{(v, w) \in V \times V \mid c_f(v, w) > 0\}$. The *reverse residual graph* G_f^R is the same graph with each edge inverted.

A *maximum flow* in G is a flow that maximizes the *flow value*, i.e. the sum of flow on edges out of the source. It so happens that a flow is maximum if and only if there is no path from s to t in the residual graph G_f [9, Corallary 5.2]. The maximum flow problem is the problem of finding such a flow function. It is closely related to the minimum cut problem, which asks for a disjoint partition $(S \subset V, T \subset V)$ of the graph with $s \in S, t \in T$ that minimizes the cumulative capacity of edges that cross from S to T. It can be shown that the value of a maximum flow is equal to the value of a minimum cut and a minimum cut can be easily computed from a given maximum flow in linear time as the set of vertices reachable from the source in the residual graph [9, §5].

2.1   Sequential Max-Flow and Min-Cut Computations

Existing work related to the maximum flow problem is generally split into two categories: work on algorithms specific to computer vision applications and work on general-purpose algorithms. Most of the algorithms that work well for the

type of grid graphs found in computer vision tend to be inferior for other graph families and vice versa [11, Concluding Remarks]. In this paper we aim to design a general-purpose algorithm that performs reasonably well on all sorts of graphs.

Traditional algorithms for the maximum flow problem typically fall into one of two categories: *Augmenting path-based* algorithms directly apply the *Ford–Fulkerson theorem* [9, Corallary 5.2] by incrementally finding augmenting paths from s to t in the residual graph and increasing the flow along them. They mainly differ in their methods of finding augmenting paths. Modern algorithms for minimum cuts in computer vision applications such as [11] belong to this family. *Preflow-based* algorithms do not maintain a valid flow during their execution and instead allow for vertices to have more incoming than outgoing flow. The difference in flow on in-edges and out-edges of a vertex is called *excess*. Vertices with positive excess are called *active*. A prominent member of this family is the classical *push-relabel* algorithm due to Goldberg and Tarjan [13]. It maintains vertex labels that estimate the minimal number of edges on a path to the sink. Excess flow can be *pushed* from a vertex to a neighbor by increasing the flow value on the connecting edge. Pushes can only happen along *admissible* residual edges to vertices of lower label. When none of the edges out of an active vertex are admissible for a push, the vertex gets *relabeled* and to a higher label. It is crucial for practical performance of push-relabel that the labels estimate the sink distance as accurately as possible. A simple way to keep them updated is to regularly run a BFS in the reverse residual graph to set them to the exact distance. This optimization is called *global relabeling*.

The more recent *pseudoflow* algorithm due to Hochbaum [16] does not need global relabeling and uses specialized data structures that allow for pushes along more than one edge. Implementations of push-relabel algorithms and Hochbaum's algorithm differ mainly in the order in which they process active vertices. *Highest label*-based implementations process active vertices in order of decreasing labels, while FIFO-based implementations select active vertices in queue order. Goldberg's *hi_pr* program [10] uses the former technique and is considered one of the fastest generic maximum flow solvers. It is often used for comparison purposes in related research. For push-relabel and Hochbaum's algorithm, it is beneficial to compute merely a *maximum preflow* that maximizes the cumulative flow on in-edges of the sink, rather than a complete flow assignment. In the case where we are looking only for a minimum cut this is already enough. In all the other cases, computing a valid flow assignment for a given maximum preflow can be achieved using a greedy decomposition algorithm that tends to take much less time than the computation of a preflow [6].

2.2   Parallel and Distributed Approaches to the Problem

Parallel algorithms for the maximum flow problem date back to 1982, where Shiloach *et al.* propose a work-efficient parallel algorithm in the PRAM model, based on blocking flows [23]. Most of the more recent work however is based on the push-relabel family of algorithms. With regard to parallelization, it has

the fundamental, distinct advantage that its primitive operations are inherently local and thus largely independent.

