Speeding up Maximum Flow Computations on Shared Memory Platforms

Bachelor Thesis of

Niklas Baumstark

At the Department of Informatics
Institute for Theoretical Computer Science

Reviewer:	Prof. Dr.rer.nat. Peter Sanders
Second reviewer:	Prof. Dr. Dorothea Wagner
Advisor:	Prof. Dr.rer.nat. Peter Sanders
Second advisor:	Prof. Guy Blelloch

Duration: May 23rd, 2014 – September 22nd, 2014

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, September 7, 2016

.....

(**Niklas Baumstark**)

Contents

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. While the existing algorithms all have at least quadratic worst-case time bounds, in practice it is possible to solve most real-world instances in a reasonable amount of time using modern algorithms, even those with hundreds of millions of nodes and billions of edges. It is only natural to ask how we can exploit readily available multi-processor systems to further decrease the computation time.

While a large part of the previous work has focused on distributed and parallel implementations of the algorithms used in computer vision, fewer publications address the problem of finding efficient parallel algorithms to solve the problem on general graphs. Existing material suggests that it is not an easy task to achieve good speedups over well-optimized sequential implementations. The proposed algorithms are almost exclusively variations of the well-known push–relabel algorithm invented by Goldberg and Tarjan in 1988, due to the localized nature of its primitive operations. However, for efficient implementations, global heuristics are absolutely necessary.

We examined the current state of art of sequential and parallel maximum flow solvers. To assess the practical efficiency of existing algorithms, we compiled a number of benchmark instances, some of which have already been used in related research. We integrated the algorithm implementations into the "Problem Based Benchmark Suite" maintained by CMU professor Guy Blelloch et al. Using the information gathered from the evaluation of these benchmarks, we drew conclusions about what techniques work well in the context of maximum flow, in particular for large, sparse graphs. We identified some problems with existing parallel implementations and tried to solve them.

With these preparations, we proceeded to design and implement our own shared memory-based parallel algorithm for the maximum flow problem. We built our design on the push–relabel based family of algorithms, which turned out to be an overall good method in the sequential case.

We adapted the optimization techniques that we found to be effective earlier to the parallel case. We assessed the performance of our implementation by comparing it to existing max-flow solvers using the assembled benchmark suite. A comparison of different configurations of our algorithm also allowed us to determine the effectiveness of the individual optimizations and heuristics in the parallel case.

2. Acknowledgements

I offer my sincere gratitude to professor Guy Blelloch and Ph.D. student Julian Shun from Carnegie Mellon University for introducing me to the PBBS framework and providing me with various high-quality testing hardware. They helped me more than once to find the right direction to go during my research, both conceptionally and with regard to implementation techniques. This thesis has profited much from their experience and foresight.

I also want to thank Christian Schulz from KIT for generating and providing me with many flow instances from graph partitioning applications.

Professor Peter Sanders from KIT offered many initial ideas before my trip to CMU and helped me lay out and improve the final document after my return, which was very helpful and is greatly appreciated.

3. Preliminaries

Let $G = (V, E, c)$ be a graph with vertices V, directed edges $E \subseteq V \times V$ and a cost function $c : E \to \mathbb{N}_0$. By convention we will use n to denote the number of vertices in a graph and m to denote the number of edges when it is clear from the context which graph we are referring to. We will further assume that for each $(v, w) \in E$, we also have $(w, v) \in E$. This does not restrict generality, because we can just set zero capacities for all the edges that are not in the original graph. Let further $s, t \in V$ be two distinguished vertices in the graph, called the *source* and *sink*, respectively.

A *flow* in the graph G with regard to source s and sink t is a function $f : E \to \mathbb{N}_0$ that fulfills two constraints:

- *capacity constraint*: For all edges $(v, w) \in E$, we have $f(v, w) \leq c(v, w)$
- *flow conservation constraint*: For all vertices $v \in V \setminus \{s, t\}$, we have

$$\sum_{(u,v)\in E} f(u, v) = \sum_{(v,w)\in E} f(v, w)$$

The *value* of a flow f is the cumulative flow on source-adjacent edges:

$$|f| := \sum_{(s,w)\in E} f(s, w)$$

Due to the flow conservation property, we also have

$$|f| = \sum_{(v,t)\in E} f(v, t)$$

The *maximum flow problem* is the problem of finding a flow f that maximizes $|f|$.

The *residual graph* G_f of G with regard to a flow f is the graph $G_f := (V, E_f)$ consisting of non-saturated edges:

$$E_f := \{(v, w) \in E \mid c_f(v, w)\},$$

where the *residual capacity* of an edge is defined as:

$$c_f(v, w) := c(v, w) - f(v, w) + f(w, v)$$

An *augmenting path* is a path in G_f from s to t.

An *s-t cut* in the graph G is a partition $(S, V \setminus S)$ of the graph with $s \in S, t \notin S$. The *capacity* $c(S)$ of an s-t cut $(S, S \setminus V)$ is defined as the cumulative capacity of edges that leave the set S:

$$c(S) := \sum_{\substack{(v,w)\in E \\ v\in S \\ w\notin S}} c(v, w)$$

A *minimum s-t cut* is one that minimizes $c(S)$ and the *minimum cut problem* is the problem of finding such a cut.

3.1 Important Theorems

Let $G = (V, E)$ be a graph. Let f be a flow in G with regard to source s and sink t.

Theorem 3.1.1 (Ford–Fulkerson) *f is a maximum flow iif there is no augmenting path in G_f.*

Proof Can be found in [FF62, §5, Corollary 5.2].

Theorem 3.1.2 (Max-Flow Min-Cut) *Let S be the set of vertices reachable from s in G_f. Then $(S, V \setminus S)$ is a minimum s-t cut iif f is a maximum flow. In that case we also have $c(S) = |f|$.*

Proof Can be found in [FF62, §5].

The latter theorem gives a simple reduction from the s-t minimum cut problem to the maximum flow problem. The reverse reduction is not as straightforward, there is no simple way known to reconstruct a valid flow function from merely an s-t cut in the graph. If one is only interested in the value of the flow, then obviously solving the minimum cut problem is sufficient.

3.2 Algorithms for the Maximum Flow Problem

There are two main families of maximum flow algorithms that play a role in the context of this work.

3.2.1 Augmenting Path-Based Algorithms

This family of algorithms makes direct use of Theorem 3.1.1: While there are augmenting paths in the residual graph, these algorithms find such a path and increase the flow along the path as much as possible. The following pseudocode shows a high-level overview of the algorithm.

```
1 AugmentingPathMaxFlow(G = (V, E), c, s, t):
2     initialize  f ≡ 0
3     while ∃ augmenting path P = (s = p_0, p_1, ..., p_{k-1}, p_k = t) with non−zero capacity:
4         Δ := min_{i=0}^{k-1} c_f(p_i, p_{i+1})
5         for i := 0 to k − 1:
6             c_f(p_i, p_{i+1}) −= Δ
7             c_f(p_{i+1}, p_i) += Δ
8     return f
```

If a depth-first search is used to implement line 3 of the above algorithm, we get the *Ford–Fulkerson algorithm*. An augmenting path can be found in $\mathcal{O}(m)$ and every iteration increases the flow value by at least one. The total runtime is thus bounded by $\mathcal{O}(m \cdot F)$ where F is the maximum flow value of the graph.

A polynomial-time version called the *Edmonds–Karp algorithm* uses breadth-first search to find an augmenting path in each iteration that has the smallest number of edges. With this small modification the number of iterations can be bounded by $\mathcal{O}(nm)$, thus yielding a total time complexity of $\mathcal{O}(nm^2)$.

One of the most efficient augmenting path algorithms is Dinic's algorithm [Din06]: It makes use of the concept of *blocking flows* to speed up the search for augmenting paths and achieves a runtime of $\mathcal{O}(n^2m)$ on general graphs and $\mathcal{O}(m \cdot \min(n^{\frac{2}{3}}, \sqrt{m}))$ if all the edge capacities are zero or one. For graphs arising from bipartite matching problems, it achieves an $\mathcal{O}(m\sqrt{n})$ bound.

Another very prominent algorithm family falls under this category: [BK04] present a new augmenting path-based algorithm that they showed to outperform existing algorithms for the special application of global energy minimization in computer vision. The graphs typically found in this context have a very particular, undirected grid structure, representing adjacency of pixels in a 2- or 3-dimensional computer image. The goal here is to find a global minimum s-t cut, after connecting both source and sink to certain subsets of vertices.

Their algorithm, which is referred to in subsequent publications as the *BK algorithm*, named after its inventors, grows two spanning trees from source and sink concurrently. At some point the trees intersect and an augmenting path is found. The edges on the path are saturated, causing some vertices to become disconnected from their respective trees. The key idea now is to reuse as much as possible from the original trees, by trying to reconnect the disconnected *orphan* subtrees via newly found paths to one of the two trees. Now they continue to grow until a new augmenting path is found.

Curiously, because the algorithm does not augment on shortest paths, no strongly polynomial bound could be established for the runtime of the algorithm as of today. Still it performs remarkably well on real-world image instances and is still the algorithm of choice in the computer vision community. [GHK+11] propose a modification of the algorithm that maintains valid breadth-first trees and thus enjoys the same bounds as the above mentioned Edmonds–Karp algorithm. It is called *incremental breadth-first search (IBFS)* and has been shown to achieve performance comparable to and often better than BK.

3.2.2 Preflow-Based Algorithms

A notable family of maximum flow algorithms that is often used in practice is based on Goldberg and Tarjan's *push–relabel* algorithm [GT88], named after the two key operations it uses. In contrast to traditional augmentation-based solutions, this algorithm does not maintain a balanced flow during its execution, but merely a *preflow*.

A *preflow* is a function $f : E \to \mathbb{N}_0$ that obeys the edge capacities but might not be strictly balanced (it might violate flow conservation). Instead it maintains the following, weaker invariant for all vertices $v \in V \setminus \{s, t\}$:

$$\sum_{(u,v)\in E} f(u, v) \geq \sum_{(v,w)\in E} f(v, w)$$

In other words, the cumulative flow on ingoing edges is allowed to exceed the cumulative flow on outgoing edges of a vertex. We call the (non-negative) difference between the left and right hand side of the inequality the *excess* of the vertex v, denoted by $e(v)$.

The basic push–relabel algorithm starts by *saturating* all source-adjacent edges, i.e. setting $f(s, v) = c(s, v)$ for all $(s, v) \in E$. The set of adjacent vertices will end up with positive excess. Such vertices are called *active*. The algorithm then proceeds to transfer the excess flow towards the sink via a series of *push* operations:

```
1 Push(v, w):
2     Δ := min(e(v), c_f(v, w))
3     c_f(v, w) -= Δ
4     c_f(w, v) += Δ
```

Pushes obviously preserve the preflow property of f. To ensure that as much excess flow as possible will reach the sink, the algorithm maintains distance labels $d : V \to \mathbb{N}_0$. The distance label of a vertex v represents a lower bound on the distance $dist(v, t)$ in the residual graph with regard to the number of non-zero capacity edges one has to traverse to get from v to t. More precisely, the invariant $d(t) = 0$ and $d(v) \leq d(w)+1$ is maintained for all residual edges $(v, w) \in E_f$ with $v \neq s$. The label of the source vertex has the fixed value $d(s) = n$. Any vertex v from which the sink can be reached in the residual graph will have $d(v) < n$ because $n - 1$ is the maximum length of a shortest path.

Excess can only ever pushed from a higher-label vertex to a lower-label vertex if their labels differ by exactly one. A residual edge $(v, w) \in E_f$ is thus *admissible* for a push if and only if v is active, $d(v) = d(w) + 1$ and $c_f(v, w) > 0$.

An active vertex without outgoing admissible edges has to be *relabeled* to maintain the label invariant. The associated primitive looks like follows:

```
1 // precondition: e(v) > 0 and there is no admissible edge (v, w) ∈ E_f
2 Relabel(v):
3     d(v) := min_{(v,w)∈E_f, c_f(v,w)>0} d(w) + 1
```

The relabel operation maintains the label invariant. The complete push–relabel algorithm in pseudocode looks like this:

```
1 PushRelabel(G = (V, E), c, s, t):
2     d(v) := 0          for all s ≠ v ∈ V
3     d(s) := n
4     f(s,v) := c(s,v)   for all (s, v) ∈ E
5     f(v,w) := 0        for all other edges (v, w) ∈ E
```

```
6   while ∃ active vertex v ∈ V \ {s,t}:
7       while e(v) > 0:
8           for all edges (v,w) ∈ E_f, while e(v) > 0:
9               // admissibility check
10              if c_f(v,w) > 0 and d(v) = d(w) + 1:
11                  Push(v,w)
12          if e(v) > 0:
13              // v is still active, needs to be relabeled
14              Relabel(v)
```

The inner loop started at line 7 is called a *discharge* of the vertex v. A series of push and relabel operations is applied until the vertex is no longer active.

In this basic formulation, with arbitrary vertices selected in line 6, we can already prove a runtime bound of $\mathcal{O}(n^2m)$, provided that all the operations can be implemented in constant time. The latter is easily possible using an adjacency list representation of the graph, with explicit reverse edges.

3.2.2.1 Selection Strategies and Global Heuristics

Two common selection rules yield a better asymptotic complexity:

- With the *FIFO selection rule*, the active vertices are maintained in a first-in first-out queue: the front vertex is always selected in line 6. If a new vertex gets activated during a push, it is appended to the back of the queue. This rule results in an $\mathcal{O}(n^3)$ runtime bound, which is asymptotically better than the basic algorithm for dense graphs.
- With the *highest label selection rule*, one of the vertices with maximal distance label among all the active vertices is selected in line 6. This yields an $\mathcal{O}(n^2\sqrt{m})$ runtime, which is again an asymptotic improvement over FIFO selection since $m \in \mathcal{O}(n^2)$.

[GT88] also describe how dynamic tree data structures can be used to speed up the algorithm, achieving a runtime bound of $\mathcal{O}(nm \cdot \log \frac{n^2}{m})$.

It has to be noted that the worst case bounds do not necessarily reflect the actual behaviour of the algorithm on real-world graph instances. Previous work has shown that on a lot of important graph families, the algorithm scales almost linearly in the size of the graph if additional global heuristics are employed. Our experiments confirm this.

One such heuristic is already mentioned in [GT88] without a specific name and called *global relabling* or *global update* in subsequent literature. With this addition, a reverse breadth-first search from the sink is initiated regularly to set every vertex label to the exact distance of the sink from that vertex.

If we are only interested in the minimum cut of the graph, rather than the complete flow function of maximum value, we can employ another heuristic that was proposed by [DM89] (amongst others) and works especially well in conjunction with highest-label selection: The key observation is the fact that if at any point during the execution of the algorithm there exists a label k such that $d(v) \neq k$ for all vertices v, we can conclude that there exists no residual edge $(v, w) \in E_f$ with $d(v) > k$ and $d(w) < k$. This is a direct consequence of the label invariant $d(v) \leq d(w) + 1 \quad \forall (v, w) \in E_f$.

In particular, this means that there can be no residual path from a vertex above label k to a vertex below label k. For this reason, none of the vertices above label k will ever be able to push their excess flow to the sink, which has label zero. We are thus free to simply remove all the vertices v with $d(v) > k$ from the graph and not consider them during the

remaining execution of the algorithm. A label k with the above mentioned property is called a *gap* and the resulting heuristic called a *gap heuristic*.

Of course with this modification of the algorithm, the final flow assignment is only valid in the sink partition of the resulting cut. Flow conservation might be violated in the source partition. It has been shown in [CG97] that gap heuristics do not give a significant speedup in conjunction with FIFO selection and global relabeling in most cases. They are however very effective when combined with highest-label selection. The authors also give an informal explanation of this behaviour.

3.2.3 Notable Implementations

The current state of generic maximum flow solvers is established by at least two open-source implementations of preflow-based algorithms that were repeatedly used as a comparison in previous research:

- The first one is the *HIPR* program published by Andrew Goldberg at `http://www.avglab.com/andrew/soft.html`. It is an improved version of the *H_PRF* algorithm variant described in [CG97].

  It uses highest-label selection in conjunction with global relabeling and gap heuristics. After a *maximum preflow* is determined (i.e. a preflow where the sink is not reachable from any of the active vertices), a second stage employs a greedy flow decomposition algorithm to reconstruct a complete flow assignment, if needed. Note that to determine the maximum flow value and even to find a minimum cut, a maximum preflow assignment is sufficient.

- The second one is an implementation of the more recent *pseudoflow algorithm* presented first by [Hoc08]. It can be obtained from `http://riot.ieor.berkeley.edu/Applications/Pseudoflow/maxflow.html`. We will call it *HPF* in the remainder of this document. During its execution, flow conservation is weakened in both directions: Vertices are also allowed to have *deficit*, which occurs when the cumulative outflow exceeds the inflow. It uses an advanced data structure to maintain a forest of groups of vertices with same-sign flow balance. [CH09] formulate the algorithm in terms of preflows, to highlight the differences to the push–relabel method.

  The published implementation includes several variants, including a version with highest-label selection and gap heuristics. A global relabeling heuristic is not needed for efficiency. Just as *HIPR*, it also only finds a maximum preflow in the first execution stage and then uses a greedy algorithm to reconstruct the complete flow function if necessary.

4. Related Work

As already shown by [GSS82], the problem to find the value of a maximum flow is $\mathcal{P}$-complete. This suggests that there is likely no inherent way to split the problem into subproblems that can be solved independently. Still in practice several approaches have been explored that aim to speed up flow and cut computations on parallel and distributed computer architectures.

[Sib97] present an overview over previous research results regarding the maximum flow problem in the PRAM model. For most of these approaches, we do not feel that they are likely to yield practical implementations that are able to solve the kind of large, sparse graphs found in many real-world applications today. Often they rely on adjacency matrix representations that are out of the question for the type of graphs we are interested in.

Existing work is generally split into two categories: 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. We will consider these categories separately.

4.1 Minimum Cut Algorithms for Computer Vision Applications

[FHM10] provide an overview and comparison of different sequential approaches to solve large vision instances. The results indicate that both the BK algorithm and HPF perform reasonably well for the types of problems, while a standard push–relabel implementation does noticably worse in most cases (often around a factor of two).

[VB12] likewise compare different sequential implementations, including HPF, different push–relabel variants, BK and the newer IBFS algorithm. IBFS and HPF outperform BK in some of the benchmarks, often by a significant margin. In the cases where BK proved superior, IBFS and HPF were not nearly as far behind.

After the publication of [BK04], several attempts have been made to utilize the readily available parallel computing power found in modern processing units.

[DB08] present one of the first attempts to specifically design an algorithm that is both able to solve common vision instances efficiently and scalable in nature. They achieve this by exploiting the grid structure of the graphs involved to derive an efficient partitioning scheme. An algorithm called *region push–relabel* is described that solves a sequence of

maximum flow subproblems in the different regions of the graph. Independent subproblems can be solved simultaneously by using parallel or distributed processors.

A second positive effect of this approach is that not all of the graph has to be kept in main memory during the entire execution of the algorithm. Instead, the different regions can be swapped in and out on demand in an efficient manner, without the thrashing effect that is common for previous algorithms with disadvantageous memory locality. Their experiments verify near-linear speedup for up to eight processing cores on the tested instances.

[LS10b] present a parallel variant of the BK algorithm that is reported to give almost linear speedups over the original version with up to four processing cores on a multi-core Intel machine.

[SK10] choose an exotic approach: They use a formulation of the minimum cut problem as a linear program. The dual of the resulting linear program is decomposed into multiple parts by partitioning the graph. The resulting subproblems are largely independent and solved using a BK variant. Dependencies are resolved by performing a series of iterations called sweeps. The experiments presented in paper indicate good speedups on different graph families. However, [SH13] note that the heuristics used to solve the real-valued max-flow instances arising during the construction are not guaranteed to terminate. In practice, for some larger instances the algorithm did not terminate after a reasonable amount of sweeps during their experiments. Furthermore, [SH13] note that the construction does not seem to work well with more than two partitions. For their experiments they test two and four regions and their results indicate that the speedup between two and four is not as good as would be expected.

[SH13] themselves present significant improvements to the region push relabel algorithm by [DB08]. In particular, augmenting path-based algorithms are used to solve the max-flow subproblems found in the separate regions. It also adds various improvements to several aspects of the algorithm, such as the partitioning scheme used and the scheduling strategy.

Judging from the benchmark results found in the various papers, we consider the implementations presented in [SH13] and [LS10b] to be the state of the art in parallel minimum cut algorithms in computer vision today, with the latter probably being superior in most cases if the instance fits into memory.

4.2 Generic Maximum Flow Algorithms

Research regarding parallel maximum flow or minimum cut algorithms for general graphs seems to be less active and the contributions are less recent than those for vision-specific instances.

Interestingly, almost all of the existing work has been based on the push–relabel family of algorithms. This is by no means coincidence: For one thing, the well-known *HIPR* implementation seems to be one of the fastest publicly available maximum flow solvers, only challenged by the more recent *HPF* implementation. But there is also a fundamental, distinct advantage of push–relabel algorithms over augmenting-path based algorithms when it comes to parallelization: While multiple augmenting paths can be found concurrently, the tasks of saturating the different paths are not independent. For this reason it seems to be hard to design a good parallel path-augmenting algorithm. The push and relabel operations on the other hand are inherently local and mostly independent.

In the original push–relabel paper, [GT88] already mention a parallel version of their algorithm for the PRAM model. In this synchronous formulation, the algorithm proceeds in a

series of *pulses* in which all the active vertices are processed in parallel and relabeled once if necessary. We are not aware of an actual implementation of this straightforward algorithm, so we chose it as the base of our first prototype. Surprisingly, with the addition of parallel global relabeling, it does remarkably well on our benchmark graphs (see subsection 5.2.5).