As far as we know, Anderson and Setubal give the first implementation of a practical parallel algorithm for the maximum flow problem [1]. In their algorithm, a global queue of active vertices approximates the FIFO selection order. A fixed number of threads fetch vertices from the queue for processing and add newly activated vertices to the queue using locks for synchronization. The authors report speedups over a sequential FIFO push-relabel implementation of up to a factor of 7 with 16 processors. The authors also describe a concurrent version of global relabeling that works in parallel to the asynchronous processing of active vertices. We will refer to this technique as *concurrent global relabeling*. Bader and Sachdeva [2] modify the approach by Anderson and Setubal and introduce the first parallel algorithm that approximates the highest-label vertex selection order used by *hi_pr*.

Hong [17] proposes an asynchronous implementation that completely removes the need for locking. Instead it makes use of atomic instructions readily available in modern processors. Hong and He later present an implementation of the algorithm that also incorporates concurrent global relabeling [18]. Good speedups over a FIFO-based sequential solver and an implementation of Anderson and Setubal's algorithm are reported. There is also a GPU-accelerated implementation of the algorithm [15].

Pmaxflow [25] is a parallel, asynchronous FIFO-based push-relabel implementation. It does not use the concurrent global relabeling proposed by [1] and instead regularly runs a parallel breadth-first search on all processors. They report speedups of up to 3 over *hi_pr* with 32 threads.

3   A Synchronous Parallel Implementation of Push-Relabel

The parallel algorithms mentioned in subsection 2.2 are exclusively implemented in an asynchronous manner and differ mainly in the load-balancing schemes they use and in how they resolve conflicts between adjacent vertices that are processed concurrently. We believe the motivation for using asynchronous methods this is that in the tested benchmark instances, often there is only a handful of active vertices available for concurrent processing at a given point in time. In this work we try to also consider larger instances, where there is an obvious need for accelerated processing and where it might not be possible to solve multiple independent instances concurrently, due to memory limitations. With a higher number of active vertices per iteration a synchronous approach becomes more attractive because less work is wasted on distributing the load.

From initial experiments with sequential flow push-relabel algorithms, we learned the following things: As expected, the average number of active vertices increases with the size of the graph for a fixed family of inputs. Also, on almost all of the graphs we tested, a FIFO-based solver outperformed the highest label-based *hi_pr* implementation. This is somewhat surprising as *hi_pr* is clearly

superior on the standard DIMACS benchmark [6]. These observations led us to an initial design of a simple synchronous parallel algorithm, inspired by an algorithm proposed in the original push-relabel article [13]. After the standard initialization, where all edges adjacent to the source are saturated, it proceeds in a series of iterations, each iteration consisting of the following steps:

1. All of the active vertices are processed in parallel. For each such vertex, its edges are checked sequentially for admissibility. Possible pushes are performed, but the excess changes are only applied to a copy of the old excess values. The final values are copied back in step 4.
2. New temporary labels are computed in parallel for vertices that have been processed in step 1 but are still active.
3. The new labels are applied by iterating again over the set of active vertices in parallel and setting the distance labels to the values computed in step 2.
4. The excess changes from step 1 are applied by iterating over the new set of active vertices in parallel.

These steps are repeated until there are no more active vertices with a label smaller than n. The algorithm is deterministic in that it requires the same amount of work regardless of the number of threads, which is a clear advantage over other parallel approaches that exhibit a considerable increase in work when adding more threads [18]. As soon as there are no more active vertices, we have computed a maximum preflow and can determine a minimum cut immediately or proceed to reconstruct a maximum flow assignment using a sequential greedy decomposition.

It is important to note that in step 1 we modify shared memory from multiple threads concurrently. To ensure correctness, we use atomic fetch-and-add instructions here to update the excess values of neighbor vertices (or rather, copies thereof). Contention on these values is typically low, so overhead caused by cache coherency mechanisms is not a problem. To collect the new set of active vertices for the next iteration we use atomic test-and-set instructions that resolve conflicts when a vertex is activated simultaneously by multiple neighbors, a situation that occurs only very rarely. We want to point out that synchronization primitives are kept to a minimum by design, which to our knowledge constitutes a significant difference to the state-of-the-art.