[AS95] is perhaps the first successful attempt at designing and implementing an efficient, practical algorithm to solve maximum flow in parallel. It essentially uses a global queue of active vertices to approximate the FIFO selection order of the basic push–relabel algorithm. There is a hardcoded number of threads that fetch vertices from the queue and discharge them completely, adding newly activated vertices to the queue as they go along. The algorithm works in an asynchronous manner, resolving shared access to vertices using locking. The authors report speedups over a sequential FIFO push–relabel implementation of up to factor 7 with 16 processors, on a 1995 computer architecture.

This implementation is also the first to include a concurrent version of global relabeling that works in parallel to the asynchronous processing of active vertices. For that to work, some extra information about the global relabeling state needs to be maintained inside the vertex data structure. We will call this technique *concurrent global relabeling* below.

[JW98] propose a distributed implemenentation of push–relabel, which partitions the graph using breadth-first search and uses ideas similar to those found in [DB08] to solve the global problem in a series of passes where the partitions are independently solved using the *HIPR* solver.

[BS06] improve upon [AS95] by providing the first parallel algorithm that approximates the highest-label selection order found in the *HIPR* implementation. The algorithm is also asynchronous in nature and work-stealing priority queues are used to avoid the contention introduced by a globally shared queue. The authors also report significant speedups by using a concurrent gap heuristic. We were unable to grasp the workings of that mechanism however, please refer to subsection 5.2.4 for details.

[Hon08] propose an asynchronous implementation that completely removes the need for locking. Instead it makes use of atomic read–update–write operations readily available in modern processors. The basic algorithm does not incorporate global relabeling, which we know to be essential for a good performance on real-world graphs.

[HH10] extend the algorithm described in [Hon08] by global relabeling, a CUDA-based GPU implementation and dynamic switching between the different modes (GPU parallel and CPU sequential). They observe speedups of up to factor 2 over *HIPR* using an eight-core GPU, on rather small graphs with vertex and edge counts in the order of few millions. What's notable is that the global relabeling scheme used is sequential and not executed concurrently with the rest of the algorithm. Also, for working sets below a certain threshold, where the overhead of copying memory to and from the GPU would be too high, *HIPR* is used instead of a parallel CPU-based implementation. We think that both of these decisions inhibit the potential speedup significantly.

[HH11] present the first efficient CPU-based implementation of [Hon08], including the concurrent global relabeling proposed by [AS95], along with an experimental evaluation. They report good speedups over a FIFO push–relabel implementation due to Goldberg, but no speedup over *HIPR*. We will show later in this document however that the FIFO-based approach seems to do better for a lot of real-world graphs. An implementation of [AS95] is also compared to their algorithm and scales worse on a lot of graphs due to locking overhead, as expected.

The authors also experimented with a version of the algorithm that does not necessarily obey the rule that excess can only be pushed between vertices with labels differing by at

most one. Instead the changed semantics here allow excess to be pushed to any vertex with a lower label. This modification is reported to decrease the runtime by as much as 50% in some cases.

Galois [PMLP+11] is a framework to make it easier to write parallel algorithms, by providing a high-level library of common data structures and execution patterns. It exploits data-parallelism inherent in the problem structure to achieve speedup. As one of the many examples, it contains a FIFO-like push–relabel implementation, described at `http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/preflow_push`. The website claims that it incorporates global relabeling and a gap heuristic, although only the former is actually implemented in the shipped source code.

Pmaxflow [SO13] is an open-source parallel FIFO-like push–relabel implementation. It does not use the concurrent global relabeling proposed by [AS95] and instead regularly runs a parallel breadth-first search on all processors.

[HYW11] choose a totally different approach, providing an implementation of the traditional Ford–Fulkerson augmenting path algorithm on top of *MapReduce* [DG08]. The result is a distributed algorithm to solve maximum flow problems on large graphs such as social networks or the internet. It employs several optimizations similar to those used in the BK algorithm, such as bidirectional search and reusing partial residual paths. Their results indicate that the algorithm scales within a constant factor of the BFS algorithm built into the MapReduce framework, which is impressive. Still we do not feel that for the scenario we are mainly interested in, when the graphs fit into main memory, this approach will lead to the kind of speedup we are hoping for.

The way we understand it, [BS06] and [HH11] can be considered the current state of the art with regard to asynchronous, parallel maximum flow algorithms for general graphs. [HYW11] seems to be the only viable distributed algorithm as of today.

5. Algorithms for the Maximum Flow Problem

5.1 Max-Flow/Min-Cut Applications and our Test Suite

The best known maximum flow algorithms all have at least complexity $\Omega(nm)$ and the algorithms used in practice usually have almost cubic worst-case runtime behaviour $\Omega(n^2\sqrt{m})$ or worse. Nevertheless, optimized solvers with global heuristics, like *HIPR* and the newer *HPF* have proven themselves in practice, making it possible to solve maximum flow problems even on very large graph families with near-linear runtime asymptotics.

Our ultimate goal is to speed up costly maximum flow computation on large graphs for real-world applications, so our first course of action was to try and identify the key scenarios where flow and cut problems occur in practice.

5.1.1 Minimum Cuts in Computer Vision

In computer vision, a lot of different problems reduce to minimum cut: This includes the obvious image segmentation problem, where one wants to optimize some property of the border between two partitions of the image, but also more complex applications like multi-view reconstruction, where multiple 2D views of an object need to be matched to form a 3D representation of the object. There are various other examples, which we will not go into here because computer vision is not the main focus of our interest. Instead we refer to [FHM13, Section 3.2] for an overview over the standard benchmark collection from the Computer Vision Research Group of the University of Western Ontario and to [VB12] for references to more recent applications.

The graphs are typically represented implicitly, with vertexes representing pixels and edges connecting pixels that are adjacent to each other in the original image.

5.1.2 Graph Partitioning with Flows and Cuts

[SS11] describe a local search technique for the *graph partitioning problem* (stated in the paper) that is based on minimum cuts. Specifically, the *adaptive flow iterations* method applies a series of minimum cut computations to improve a two-partition of the graph under the restrictions of the partitioning problem. The *balanced adaptive flow iterations* tactic uses the residual network after a minimum cut is found to optimize the balance of the cut in each iteration. The algorithm uses a multi-layered approach, resulting in different

graphs with decreasing number of vertices, representing hierarchies of vertex sets. Graphs of higher hierarchies contain fewer vertices and have larger edge capacities, representing the number of edges between two sets of vertices.

The algorithm described in the paper is implemented by the *KaHIP* (Karlsruhe High Quality Partitioning) program [SS13] maintained by Christian Schulz, when used with one of the configurations *Strong* or *Eco*.

Conveniently, the *KaHIP* website at `http://algo2.iti.kit.edu/documents/kahip/` provides a comprehensive selection of flow graphs for research purposes which we will use as part of our test suite. According to the author, *KaHIP* uses *HIPR* internally to solve the arising flow subproblems.

5.1.3 Analysis of Social Networks and Web Graphs

[FLG00] describe a minimum cut-based approach to identify *communities* in the web. A community is a set of sites that have more links to members of the community than to the rest of the web.

They propose a cut-based approach to find the community around an initial set (seed) of known members: Essentially, all links are bidirected by introducing a reverse edge if none exists. Unit capacities (edge capacities of value one) are assigned to the now undirected edges. The seed vertices are connected to a newly introduced super-source via edges of infinite capacity. A well-connected, known non-member site is used as the sink. The source partition of a minimum cut in the resulting graph are used as the community around the seed.

[STKA07] propose a similar approach to identify spam sites in the web. They observe that spam sites link to non-spam sites a lot while the reverse is rarely the case. Therefore the connectivity of "good" sites to spam sites is small and a minimum cut algorithm can be applied to identify new spam sites based on a smaller set of known spam sites: A super-source is connected to a known set of good sites via edges of capacity infinity. The seed set of spam sites is connected to a super-sink, again via infinite-capacity edges. The authors show that the sink partition of a minimum cut in the graph constructed in this manner tends to contain almost only spam sites. One can imagine that the same technique can be applied on similar types of graphs, such as social networks which tend to be very large as well.

[YKGF06] make the observation that a *sybil attack* on a decentralized communication network exhibits a structure that can be detected using minimum cut techniques: In this attack scenario, a single user acquires multiple identities and pretends to be multiple, distinct nodes, thus increasing their voting power inside the network illegitimately. The authors observe that while it is often reasonably easy to obtain new identities, it is harder to establish trust relationships to honest users. Thus the graph cut induced by a rogue user's fake identities tends to have a small value and such cuts can be found using maximum flow techniques.

It has been shown that the graphs particularly interesting in these scenarios, social networks and the web, expose certain common characteristics, in particular "small-worldness". We say a graph exhibits a *small-world* structure if the length of the shortest path between a pair of vertices in the graph tends to be small. [HYW11] refer to a comprehensive collection of references that observe this property for different types of graphs, such as the web, social networks and document graphs from Wikipedia. In the context of parallel max-flow, small-worldness is a desirable property because such graphs tend to have short augmenting paths, leading to fewer dependencies between the applicable operations. Also,

in almost all cases, the relevant graphs are sparse, with average vertex degrees ranging in the order of dozens to hundreds.

5.1.4 The Test Suite

Interestingly, when comparing maximum flow algorithms, most references that do not specifically target computer vision applications use problem families from the twenty-year-old first DIMACS implementation challenge [JM93] for experimental evaluation.

Examples of papers that use these graph families more or less exclusively are [AS95], [CG97], [JW98], [BS06], [CH09], [Gol09], [HH10] and [HH11]. Please note that these include the evaluations of *HIPR*, *HPF* and of almost all of the algorithms described in section 4.2, except for [HYW11], which uses the Facebook graph with multiple billions of edges. We ask whether these experiments really justify the dominance of *HIPR* for the graphs modern research is interested in. Our test suite tries to cover at least a subset of the original graph families from the DIMACS challenge as a control, but also aims to include graphs from the real-world applications we mentioned above.

Since a lot of work has already been published in the context of computer vision applications, our main focus lies on non-vision instances. We will use the following graph families:

- As a control, two instances from the from the DIMACS max-flow challenge:
  - *RMF-Wide* is reported by [Gol09] to be the hardest of the DIMACS graph families for modern push–relabel algorithms. It also happens to be the instance family with the lowest speedup for Anderson's parallel algorithm [AS95]. [HH11] also seems to struggle on this family, doing worse then *HIPR* even with 24 processors.

    The instances are generated by the *genrmf* program, using an algorithm due to [GG87]: The program takes as its input four integers a, b, c_1, c_2 and creates a graph consisting of b square grids (called *frames*) of size $a \times a$ each. Each vertex is connected to its four grid neighbors. Furthermore, the vertices in frame i are connected one-to-one with the vertices in frame $i+1$. The permutation between the vertices is chosen randomly. Capacities between vertices inside the same frame are set to $c_2 \cdot a^2$, the other edge capacities are set to integers randomly sampled from the range $[c1, c2]$. The source is a corner of the first frame, the sink is the opposite corner of the last frame.