Instead of running global relabeling concurrently with the rest of the algorithm as done by [1] and [18], we regularly insert a global relabeling step in between certain iterations. The work threshold we use to determine when to do this is the same as the one used by *hi_pr*.[1] The global relabeling is implemented as a simple parallel reverse breadth-first search from the sink. Atomic compare-and-swap primitives are used during the BFS to test whether adjacent vertices have already been discovered. Apart from global relabeling, we also experimented with other heuristics such as *gap relabeling*, described in [6, Chapter 3], but could not achieve speedups by applying them to the parallel case.

---

[1] Global relabeling is performed approximately after every $12n + 2m$ edge scans.

3.1   Improving the Algorithm

We implemented the above algorithm in C++ with OpenMP extensions. For common parallel operations like prefix sums and filter, we used library functions from our *Problem Based Benchmark Suite* [24] for parallel algorithms. Even with this very simple implementation, we could measure promising speedups compared to sequential solvers. However, we conjectured that the restriction of doing at most one relabel per vertex in each iteration has some negative consequences: For one, it hinders the possible parallelism: A low-degree vertex can only activate so many other vertices before getting relabeled. It would be preferrable to imitate the sequential algorithms and completely discharge each active vertex in one iteration by alternating push and relabel operations until its excess is zero. Also, the per-vertex work is small. As we parallelize on a vertex level, we want to maximize the work per vertex to improve multi-threaded performance in the common case that only few vertices are active.

To be able to relabel a vertex more than once during one iteration, we need to allow for non-determinism and develop a scheme to resolve conflicts between adjacent vertices when both are active in the same iteration. We experimented with several options here, including the lock-free vertex discharge routine introduced by Hong and He [18]. Another approach turned out to be more successful and works without any additional synchronization: In the case where two adjacent vertices v and w are both active, a deterministic winning criterion is used to decide which one of the vertices owns the connecting edges during the current iteration. We say that v wins the competition if $d(v) < d(w)-1$ or $d(v) = d(w)+1$ or $v < w$ (the latter condition is a tie-breaker in the case where $d(v) = d(w)$). In this case, v is allowed to push along the edge (v, w) but w is not allowed to push along the edge (w, v). The discharge of w is thus aborted if (w, v) is the last remaining admissible edge. The particular condition is chosen such that one of the vertices can get relabeled past the other, to ensure progress. There is an edge case to consider where two adjacent vertices v and w are active, v owns the connecting edge but w is still relabeled because the residual capacity $c_f(w, v)$ is zero. We allow this scenario, but apply relabels only to a copy of the distance function d, called d'. The new admissibility condition for an edge $(x, y) \in E_f$ becomes $d'(x) = d(y) + 1$, i.e. the old distance of y is considered. The new labels are applied at the end of the iteration.

By using this approach, we ensure that for each sequence of push and relabeling operations in our algorithm during one iteration, there exists an equivalent sequence of pushes and relabels that is valid with regard to the original admissibility conditions from [13]. Thus the algorithm is correct as per the correctness proof for the push-relabel algorithm.

The resulting algorithm works similar to the simple algorithm stated above, but mixes steps 1 and 2, to enable our changes. We will refer to our own implementation of this algorithm as *prsn* in the remainder of this document. Pseudocode can be found in the longer version of our paper [4].

4   Evaluation

4.1   A Modern Benchmark Suite for Flow Computations

Traditionally, instance families from the twenty-year-old DIMACS implementation challenge [19] are used to compare the performance of maximum flow algorithms. Examples of publications that use primarily these graph families are [1, 2, 5, 6, 12, 18]. We believe that the instance families from the DIMACS benchmark suite do not accurately represent the flow and cut problems that are typically found today. Based on different applications where flow computations occur as subproblems, we compiled a benchmark suite for our experiments that we hope will give us better insight into which approaches are the most successful in practice.

Saito *et al.* describe how minimum cut techniques can be used for spam detection on the internet [21]: They observe that generally spam sites link to "good" (non-spam) sites a lot while the opposite is rarely the case. Thus the sink partition of a minimum cut between a seed set of good and spam sites is likely to contain mostly spam sites. We used their construction on a graph of pay level domains provided by a research group at the University of Mannheim with edges of capacity 1 between domains that have at least one hyperlink.[2] A publicly accessible spam list[3] and a list of major internet sites[4] helped us build good and bad seed sets of size 100 each, resulting in the *pld_spam* graph.

Very similar constructions can be used for community detection in social networks [8] It is known that social networks, the Web and document graphs like Wikipedia share a lot of common characteristics, in particular sparsity and low diameter. Halim *et al.* include in their article a comprehensive collection of references that observe these properties for different classes of graphs [14]. Based on this we believe that *pld_spam* is representative of a more general class of applications that involve community detection in such graphs.

Graph partitioning software such as *KaHIP* due to Sanders and Schulz commonly use flow techniques internally [22]. The KaHIP website[5] provides an archive of flow instances for research purposes which we used as part of our test suite. We included multiple instances from this suite, because the structure of the flow graphs is very close to the structure of the input graphs and those cover a wide range of practical applications.

The input graphs for KaHIP are taken from the 10th DIMACS graph partitioning implementation challenge [3]: *delaunay* is a family of graphs representing the Delaunay triangulations of randomly generated sets of points in the plane. *rgg* is a family of random geometric graphs generated from a set of random points in the unit square. Points are connected via an edge if their distance is smaller than $0.55 \cdot \frac{\ln n}{n}$. *europe.osm* is the largest amongst a set of street map graphs. *nlp-kkt240* is the graph representation of a large sparse matrix arising in non-linear

---

[2] http://webdatacommons.org/hyperlinkgraph/
[3] http://www.joewein.de/sw/blacklist.htm
[4] https://www.quantcast.com/top-sites
[5] http://algo2.iti.kit.edu/documents/kahip/

optimization. For the cases where graphs of different sizes are available (*delaunay* and *rgg*), we included the largest instance whose internal representation fits into the main memory of our test machine.

As a third application, in computer vision a lot of different problems reduce to minimum cut: For reference, Fishbain and Hochbaum [7, Section 3.2] describe various examples of applications. We included *BL06-camel-lrg*, an instance of multi-view reconstruction from the vision benchmark suite of the University of Western Ontario.[6]

For completeness, we also included instances of two of the harder graph families from the DIMACS maximum flow challenge, *rmf_wide_4* and *rlg_wide_16*, which are described for example by [6]. Table 1 shows the complete list of graphs we used in our benchmarks, together with their respective vertex and edge counts, as well as the maximum edge capacities.

Table 1. Properties of our benchmark graph instances. The maximum edge capacity is excluding source or sink adjacent edges.

graph name	num. vertices	num. edges	max. edge capacity
rmf_wide_4	1,048,576	5,160,960	10000
rlg_wide_16	4,194,306	12,517,376	30000
delaunay_28	161,061,274	966,286,764	1
rgg_27	80,530,639	1,431,907,505	1
europe.osm	15,273,606	32,521,077	1
nlpkkt240	8,398,082	222,847,493	1
pld_spam	42,889,802	623,056,513	1
BL06-camel-lrg	18,900,002	93,749,846	16000

4.2   Comparison and Testing Methodology

Our aim was to compare the practical efficiency of our algorithm to the sequential and parallel state-of-the-art on a common Intel architecture. For comparison with sequential implementations, we selected the publicly available *f_prf*[7], *hi_pr* and *hpf*[8] programs, implementing FIFO and highest label-based push-relabel and Hochbaum's pseudoflow algorithm, respectively. For *hi_pr*, we did not find a canonical URL for the most recent 3.7 version of the code and instead used the copy embedded in a different project.[9] Our results show that *hpf* is the best sequential solver for our benchmark suite, only outperformed by *f_prf* on the *pld_spam* graph. The most recent parallel algorithm is the asynchronous lock-free algorithm by Hong and He [18]. Since it has no public implementation, we implemented their algorithm based on the pseudocode description. We will refer to it as *hong_he* in the remainder of this document. Our own implementation of the algorithm described in subsection 3.1 is called *prsn*. We also experimented with a parallel push-relabel implementation that is part of the Galois project.[10]

---

[6] http://vision.csd.uwo.ca/data/maxflow/

[7] http://www.avglab.com/soft.html

[8] http://riot.ieor.berkeley.edu/Applications/Pseudoflow/maxflow.html

[9] https://code.google.com/p/pmaxflow/source/browse/trunk/goldberg/hipr

[10] http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/preflow_push

Although their code is competitive on certain small inputs, it did not complete within a reasonable amount of time on larger instances.