    For the wide family, we choose some integer x and use $b = 2^x$ and $a = b^2$ as the parameters for the program, along with $c_1 = 0$ and $c_2 = 10^4$. For our experiments, we used $x = 4$, resulting in the graph **genrmf_wide_4**, with 2^{20} vertices and around five times as many edges.

  - *Washington-RLG-Wide* is an easy family of random level graphs generated by the *washington* program. It constructs a grid of a rows of b vertices each where every vertex is connected to three randomly chosen vertices in the next row. The source is connected to all the vertices in the first row and all the vertices in the last row are connected to the sink. All edge capacities are randomly sampled from the range $[1, 10^4]$.

    For the RLG-Wide family, we have 64 rows of 2^x vertices each. We use $x = 16$ in our experiments, resulting in the graph **washington_rlg_wide_16** with 2^{22} vertices and about three times as many edges.

During the execution of a push–relabel algorithm on this graph family, there tend to be a lot of active vertices and low contention, which is very favorable for parallel implementations. We use it as an example of best-case input in terms of self-relative speedup.

- KaHIP instances from different graph partitioning applications:

  These instances were provided by the maintainer Christian Schulz. Some of them are available at the KaHIP web page `http://algo2.iti.kit.edu/documents/kahip/`. The problem here is to partition a graph into two parts with a small number of crossing edges, while maintaining a certain balance between the partition sizes (the exact value of the ϵ parameter is varying from case to case and chosen so that approximately half of the original graph remains in the flow problem). Graphs from the 10th DIMACS implementation challenge about graph partitioning [BMSW13] are used as inputs. A multi-layered approach is applied, with the solution from higher hierarchies being used as the initial solution for the next smaller hierarchy. The initial solution is then improved using the method *adaptive flow iterations* described briefly above. At this point flow instances arise.

  The instances share a number of common characteristics, such as small capacities and the fact that both source and sink have a large degree. We chose some of the largest graphs and graph families for our experiments:

  - **delaunay** is a family of graphs representing the Delaunay triangulations of randomly generated sets of points in the plane. Instance sizes range up to billions of edges. We included the first hierarchy for each graph, which has edge capacities of one. We also included the fourth hierarchy of the large *delaunay_28* graph, which features larger edge capacities (between 1 and 31).
  - **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 $0.55 \cdot \frac{\ln n}{n}$. Instance sizes again range into the order of billions of edges. We included the first hierarchy of each graph.
  - **europe.osm** is the largest amongst a set of street map graphs. It has about 50 million vertices and edges, representing the largest connected component in the European street network provided by the OpenStreetMap project [HW08].
  - **nlpkkt240** is the graph interpretation of a large sparse matrix arising in non-linear optimization. It has 27 million vertices and 373 million edges.
  - **grid** is based on a regular, eight-connected grid graph. We included the graph representing the fourth partitioning hierarchy of a large grid.

- A manually created benchmark instance for spam detection in the Internet:

  We used the construction described in [STKA07] to implement a min-cut based spam detection in a web graph. [MVLB14] provide a hyperlink graph extracted from a comprehensive web crawl performed as part of the Common Crawl project in 2012. The version of the graph that considers a set of web pages with the same top-level domain as one vertex is called *pay level domain graph* or *pld*, as we will refer to it. To this graph, we added a super-source and super-sink, connected to a seed set of non-spam and spam sites, respectively. For the "good" seed we used the top list of Internet sites provided by Quantcast and available at `https://www.quantcast.com/top-sites`. For the "spam" seed we used a spam blacklist obtained from joewein.de LLC, at `http://www.joewein.de/sw/blacklist.htm`. We created two versions of

the resulting network, called **pld_spam_100** and **pld_spam_5000**, with seed sizes 100 and 5000, respectively.

Since social networks and the web share a lot of common characteristics, we believe that this benchmark is representative of the more general class of community detection algorithms based on minimum cuts.

- A 3D segmentation instance from computer vision: For completeness, we also included **BL06-camel-lrg**, an instance of multi-view reconstruction. It is part of the vision benchmark suite of the University of Western Ontario, located at `http://vision.csd.uwo.ca/data/maxflow/`

  We do not expect to do be able to do particularly well with our push–relabel-based approaches on this graph, seeing as multiple sources, including [VB12], report poor performance of both *HIPR* and *F_PRF* on such instances.

5.2 Experiences with Existing Algorithms and Implementations

5.2.1 Sequential Competition

To establish a base line and get an initial idea of the performance of different approaches, we executed the test suite on a few different sequential solvers:

- *HPF*, obtained from `http://riot.ieor.berkeley.edu/Applications/Pseudoflow/maxflow.html`. We used the *pseudo_fifo* variant because it seems to be the overall fastest of the alternatives.
- *HIPR*, due to Goldberg and obtained from `http://www.avglab.com/andrew/soft.html`. Details about the workings of both *HPF* and *HIPR* can be found in subsection 3.2.3.
- *F_PRF*, also available from `http://www.avglab.com/andrew/soft.html`, a simple FIFO-based push–relabel implementation.
- The push–relabel example included in the *dtree* library written and maintained by David Eisenstat and available at `http://www.davideisenstat.com/dtree/`. It serves as an example of an implementation using dynamic trees, which in theory has better asymptotic bounds than all the other implementations. The example reimplements *HIPR* in two different variants, with dynamic trees and without. The version without dynamic trees is competitive with Goldberg's implementation according to the author, which we confirmed in preliminary experiments.

The sequential timings can be found in Table 7.2. Perhaps surprisingly, *F_PRF* shows superiour performance compared to *HIPR* although it uses a suboptiomal graph representation, except for the DIMACS instances where *HIPR* does in fact do significantly better, as expected. We can also see that *HPF* has the best overall performance, but performs significantly more than twice as good as *F_PRF* only in a few cases, including the selected DIMACS challenges and the computer vision instance. It performs worse than *F_PRF* in the spam detection benchmarks.

Both of these observations are very good news from a parallelization standpoint: FIFO selection is much easier to approximate in a parallel setting than highest-label selection. Also, it seems likely that the complexity of *HPF* is not necessary to get good performance on real-world, non-vision graphs.

5.2.2 Galois

We obtained the Galois framework from `http://iss.ices.utexas.edu/?p=projects/galois/benchmarks/preflow_push` in version 2.2.1 and were somewhat disappointed by the included push–relabel implementation. It uses FIFO selection and a global relabeling amortization scheme that tries to mimick the one used by *HIPR* for comparison purposes.

Even in the smallest of our test suite instance, the program did not terminate at all with one or more threads. We assume that it infinitely loops due to some unforseen pattern of active vertices. [Hon08] and [SO13] both demonstrate that it is non-trivial to get the invariants right for an asynchronous implementation and we could not identify any attempt at solving the correctness challenges described and solved in those papers in the Galois source code.

To get an idea of the performance in the case of termination, we had to use another graph instance. On *randLocalGraph_5_500000*, a test graph from the PBBS, we were able to compute the correct flow value using the Galois program.

However, with one thread the program took 20 seconds to find a maximum preflow, while *F_PRF* uses a very similar algorithm (push–relabel with FIFO selection and global relabeling) and finished in under a second. The run time did improve to 12 seconds with two threads, but beyond that we could not get any speedup. We know from other experiments with similar algorithms that the graph has a lot of active vertices at every point during the execution of the algorithm, so the Galois implementation is clearly not optimal. We conjecture that the speculative execution model with rollbacks induces a lot of overhead that makes the approach unfeasible for the application of maximum flow.

5.2.3 Anderson *et al.*'s Algorithm

[AS95] propose an asynchronous parallel algorithm for the maximum flow problem, based on the push–relabel approach. Shared access to vertices is resolved using locks, and an approximate FIFO order of active vertices is maintained using a sophisticated queue data structure, with local queues for fast access and an global queue for work distribution.

It introduces a concurrent global relabeling scheme that can run in a dedicated thread. Every vertex stores additional information about it's relabeling state and the push operation is modified to take this into consideration to prevent pushes that would violate the distance invariant.

We are not aware of a publicly available existing implementation of the algorithm for experimenting. We have tried to come up with an efficient implementation ourselves, based on TBB's work-stealing queues and spinlocks, but performance was bad.

We have done some experiments with *Pmaxflow*, a solver which uses similar ideas but a slightly different implementation. It is available at `https://code.google.com/p/pmaxflow/` and [SO13] describe the approach and implementation. The most notable difference is that it does not use the concurrent global relabeling scheme. Instead it regularly runs a parallel breadth-first search to restore exact labels. The results were not very encouraging: In the cases we tested (including *europe.osm* and the *delaunay* family), the sequential performance was worse even than that of *F_PRF*. With two threads we never saw more than a 20% decrease in runtime and with four threads it was always slower than with one. In some random runs it did not finish at all. The fact that we could not achieve any speedup indicates that maybe we used the tool improperly, but how to fix this is beyond us. We feel that *Pmaxflow* is not a particularly good implementation overall, especially because it uses a global queue, which obviously experiences a lot of contention for the push–relabel algorithm, where a vertex discharge is a very cheap operation.

It might be worth looking at a modern, optimized implementation of the original [AS95] or the lock-free [HH11] algorithm, which uses slightly different semantics, and compare them to our synchronized implementations, described later in this work. We believe that for small graphs, they have an inherent advantage because they can utilize the available parallelism well when only a small number of vertices is active. [AS95] reports speedup even with as few as hundreds of active vertices, while our algorithm usually needs ten times more than that to be efficient (please refer to subsection 7.1.7 for details). It remains to be seen whether this is reproducible on modern architectures where cache coherency overhead and lock contention might be a bigger factor than on 1995 systems.

5.2.4 Bader *et al.*'s Algorithm

Since the main contribution of [BS06] over [AS95] is the use of a priority queue instead of a FIFO queue to approximate the highest-label selection heuristic used by *HIPR*, we do not consider it to be a good candidate for our purposes, seeing as *HIPR* itself performs worse than *F_PRF* on our test suite, which uses FIFO-based selection.

One other key contribution of the paper is their implementation of the gap heuristic: According to the description and the pseudocode, the algorithm maintains a global counter for every possible label from 0 to $n - 1$. Whenever a counter is decreased to zero, a gap has occured and the algorithm stores the gap label in a global variable. Vertices above the gap are then lazily removed the next time they are relabeled.

We see a fundamental correctness problem here: It can happen that after a gap is found, a vertex legitimately gets relabeled from below the gap to above the gap. In such a case, the algorithm described in [BS06] would remove that vertex the next time it gets relabeled, which is incorrect.