To eliminate differences due to graph representation and initialization overhead, we modified the algorithms to use the same internal graph representation, namely adjacency arrays with each edge also storing the residual capacity of its reverse edge, as described in [2]. For all five algorithms we only measured the time to compute the maximum preflow, not including the data structure initialization time. The reconstruction of the complete flow function from there is the same in every case and takes only a negligible fraction (less than 3 percent) of the total sequential computation time for all of our input graphs. We measured each combination of algorithm and input at least five times.

We carried out the experiments on a NUMA Intel Nehalem machine. It hosts four Xeon E7-8870 sockets clocked at 2.4 GHz per core, making for a total of 40 physical and 80 logical cores. Every socket has 64 GiB of RAM associated with it, making for a total of 256 GiB.

4.3   Results

The longer version of our paper contains comprehensive tables of the absolute timings we collected in all our experiments [4]. *rgg_27*, *delaunay_28* and *nlpkkt240* are examples of graphs where an effective parallel solution is possible: Figure 1 shows that both *hong_he* and *prsn* outperform *hpf* in the case of *rgg_27*; furthermore *prsn* is three times faster than *hong_he* with 32 threads. The speedup plot for *delaunay_28* looks almost identical to the one for *rgg_27*. In the case of *nlpkkt240*, we can tell from Figure 2 that *prsn* outperforms *hpf* with four threads and achieves a speedup of 5.7 over *hpf* with 32 threads. *hong_he* does not achieve any absolute speedup even with 40 threads. *prsn* does remarkably well on our spam detection instance *pld_spam*: Even with one thread, our implementation outperforms *hpf* and *hi_pr* and is on par with *f_prf*. Figure 3 shows that with 40 threads, an absolute speedup of 12 is achieved over the best sequential run. We noticed here that the algorithm spends most of the time performing a small number of iterations on a very large number of active vertices, which is very advantageous for parallelization. Note that *hong_he* did not finish on the *pld_spam*
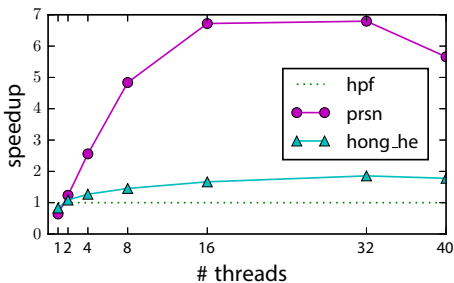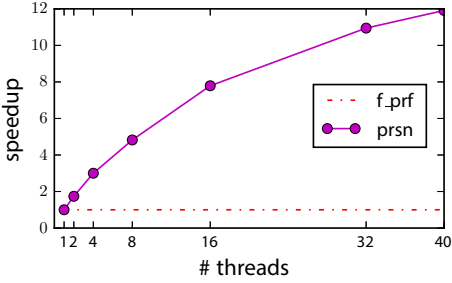

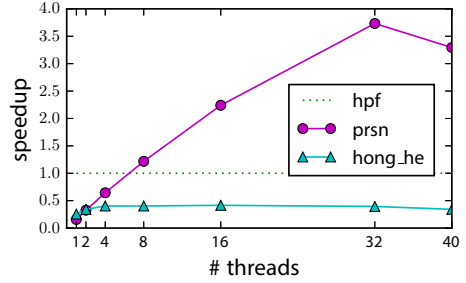**Fig. 1.** Speedup for *rgg_27* compared to the best single-threaded timing
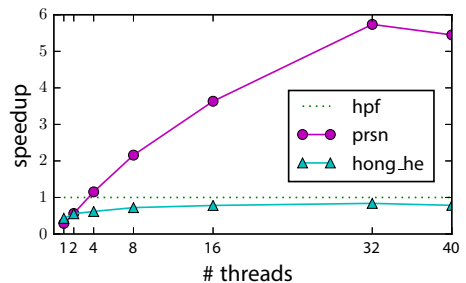
Fig. 2. Speedup for *nlpkkt240* compared to the best single-threaded timing

benchmark after multiple hours of run time. We conjecture that this is because the algorithm is simply very inefficient for this particular instance and were able to confirm that this is the case by reimplementing the same vertex discharge in the sequential *f_prf* program. Before each push, it scans all the edges of a vertex to find the neighbor with the lowest label. This is necessary in *hong_he* because the algorithm does not maintain the label invariant $d(x) \leq d(y) + 1$ for all $(x, y) \in E_f$. The modified *f_prf* also did not finish solving the benchmark instance within a reasonable time frame.
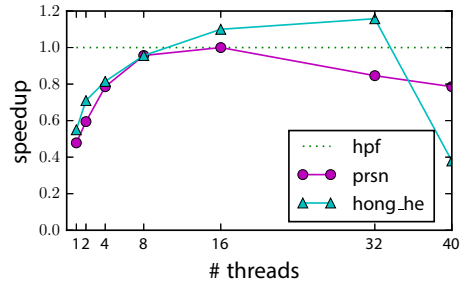

Fig. 3. Speedup for *pld_spam* compared to the best timing. *hong_he* did not finish in our experiments

Fig. 4. Speedup for *BL06-camel-lrg* compared to the best single-threaded timing
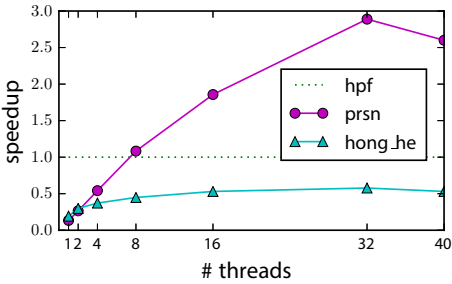
Fig. 5. Speedup for *rlg_wide_16* compared to the best sequential timing

Fig. 6. Speedup for *europe.osm* compared to the best single-threaded timing

BL06-camel-lrg is a benchmark from computer vision. Figure 4 shows that *prsn* is able to outperform *hpf* with 8 threads and achieves a speedup of almost four with 32 threads. *hpf* has in turn been shown to perform almost as well as the specialized BK algorithm on this benchmark [7].

As we can tell from Figure 5, in the case of *rlg_wide_16*, *prsn* requires eight threads to outperform *hpf* and achieves an absolute speedup of about three with 32 threads. *europe.osm* appears to be a hard instance for the parallel algorithms, as shown in Figure 6: Only *hong_he* achieves a small speedup with 32 threads. Both parallel algorithms fail to outperform the best sequential algorithm in the case of the *rmf_wide_4* graph. In all cases, making use of hyper-threading by

running the algorithms with 80 threads did not yield any performance improvements, which we attribute to the fact that the algorithm is mostly memory bandwidth-bound.

Overall, different graph types lead to different behaviour of the tested algorithms. We have shown that especially for large, sparse graphs of low diameter, our algorithm can provide significant speedups over existing sequential and parallel maximum flow solvers.

5   Conclusion

In this paper, we presented a new parallel maximum flow implementation and compared it with existing state-of-the-art sequential and parallel implementations on a variety of graphs. Our implementation uses coarse-grained synchronization to avoid the overhead of fine-grained locking and hardware-level synchronization used by other parallel implementations. We showed experimentally that our implementation outperforms the fastest existing parallel implementation and achieves good speedup over existing sequential implementations on different graphs. Therefore, we believe that our algorithm can considerably accelerate many flow and cut computations that arise in practice. To evaluate the performance of our algorithm, we identified a new set of benchmark graphs representing maximum flow problems occuring in practical applications. We believe this contribution will help in evaluating maximum flow algorithms in the future.