Unfortunately the authors were unable to provide us with the implementation they used for their final experiments, so we could neither verify the problem or examine the means by which the algorithm avoids it. Our proposed solution would be to keep a list of gaps, ordered by their detection time. If we store the current length of the list in each vertex after every relabel operation, we can look up the gaps that have been found since the last relabeling of a vertex. This approach would definitely increase the work necessary for the gap heuristic significantly, which makes it harder to achieve absolute speedup over a good sequential implementation that naturally does not have to deal with this issue.

5.2.5 Goldberg *et al.*'s Algorithm

[GT88] already mention a basic synchronized, deterministic version of the FIFO-based push–relabel algorithm. It proceeds in a series of *pulses*, each of which consists of four phases.

1. *phase 1*: All the edges of the graph are tested for admissibility in parallel. Flow is pushed in parallel from active vertices along the admissible edges. However, the excess of the receiving vertices is not immediately increased. Instead the changes are applied to copies of the memory locations representing the vertex excesses.
2. *phase 2*: The new labels of all the vertices that are still active get computed in parallel (but not yet applied).
3. *phase 3*: The new labels are applied in parallel.
4. *phase 4*: The excess changes recorded in phase 1 get applied.

The algorithm terminates as soon as there are no longer any active vertices. Interestingly, we are not aware of any straightforward implementation of this very simple algorithm, so we implemented our own. We made a series of addition to bring it up to date with techniques found in modern push–relabel implementations:

- An amortization scheme is used to trigger global relabeling after a certain amount of work has been performed during the pulses. We use the same scheme as the one used by *HIPR* for comparison purposes.
- The active vertices are maintained in an array, so that in each pulse only the active vertices are actually considered. Newly discovered vertices and vertices that still have excess after phase 1 are stored locally inside an array associated with each vertex. Duplicate additions are avoided through atomic *test-and-set* instructions. At the end of a pulse all the still active and newly activated vertices are concatenated to form the new working set array.
- A very similar algorithm is used to implement parallel global relabeling: A standard parallel breadth-first search is performed by processing the graph layer by layer. Test-and-set is used here as well to prevent duplicate discoveries.

5.2.5.1 Pseudocode

In the following pseudocode and in future sections as well we will use parallel list comprehensions of the form [f(x) | x ← A, P(x)] where f is a function, A is an array and P is a predicate. The result is a new array B consisting of the values of the function f applied to all the elements in A that fulfill the predicate P. Parallel list comprehensions can be implemented using other parallel primitives such as *map*, which maps a function over an array and *filter*, which filters an array using a predicate.

Figure 5.1 shows a pseudocode representation of Goldberg *et al.*'s algorithm, which uses the subroutine shown in Figure 5.2 for global relabeling.

5.2.5.2 Implementation

We integrated *PRSyncDet* into the *Problem Based Benchmark Suite (PBBS)* [SBF+12] maintained by professor Guy Blelloch et al. from Carnegie-Mellon University in Pittsburgh, PA. It is available at `http://www.cs.cmu.edu/~pbbs/`.

The benchmark suite provides an comprehensive and highly-tuned library for parallel primitives like prefix sums, *map*, *filter* and semi-group reducers. It also includes random graph generators that we used a lot for prototyping. PBBS is built to work with both *Cilk* and *OpenMP*, two C++ language extensions to simplify parallel programming. As our main compiler we used G++ 4.8, which supports both extensions.

Cilk uses a thread pool and together with a work-stealing scheduler. It allows the explicit generation of tasks using the cilk_spawn primitive that allows the developer to call a function asynchronously. cilk_sync allows a function to explicitly wait for all the asynchronous tasks it has spawned to finish before execution is continued. A parallel loop implementation is provided by cilk_for , allowing iteration over a range of integers in parallel. This is implemented in a divide-and-conquer fashion: The range is split until it reaches a certain granularity. For each resulting subrange, a task is created. Since cilk_for internally uses the cilk_spawn primitive, nested parallelism is well-supported.

The only OpenMP feature we used is a parallel for loop, similar to cilk_for . The OpenMP implementation included in G++ uses a static loop scheduling algorithm that assumes the loop iterations to be reasonably balanced. In the case where this works, it seems to induce much less overhead than the Cilk approach, so we used it for most of our implementations.

```
1  // Input: G = (V, E), c, s, t
2  PRSyncDet():
3      for all vertices v ∈ V in parallel:
4          d(v) := 0
5          e(v) := 0
6          v.addedExcess := 0
7          v.isDiscovered := 0
8      d(s) := n
9      f(v,w) := 0        for all edges (v, w) ∈ E in parallel
10     // initially saturate all source−adjacent edges
11     for each edge (s, v) ∈ E in parallel:
12         f(s,v) := c(s,v)
13         e(v) := c(s,v)
14     workingSet := [ v | (s, v) ← E, e(v) > 0 ]
15     while workingSet ≠ ∅:
16         if first pulse or work threshold for global update is exceeded:
17             GlobalRelabel()
18             // Label n means that the sink is not reachable from a vertex.
19             // Global relabeling might have identified such vertices
20             workingSet = [ v | v ← workingSet, d(v) < n ]
21
22         for each v ∈ workingSet in parallel:      // phase 1 (push)
23             v.discoveredVertices := []
24             for each residual edge (v, w) ∈ E_f:
25                 if e(v) = 0:   // vertex is already discharged completely
26                     break
27                 if d(v) = d(w) + 1 and c_f(v, w) > 0: // edge is admissible
28                     Δ := min(c_f(v, w), e(v))
29                     f(v,w) += Δ
30                     e(v) −= Δ
31                     w.addedExcess += Δ // atomic fetch−and−add
32                     if w ≠ sink and TestAndSet(w.isDiscovered):
33                         v.discoveredVertices.pushBack(w)
34
35         for each v ∈ workingSet in parallel:      // phase 2 (relabel)
36             if e(v) > 0:   // relabel still active vertex
37                 v.newLabel := n
38                 for each edge (v, w) ∈ E_f with c_f(v, w) > 0:
39                     v.newLabel := min(v.newLabel, d(w))
40                 if TestAndSet(v.isDiscovered):
41                     v.discoveredVertices.pushBack(v)
42             else:
43                 v.newLabel := d(v)
44
45         // phase 3 (apply new labels) + phase 4.1 (apply excess changes for old working set)
46         for each v ∈ workingSet in parallel:
47             d(v) := v.newLabel
48             e(v) += v.addedExcess
49             v.addedExcess := 0
50             v.isDiscovered := 0
51
52         // Build the new working set. The concatenation can be implemented using parallel prefix sums
53         // to find for every list the offset in the final array
54         workingSet := Concat([ v.discoveredVertices | v ← workingSet ])
55         // We filter out vertices of distance n, those are unreachable from the sink in the reverse
56         // residual graph.
57         workingSet = [ v | v ← workingSet, d(v) < n ]
58
59         for each v ∈ workingSet in parallel: // phase 4.2 (apply excess for new working set)
60             e(v) += v.addedExcess
61             v.addedExcess := 0
62             v.isDiscovered := 0
```

Figure 5.1: Pseudocode implementation of Goldberg *et al.*'s parallel maximum flow algorithm

```
1  GlobalRelabel():
2      d(v) := n for all v ∈ V in parallel
3      d(t) := 0
4      Q := [t]
5      while Q ≠ ∅:
6          for each v ∈ Q in parallel :
7              v.discoveredVertices := []
8              for each edge (v, w) ∈ E_f with w ≠ s and c_f(v, w) > 0:
9                  // this branch must be implemented atomically using compare−and−swap
10                 if w ≠ t and d(w) = n:
11                     d(w) := d(v) + 1
12                     v.discoveredVertices.pushBack(w)
13         // concatenation again in parallel, see above
14         Q := Concat([ v.discoveredVertices | v ← Q ])
```

Figure 5.2: Simple global relabeling implementation

5.2.5.3 What we Learned

- The amortization scheme to trigger global relabeling used by *HIPR* is not optimal, because it does not consider the work performed by global relabeling. It might seem like global relabeling does the same amount of work every time it runs (linear in the number of vertices and edges), but that is not the case. In fact the actual work can vary noticably between the runs, depending on the size of the residual subgraph induced by the sink. In a parallel context the effect is even more apparent, because the available parallelism might differ between the normal execution and global relabeling.

  In our experiments, the overall performance was better with an introspective scheme that ensures a certain fraction of the total runtime being spent during global relabeling, relative to the total time. The optimal fraction for most of our test instances was between $\frac{1}{3}$ and $\frac{1}{2}$. The timings reported here will however use the deterministic *HIPR* amortization scheme.

- Initially we thought it would be trivially possible to combine phases 1 and 2 into one, increasing the amount of work per vertex and avoiding one synchronization point. However, as it turns out, the separation is crucial if one wants to keep deterministic behaviour, since the residual capacity of "uphill" edges is dependent on the push phase of the higher of the two adjacent vertices. In chapter 6 we describe our design of a non-deterministic algorithm that works around this issue.

- We experimented with different ways to avoid the separation between phases 1 and 4. One obvious variation is to apply the excess changes directly in phase 1, eliminating phase 4 entirely. This introduces non-determinism and decreases the number of pulses in some cases, but increases it in others. We also tried to combine this with presorting the working set of vertices by decreasing label, in an attempt to get as much excess as possible towards the sink in every pulse.

  Both modifications did not have as significant of an effect as we had hoped for, so for consistency, we sticked with the original, deterministic version when collecting the timings.

- The algorithm can be implemented completely without atomic operations, by recording excess changes induced by every vertex and assigning those in a second pass. However, we found that since the number of pushes is much lower than the number of edge scans, the contention is low on the excess values associated with the vertices.

On the Intel platform we used for testing, a *fetch-and-add* does not have any overhead if there is no contention, so it can be used to synchronize access to these memory locations rather efficiently. This turned out to be more efficient than to record the pushes and group them by target vertex in an additional phase.

- The Cilk language extension is not ideal for this algorithm. It shines when there is a number of reasonably large, possibly imbalanced tasks to be performed. OpenMP does much better for loops with less parallelism, a predictable balancing and little work per iteration, the situation found in most of our test instances. The one exception to this is the *pld_spam* family, where there is a lot of parallelism in every pulse. The Cilk scheduler can provide very good load balancing in this scenario and outperforms OpenMP.
- The *admissible edge data structure* used by *HIPR* – essentially just a pointer to the first edge not known to be admissible or inadmissible – is only marginally useful. The situation where it is effective is when a vertex gets discharged completely and not relabelled at all, but this is less likely to happen for low-degree vertices with small capacities, as often encountered in practice (and in our test suite).
- We experimented with different implementations of parallel breadth-first search such as PBFS by [LS10a], which uses the *bag* data structure. We were not able to achieve overall significantly faster results with those.
- Parallelization on an edge level does not yield good results. One reason for this is that vertex degrees are usually low on average. Also, nested parallelism is not handled gracefully by either Cilk or OpenMP. Cilk supports it but uses the same range splitting threshold even if there is already a lot of tasks created by outer loops.

6. A Semi-Synchronous Push–Relabel Algorithm with Parallel Discharge

While *PRSyncDet* already worked surprisingly well in practice, the restriction that only one relabel operation can be performed per vertex and pulse is problematic. We conjectured that this might have at least two negative consequences:

- The possible parallelism is restricted: A low-degree vertex can only activate so many other vertices before getting relabelled. It would be preferrable to completely *discharge* a vertex in one pulse. What's more, the analysis of the FIFO selection rule assumes a complete discharge of a vertex before the next vertex is processed, so the theoretical bound of $\mathcal{O}(n^3)$ does not hold for our implementation, but the amount of edge scans did not differ significantly between *PRSyncDet* and *F_PRF* in our experiments.
- The per-vertex work is small. This puts an upper bound on the effective speedup because the mechanisms needed to implement the parallelism induce a certain overhead which is harder to amortize if the total amount of work is not particularly high. This is especially a problem if there are not a lot of active vertices to begin with, which is often the case towards the end of the algorithm's execution.

We soon realized that in order to achieve multiple relabels per vertex in one pulse, it would not be possible to stick with the requirement of determinism. The interleaving of relabel operations influences the outcome and makes it necessary to synchronize access to the edges that are shared by two active vertices. We implemented two variants:

- *PRSyncNondetMutex* uses spinlocks to synchronize access to edges shared between active vertices. Figure 6.1 shows a pseudocode representation of the algorithm.
- *PRSyncNondetWin* uses a different approach: The algorithm works on copies d' of the distance labels, so that the original labels can still be read. When looking for admissible edges $(v, w) \in E_f$, the old label of w is considered, while the current label of v is used. The admissibility condition becomes $d'(v) = d(w) + 1$. For an edge $(v, w) \in E$ where both v and w are active, a winning criterion determines which one of two the adjacent vertices can push along the edge in the next pulse: 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 $d(v) = d(w)$).

  Figure 6.2 shows a pseudocode representation of the algorithm.

```
1  // Input: G = (V, E), c, s, t
2  PRSyncNondetMutex():
3      // initialization and saturation of source−adjacent edges as in PRSyncDet Figure 5.1
4      while true:
5          if first pulse
6                  or work threshold for global update is exceeded
7                  or workingSet = ∅
8              GlobalRelabel()  // implementation shown in Figure 5.2
9
10         workingSet = [ v | v ← workingSet, d(v) < n and e(v) > 0 ]
11         if workingSet = ∅:
12             break
13
14         for each v ∈ workingSet in parallel :
15             v.discoveredVertices := []
16             if TestAndSet(v.isDiscovered):
17                 v.discoveredVertices.pushBack(v)
18             while e(v) > 0:
19                 newLabel := n
20                 for each residual edge (v, w) ∈ E_f :
21                     if e(v) = 0: // vertex is already discharged completely
22                         break
23                     Δ := 0
24                     if d(v) = d(w) + 1 and c_f(v, w) > 0: // edge is admissible
25                         with mutex for undirected edge {v,w}:
26                             Δ := min(c_f(v, w), e(v))
27                             f(v,w) += Δ
28                     if Δ > 0:
29                         e(v) −= Δ // atomic fetch−and−add
30                         e(w) += Δ // atomic fetch−and−add
31                         if w ≠ t and TestAndSet(w.isDiscovered):
32                             v.discoveredVertices.pushBack(w)
33                     if c_f(v, w) > 0 , %and d(w) ≥ d(v):
34                         newLabel := min(newLabel, d(w) + 1)
35                 if e(v) = 0:
36                     break
37                 d(v) := newLabel
38                 if d(v) = n:
39                     break
40
41         workingSet := Concat([ v.discoveredVertices | v ← workingSet ])
42         for each v ∈ workingSet in parallel :
43             v.isDiscovered := 0
```

Figure 6.1: Pseudocode implementation of *PRSyncNondetMutex*

Due to deliberate race conditions, both algorithms do not maintain correct labels at all times. Since for a good performance of the algorithm correct labels are crucial, we address this issue by running global relabeling more often. We also run a global relabeling phase if the working set becomes empty, to ensure correctness.

6.1 Implementation

As with the *PRSyncDet* algorithm variant, we integrated *PRSyncNondet** into the *Problem Based Benchmark Suite (PBBS)* [SBF+12] maintained by professor Guy Blelloch et al. from Carnegie-Mellon University in Pittsburgh, PA.

The included library provides the necessary primitives to implement the algorithms in a straightforward manner. We used OpenMP for parallel iteration and TBB's spinlock implementation (tbb::spin_mutex) for the mutexes used by *PRSyncNondetMutex*.

```
1  // Input: G = (V, E), c, s, t
2  PRSyncNondetWin():
3      // initialization and saturation of source−adjacent edges as in PRSyncDet Figure 5.1
4      while true:
5          if first pulse
6                  or work threshold for global update is exceeded
7                  or workingSet = ∅
8              GlobalRelabel()  // implementation shown in Figure 5.2
9              workingSet = [ v | v ← workingSet, d(v) < n ]
10
11         if workingSet = ∅:
12             break
13
14         for each v ∈ workingSet in parallel :
15             v.discoveredVertices := []
16             d'(v) := d(v)
17             e := e(v) // local copy
18             while e > 0:
19                 newLabel := n
20                 skipped := 0
21                 for each residual edge (v, w) ∈ E_f :
22                     if e = 0:   // vertex is already discharged completely
23                         break
24                     admissible := (d'(v) = d(w) + 1)
25                     if e(w):   // is the edge shared between two active vertices ?
26                         win := d(v) = d(w) + 1 or d(v) < d(w) − 1 or (d(v) = d(w) and v < w)
27                         if admissible and not win:
28                             skipped := 1
29                             continue  // skip to next residual edge
30                     if admissible and c_f(v, w) > 0: // edge is admissible
31                         Δ := min(c_f(v, w), e(v))
32                         f(v,w) += Δ
33                         e −= Δ
34                         w.addedExcess += Δ // atomic fetch−and−add
35                         if w ≠ t and TestAndSet(w.isDiscovered):
36                             v.discoveredVertices.pushBack(w)
37                     if c_f(v, w) > 0 and d(w) \geq d'(v):
38                         newLabel := min(newLabel, d(w) + 1)
39                 if e = 0 or skipped:
40                     break
41                 d'(v) := newLabel
42                 if d'(v) = n:
43                     break
44             v.addedExcess := e − e(v)
45             if e'(v) and TestAndSet(v.isDiscovered):
46                 v.discoveredVertices.pushBack(v)
47
48         for each v ∈ workingSet in parallel :
49             d(v) := d'(v)
50             e(v) += v.addedExcess
51             v.addedExcess := 0
52             v.isDiscovered := 0
53
54         workingSet := Concat([ v.discoveredVertices | v ← workingSet ])
55         workingSet = [ v | v ← workingSet, d(v) < n ]
56
57         for each v ∈ workingSet in parallel :
58             e(v) += v.addedExcess
59             v.addedExcess := 0
60             v.isDiscovered := 0
```

Figure 6.2: Pseudocode implementation of *PRSyncNondetWin*

6.2 Unresolved Issues

It seems that *PRSyncNondetMutex* is not correct in its current form. Similar to the Galois implementation (which coincidently uses a similar algorithm), it does not terminate on all instances we tested. We are not sure whether this is due to an implementation bug or a more fundamental problem. Our experiments on other graphs show that it does a similar amount of edge scans as the *PRSyncDet* implementation. However, it does not scale nearly as well with the number of threads (only up to a self-relative speedup of 2 to 3) and the timings vary a lot due to mutex contention.

We instead tried to incorporate the vertex-based locking known from [AS95] with similar results (non-termination in some cases and disappointing timings in other cases). Based on this lack of success, we decided to not persue the idea further and did not bring the version up to a state where it could be included in the final evaluation.

6.3 Techniques to Increase Parallelism

For the proposed parallel push–relabel implementations, the available parallelism (i.e. the average size of the working set) is partly dependent on the number of non-saturating pushes the algorithm makes. The reason for that is that the less excess is pushed across an edge, the more is left on the source vertex to activate other vertices.

Unfortunately in networks with small capacities, which constitute the vast majority of graphs in our test suite, most pushes are saturating, so the observation is not helpful in this case. Of course for other graphs, it might be useful to investigate techniques that help increase the number of non-saturating pushes.

6.3.1 Excess Scaling

[AO89] describe an $\mathcal{O}(nm + n^2 \log U)$ approach to the maximum flow problem, where the edge capacities are integers bounded by U. In the case of unit capacities, this yields an $\mathcal{O}(nm)$ algorithm.

The idea is to maintain an excess upper bound $\Delta = 2^k$ such that $\Delta \geq e(v)$ for all vertices v. Only active vertices with an excess value between $\frac{\Delta}{2}$ and Δ are considered. For an edge $(v, w) \in E_f$, the maximum amount of flow that can be pushed across the edge from v to w is $\Delta - e(w)$. At some point all the vertices will have $e(v) \leq \frac{\Delta}{2}$. If that happens, we set $\Delta := \frac{\Delta}{2}$ and continue.

We implemented this technique in *PRSyncNondetWin*, but with a slight modification: We *do* process active vertices v with an excess $e(v) < \frac{\Delta}{2}$, because otherwise we would give up a lot of the parallelism we have available.

Unfortunately, experiments showed that while the parallelism was increased a bit by this technique in randomly generated graphs with large edge capacities, the algorithm performed more work overall (more pulses and edge scans) and there was no observable speedup overall.

6.3.2 Two-Level Pushes

Inspired by the *push two layers (P2L)* push–relabel variant presented by [Gol09], we implemented a simple heuristic to counter a problematic case where flow is pushed back and forth between two vertices. By maintaining the sum of outgoing residual capacities of each vertex, we can compute an upper bound on the flow pushed along an edge $(v, w) \in E_f$ with $w \neq t$:

$$\sum_{\substack{(w,x)\in E_f \\ x\neq v}} c(w,x) = \left(\sum_{(w,x)\in E_f} c(w,x) \right) - c(x,w)$$

If more than this amount of flow is pushed along the edge, some of it will necessarily return to v at a later point of the algorithm's execution. If at most this amount is pushed, it can certainly be pushed on to a third vertex $x \neq v$.

We also added a modification timestamp to each edge to ensure that flow is pushed to the neighbor from which flow was received the least recently, to prevent pushing flow back to where it came from.

In most of our experiments this optimization had no effect at all because the problematic cases never occurred. This leads us to believe that it is not very useful for a large class of real-world graphs (including those in our test suite) that tend to be very easy to solve.

6.4 A Word about Gap Heuristics

In both *PRSyncDet* and an earlier modification of it similar to *PRSyncNondetWin*, we implemented contention-free gap heuristics. In both cases, while gaps occured occasionally, the overhead of finding them was not offset by the time savings due to removed vertices, and the total runtime was increased in all the tested cases (which did not cover the whole test suite).

This is a bit different from the sequential case, where gap heuristics also do not have a great effect when combined with FIFO selection and global relabeling, but where the cost of finding them is smaller and thus overall the performance can be improved slightly, according to [CG97].