References

1. Anderson, R., Setubal, J.: A Parallel Implementation of the Push–Relabel Algorithm for the Maximum Flow Problem. Journal of Parallel and Distributed Computing (1995)
2. Bader, D., Sachdeva, V.: A Cache-aware Parallel Implementation of the Push–Relabel Network Flow Algorithm and Experimental Evaluation of the Gap Relabeling Heuristic (2006)
3. Bader, D.A., Meyerhenke, H., Sanders, P., Wagner, D. (eds.): Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings, Contemporary Mathematics, vol. 588. American Mathematical Society (2013)
4. Baumstark, N., Blelloch, G., Shun, J.: Efficient Implementation of a Synchronous Parallel Push-Relabel Algorithm. CoRR abs/1507.01926 (2015)
5. Chandran, B.G., Hochbaum, D.S.: A Computational Study of the Pseudoflow and Push–Relabel Algorithms for the Maximum Flow Problem. Operations Research 57(2), 358–376 (2009)

6. Cherkassky, B.V., Goldberg, A.V.: On Implementing Push–Relabel Method for the Maximum Flow Problem. Algorithmica 19(4), 390–410 (1997)
7. Fishbain, B., Hochbaum, D.S., Mueller, S.: A Competitive Study of the Pseudoflow Algorithm for the Minimum st Cut Problem in Vision Applications. Journal of Real-Time Image Processing (2013)
8. Flake, G.W., Lawrence, S., Giles, C.L.: Efficient Identification of Web Communities. In: KDD 2000, pp. 150–160. ACM, New York (2000)
9. Ford, L., Fulkerson, D.: Flows in Networks (1962)
10. Goldberg, A.: hi_pr Maximum Flow Solver, `http://www.avglab.com/soft.html`
11. Goldberg, A.V., Hed, S., Kaplan, H., Tarjan, R.E., Werneck, R.F.: Maximum Flows by Incremental Breadth-first Search. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 457–468. Springer, Heidelberg (2011)
12. Goldberg, A.: Two-level Push–Relabel Algorithm for the Maximum Flow Problem. Algorithmic Aspects in Information and Management (2009)
13. Goldberg, A., Tarjan, R.: A New Approach to the Maximum-Flow Problem. Journal of the ACM (JACM) 35(4), 921–940 (1988)
14. Halim, F., Yap, R., Wu, Y.: A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs. In: ICDCS 2011, pp. 192–202 (June 2011)
15. He, Z., Hong, B.: Dynamically Tuned Push–Relabel Algorithm for the Maximum Flow Problem on CPU-GPU-Hybrid Platforms. In: IPDPS 2010 (April 2010)
16. Hochbaum, D.S.: The Pseudoflow Algorithm: A New Algorithm for the Maximum-Flow Problem. Operations Research 56(4), 992–1009 (2008)
17. Hong, B.: A Lock-Free Multi-Threaded Algorithm for the Maximum Flow Problem. In: MTAAP 2008 (April 2008)
18. Hong, B., He, Z.: An Asynchronous Multithreaded Algorithm for the Maximum Network Flow Problem with Nonblocking Global Relabeling Heuristic. IEEE Transactions on Parallel and Distributed Systems 22(6), 1025–1033 (2011)
19. Johnson, D.S., McGeoch, C.C. (eds.): Network Flows and Matching: First DIMACS Implementation Challenge. American Mathematical Society, Boston (1993)
20. Kelner, J.A., Lee, Y.T., Orecchia, L., Sidford, A.: An Almost-linear-time Algorithm for Approximate Max Flow in Undirected Graphs, and Its Multicommodity Generalizations. In: SODA 2014, pp. 217–226. SIAM (2014)
21. Saito, H., Toyoda, M., Kitsuregawa, M., Aihara, K.: A Large-Scale Study of Link Spam Detection by Graph Algorithms. In: AIRWeb 2007 (2007)
22. Sanders, P., Schulz, C.: Engineering multilevel graph partitioning algorithms. In: Demetrescu, C., Halldórsson, M.M. (eds.) ESA 2011. LNCS, vol. 6942, pp. 469–480. Springer, Heidelberg (2011)
23. Shiloach, Y., Vishkin, U.: An $O(n^2 \log n)$ Parallel Max-Flow Algorithm. J. Algorithms 3(2), 128–146 (1982)
24. Shun, J., Blelloch, G.E., Fineman, J.T., Gibbons, P.B., Kyrola, A., Simhadri, H.V., Tangwongsan, K.: Brief Announcement: The Problem Based Benchmark Suite. In: SPAA 2012, pp. 68–70. ACM (2012)
25. Soner, S., Ozturan, C.: Experiences with Parallel Multi-threaded Network Maximum Flow Algorithm. Partnership for Advanced Computing in Europe (2013)