7. Experimental Evaluation

7.1 Test Setup

7.1.1 The Hardware

All of the timings presented in this work were collected on a NUMA Intel machine, to which I was generously provided access by the research group around CMU professor Guy Blelloch. It hosts four Xeon E7-8870 at 2.40GHz per core, making for a total of 40 physical cores and 80 logical cores with hyperthreading. Every processor has 64 GiB of RAM associated with it, for a total of 256 GiB. Due to the nature of the architecture, the memory bank of the processor where a thread is currently running at has a lower latency and higher bandwidth than the other three banks.

The scheduling of allocations is left to the operating system, but in the case of multi-threaded timings we force a certain pattern: Large blocks of memory, such as arrays, are allocated in a striped fashion: The pages are allocated on all four banks alternatingly. In our experiments this helped to get much more consistent timings because the allocation is deterministic, although this approach slightly penalizes executions on less than four cores, which should be kept in mind when interpreting the results. The differences are marginal from what we've seen.

7.1.2 Graph Representations and Comparability

When comparing different algorithms, we tried to use internal representations of the graph as similar as possible to each other. For simplicity, we used 64-bit integers for vertex and edge indices as well as capacities.

The most widely employed representation of the graph in memory seems to be adjacency lists. Outgoing edges for a single vertex are stored contiguously in memory, allowing for cache-efficient scans. Only if the flow on an edge is actually changed, the reverse edge needs to be looked up (via a pointer in the original edge) and modified as well.

One additional optimization is described by [BS06]: Along with the own residual capacity, every edge also stores the residual capacity of its reverse. This speeds up reverse breadth-first searches in the residual network, such as the one employed by the global relabeling heuristic. We used this optimization by default for all our own implementations and added it to *HIPR* as well for fairness.

When timing max-flow algorithms, we usually only measure the time needed to compute a minimum cut, which in the case of preflow–push based implementations is the time to compute a *maximum preflow* (that is, a preflow with no augmenting path from any active vertex to the sink). The time needed to read the graph from disk and build up the internal representation is not included. As mentioned earlier, both *HIPR* and *HPF* use a flow decomposition algorithm to compute the actual flow on each edge only in a second stage. However, for almost all of our real-world test graphs, only the minimum cut is of interest, which can already be reconstructed from the result of the first stage.

One exception is the graph partitioning application: To employ the balanced adaptive flow iterations method, the actual flow function is needed. In our experiments with the KaHIP instances, *HIPR* only spent a very small fraction of its total runtime (usually well below 2%) on the second stage. Simply using *HIPR*'s second stage on top of any other implementation of the first stage is thus a reasonable option in practice. We did not attempt to parallelize the second stage, which could be non-trivial seeing as it is based on a modified depth-first search.

In every experiment we verified the correctness of the resulting flow value manually. In the case of *HIPR*, *HPF* and our own implementations, we also enabled routines that verify the maximality of the final computed flow or preflow function via an augmenting-path search.

7.1.3 The Competition

The different algorithms we ran our test suite on are named as follows:

- **hipr** is the sequential *HIPR* program in its default configuration, with the *GLOB_REL_FREQ* set to 0.5. The parameter regulates the frequency in which global relabelings are run (a higher value means a higher frequency). We modified it to use 64-bit integers and the reverse residual capacity cache optimization mentioned above. It was strictly faster than *HIPR* in all the cases we checked, but only by a small margin.
- **hipr3.0** is **hipr** with *GLOB_REL_FREQ* set to 3.0.
- **hpf** is the sequential *pseudo_fifo* variant of *HPF*. We modified it the same way as *HIPR*.
- **f_prf** is the sequential *F_PRF* in its default configuration. We modified it to work with 64-bit integers, but unfortunately it uses a suboptimal internal graph representation using linked lists instead of arrays, so the runtime could probably be improved by a noticable constant factor (up to 20–30% by our estimation). It also uses a different (and worse) method of determining when to run a global relabeling, but we took it as it is just to demonstrate the surprising point that FIFO-based vertex selection can easily outperform highest-label selection on most of the graphs in our test suite.
- **dtree** is David Eisenstat's sequential dynamic tree-based push–relabel implementation. We modified it to use 64-bit integers.
- **PRSyncDet** is an implementation of the algorithm described in subsection 5.2.5, with *GLOB_REL_FREQ* set to 3.0. This value yielded overall good results for our test suite.
- **PRSyncNondetWin** is an implementation of the algorithm described in chapter 6, with *GLOB_REL_FREQ* set to 3.0.

For more details about the sequential implementations and their respective sources, please refer to subsection 5.2.1.

graph name	num. vertices	num. edges	max. capacity
genrmf_wide_4	1,048,576	5,160,960	10000
washington_rlg_wide_16	4,194,306	12,517,376	30000
delaunay_24	10,066,330	60,378,613	1
delaunay_25	20,132,660	120,768,109	1
delaunay_26	40,265,322	241,552,557	1
delaunay_27	80,530,640	483,126,599	1
delaunay_28	161,061,274	966,286,764	1
delaunay_28_hier4	17,290,574	100,463,233	31
rgg_24	10,066,332	159,076,553	1
rgg_25	20,132,661	331,386,145	1
rgg_26	40,265,320	689,424,569	1
rgg_27	80,530,639	1,431,907,505	1
grid_28_hier4	11,754,212	47,006,759	4
europe.osm	15,273,606	32,521,077	1
nlpkkt240	8,398,082	222,847,493	1
pld_spam_100	42,889,802	623,056,513	1
pld_spam_5000	42,889,802	623,066,313	1
BL06-camel-lrg	18,900,002	93,749,846	16000

Table 7.1: Properties of our benchmark graph instances. The maximum edge capacity is excluding source or sink adjacent edges

7.1.4 The Benchmarks

The different graph families in our test suite have been described in detail in subsection 5.1.4. Table 7.1 lists the graphs with their precise vertex and edge counts.

7.1.5 Testing methodology

We ran every solver multiple times on every instance, each time measuring the time to find a maximum preflow after the graph representation has been built up in memory. For the deterministic algorithm variants **hipr**, **hpf**, **f_prf**, **dtree** and **PRSyncDet**, we averaged the timings over at least three different runs. For **PRSyncNondet**, we performed at least five runs because the variance is higher due to a non-deterministic number of pulses.

7.1.6 Results

A comprehensive matrix of all the sequential test results can be found in Table 7.2. As mentioned above, *HPF* seems to be the overall best performer, but *PRSyncDet* with one thread often achieves timings within a factor of two. Exceptions are the DIMACS graphs, the vision instance and the three smallest KaHIP instances. On the spam detection benchmarks, *HPF* is outperformed by the others.

We chose the global relabeling frequency for *PRSyncDet* in an attempt to get the best performance on our test suite. To show that this does not give an unfair advantage over *HIPR*, we showed the influence of the parameter using the *hipr3.0* configuration. In all but one cases, increasing the frequency yields worse results in the case of *HIPR*. Overall, on our test suite, *HIPR* is inferior to both *HPF* and *F_PRF*. We conclude that FIFO-based selection is superior to highest-label selection for the type of applications we tried to capture in our test suite.

For our test instances, dynamic trees do not improve the actual run time of the push–relabel algorithm. This is shown by the timings from the *dtree* implementation.

Table 7.3 and Table 7.4 show how the *PRSyncDet* and *PRSyncNondetWin* algorithm variants scale with the available number of threads. We can see that on the largest *delaunay* and *rgg* instances, on the spam detection instances as well as on the *nlpkkt240*, *BL06-camel-lrg* and *washington_rlg_wide_16* graphs, self-relative speedups of over 8 could be achieved with 16 threads. In most cases, the difference between 16 and 32 threads is small, indicating either a lack of available parallelism or possibly that main memory becomes the bottleneck beyond a certain point of concurrency. In some cases the runtime even increases when switching from 16 to 32 threads, an effect that is probably due to a suboptimal use of OpenMP.

For all but the smallest instance of the *delaunay* and *rgg* family, two threads are sufficient to beat the best sequential solution time. For *nlpkkt240* it takes four threads and for *washington_rlg_wide_16* as well as *BL06-camel-lrg* it takes eight. *europe.osm*, *genrmf_wide_4* and *grid_28_hier4* exhibit too little average active vertices per pulse to be parallelized efficiently using our method. Figure 7.1 and Figure 7.2 show how *PRSyncDet* scales on the *rgg_27* graph, compared to *HPF*, the solver achieving the best sequential time.

We want to point out especially the promising results for the *pld_spam* family, which we specifically designed to be a practical application of a minimum cut technique. An absolute speedup over *HIPR* of almost 12 is achieved with 32 threads, as can be seen in Figure 7.4. These graphs have a large number of active vertices on average in each pulse, which we attribute to the small diameter of the graph. This allows for very efficient parallelization. For comparison purposes, the numbers presented here are measured with OpenMP, but even better timings are achieved in this particular case with Cilk (*pld_spam100* is solved by *PRSyncNondetWin* in under 30 seconds with 32 threads and Cilk scheduling).

For the *delaunay* and *rgg* families, the self-relative and absolute speedups are better for the larger instances, again because there are more vertices active in each pulse on average.

On the vision instance *BL06-camel-lrg*, it takes 8 threads for *PRSyncNondetWin* to beat the performance of *HPF*. Nevertheless, with 32 threads we still achieve an absolute speedup of almost 4, due to the near-perfect linear scaling on this instance.

Our hope was that the modifications introduced in *PRSyncNondetWin* allow for more effective parallelization. This has turned out to be partly successful, seeing as for all but one of the tested graphs, the best average time achieved by *PRSyncNondetWin* is at least as good as the best average time achieved by *PRSyncDet*. Sometimes the difference is very noticable, especially on the medium-sized KaHIP instances: in the case of *delaunay_26*, the difference is almost a minute, or 30% of the total time. What's notable is that with only one thread, *PRSyncDet* is often faster, so the self-relative speedups of *PRSyncNondetWin* are slightly better than those of *PRSyncDet*. Overall, the difference is not significant in most cases. We would thus recommend the conceptionally simpler *PRSyncDet* variant in practice, especially if only a small number of threads is available.

7.1.7 Speedup as a Function of the Number of Active Vertices

Intuition tells us that for our synchronized algorithm implementations, the achievable speedup will be better the more active vertices there are on average in each pulse. This assumption was confirmed by our experiments. For the *delaunay_28* graph, Figure 7.5 shows the self-relative speedup with regard to the number of active vertices in a pulse, with 8 threads enabled. We can see that decent speedups of over 4 are achieved when there are more than 3000 active vertices. Below that, speedups are much worse. Below 1000 active vertices, the synchronization overhead becomes too much to get any speedup at all. Of course the exact thresholds are dependent on the vertex degrees (and thus the

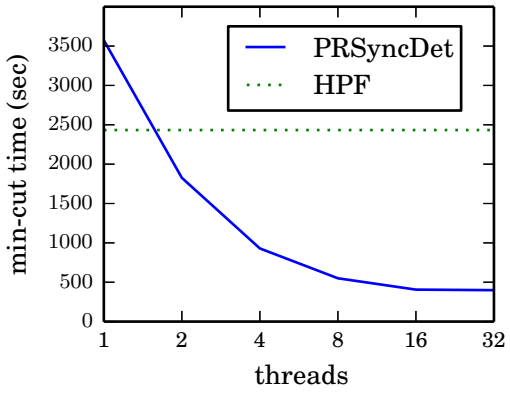

Figure 7.1: Timings for *rgg_27*

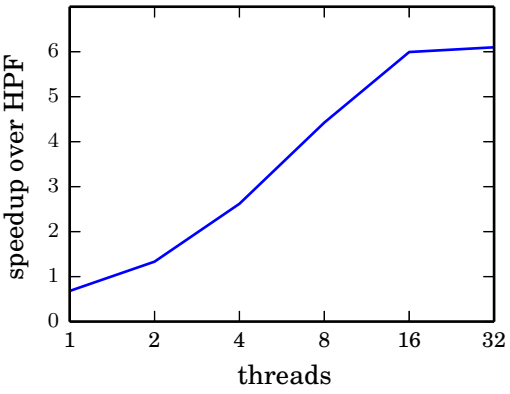

Figure 7.2: Speedup for *rgg_27*

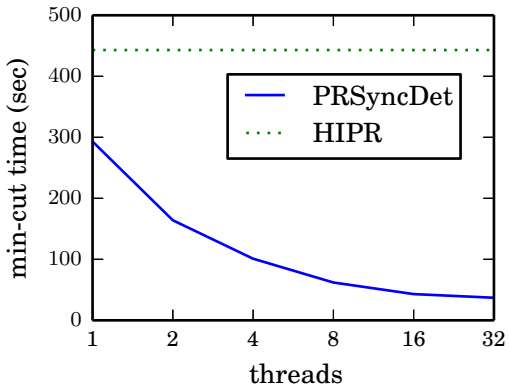

Figure 7.3: Timings for *pld_spam_100*

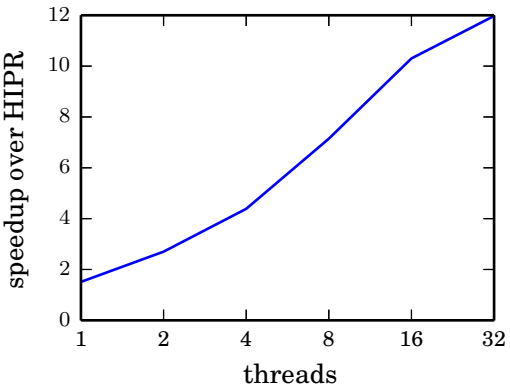

Figure 7.4: Speedup for *pld_spam_5000*

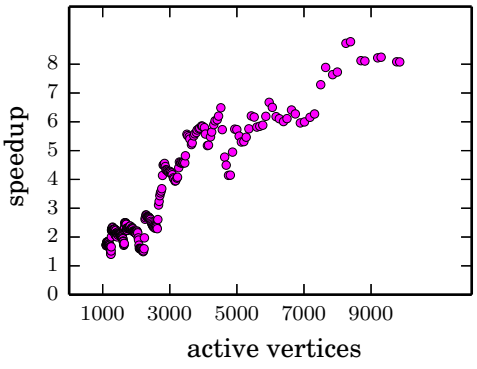

Figure 7.5: *PRSyncNondetWin* Self-relative speedup on *delaunay_28* with 8 threads, depending on number of active vertices. The times exclude global relabeling.

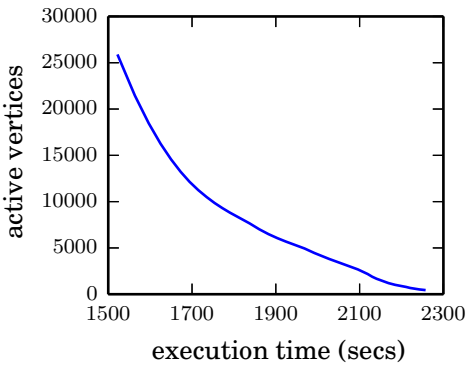

Figure 7.6: *PRSyncNondetWin* active vertices during the course of execution on *delaunay_28* with 1 thread. The times exclude global relabeling.

amount of work per vertex), but these numbers seem to be consistent across our benchmark instances.

Figure 7.6 shows the number of active vertices during the course of the algorithm's execution in the case of the *delaunay_28* graph. We can see that in this case, only during a small fraction of the total runtime (about 200 seconds) the number of active vertices drops below the threshold of 3000. This tail is hard to parallelize and the reason for the non-linear speedup seen on this graph.

graph	hipr	hipr3.0	hpf	f_prf	dtree	PRSyncDet	PRSyncNondetWin
genrmf_wide_4	41	104	**11**	81	49	275	260
washington_rlg_wide_16	88	68	**26**	118	66	168	194
delaunay_24	361	664	**77**	194	372	154	152
delaunay_25	1193	1680	**208**	494	1021	312	316
delaunay_26	2502	4142	**562**	1028	2452	826	804
delaunay_27	8032	13,308	**1180**	2648	9073	1715	1780
delaunay_28	21,564	35,008	**2905**	8124	*	4359	4665
delaunay_28_hier4	965	1757	**269**	449	976	539	568
rgg_24	643	1108	259	355	521	257	**245**
rgg_25	1468	2575	543	980	1245	**536**	573
rgg_26	4737	6648	**1338**	3321	3985	1363	1403
rgg_27	13,082	19,419	**2433**	6937	*	3574	3807
grid_28_hier4	1015	1649	**64**	372	594	125	119
europe.osm	359	445	**22**	76	372	59	46
nlpkkt240	521	1119	**218**	711	1418	609	752
pld_spam_100	443	508	907	405	1475	**293**	405
pld_spam_5000	486	514	614	444	*	**443**	709
BL06-camel-lrg	259	491	**56**	220	**	317	358

Table 7.2: Sequential benchmark results. The numbers represent the time in seconds to find a maximum preflow (excluding initialization time). Timings are averaged over multiple runs. We skipped the runs marked with (*) because the results would not have been particular interesting and our computation time on the machine was limited. The run marked with (**) raised an error because there are multiple edges between the same vertices in this instance, which *dtree* cannot handle. For each instance, the best timeing is presented underlined.

39

graph	number of threads					
	1	**2**	**4**	**8**	**16**	**32**
genrmf_wide_4	275	195 (1.4)	134 (2.1)	106 (2.6)	114 (2.4)	181 (1.5)
washington_rlg_wide_16	168	84 (2.0)	41 (4.1)	22 (7.6)	13 (12.9)	10 (16.8)
delaunay_24	154	86 (1.8)	57 (2.7)	43 (3.6)	40 (3.9)	56 (2.8)
delaunay_25	312	184 (1.7)	111 (2.8)	81 (3.9)	75 (4.2)	94 (3.3)
delaunay_26	826	438 (1.9)	282 (2.9)	203 (4.1)	191 (4.3)	233 (3.5)
delaunay_27	1715	979 (1.8)	592 (2.9)	411 (4.2)	357 (4.8)	404 (4.2)
delaunay_28	4359	2082 (2.1)	1065 (4.1)	616 (7.1)	478 (9.1)	574 (7.6)
delaunay_28_hier4	539	285 (1.9)	162 (3.3)	110 (4.9)	100 (5.4)	124 (4.3)
rgg_24	257	145 (1.8)	89 (2.9)	62 (4.1)	55 (4.7)	62 (4.1)
rgg_25	536	312 (1.7)	177 (3.0)	114 (4.7)	96 (5.6)	114 (4.7)
rgg_26	1363	683 (2.0)	347 (3.9)	196 (7.0)	137 (9.9)	154 (8.9)
rgg_27	3574	1826 (2.0)	929 (3.8)	550 (6.5)	406 (8.8)	399 (9.0)
grid_28_hier4	125	105 (1.2)	95 (1.3)	89 (1.4)	93 (1.3)	119 (1.1)
europe.osm	59	44 (1.3)	35 (1.7)	32 (1.8)	34 (1.7)	45 (1.3)
nlpkkt240	609	340 (1.8)	170 (3.6)	93 (6.5)	56 (10.9)	46 (13.2)
pld_spam_100	293	164 (1.8)	101 (2.9)	62 (4.7)	43 (6.8)	37 (7.9)
pld_spam_5000	443	305 (1.5)	170 (2.6)	96 (4.6)	60 (7.4)	49 (9.0)
BL06-camel-lrg	317	156 (2.0)	77 (4.1)	44 (7.2)	29 (10.9)	29 (10.9)

Table 7.3: Parallel benchmark results for *PRSyncDet*. The numbers outside parantheses represent the time in seconds to find a maximum preflow (excluding initialization time). The number in parens represents the self-relative speedup over the run with only one thread. Timings are averaged over multiple runs.

graph	number of threads					
	1	**2**	**4**	**8**	**16**	**32**
genrmf_wide_4	260	202 (1.3)	139 (1.9)	102 (2.5)	93 (2.8)	109 (2.4)
washington_rlg_wide_16	194	98 (2.0)	48 (4.0)	24 (8.1)	14 (13.9)	9 (21.6)
delaunay_24	152	90 (1.7)	55 (2.8)	41 (3.7)	38 (4.0)	40 (3.8)
delaunay_25	316	182 (1.7)	99 (3.2)	70 (4.5)	52 (6.1)	54 (5.9)
delaunay_26	804	420 (1.9)	241 (3.3)	154 (5.2)	133 (6.0)	152 (5.3)
delaunay_27	1780	920 (1.9)	526 (3.4)	320 (5.6)	253 (7.0)	251 (7.1)
delaunay_28	4665	2151 (2.2)	1180 (4.0)	619 (7.5)	560 (8.3)	493 (9.5)
delaunay_28_hier4	568	290 (2.0)	169 (3.4)	98 (5.8)	80 (7.1)	85 (6.7)
rgg_24	245	131 (1.9)	78 (3.1)	49 (5.0)	35 (7.0)	35 (7.0)
rgg_25	573	320 (1.8)	182 (3.1)	111 (5.2)	90 (6.4)	86 (6.7)
rgg_26	1403	680 (2.1)	345 (4.1)	189 (7.4)	128 (11.0)	105 (13.4)
rgg_27	3807	1970 (1.9)	951 (4.0)	503 (7.6)	362 (10.5)	358 (10.6)
grid_28_hier4	119	101 (1.2)	80 (1.5)	72 (1.7)	74 (1.6)	83 (1.4)
europe.osm	46	37 (1.2)	28 (1.6)	23 (2.0)	22 (2.1)	26 (1.8)
nlpkkt240	752	388 (1.9)	189 (4.0)	101 (7.4)	60 (12.5)	38 (19.8)
pld_spam_100	405	233 (1.7)	135 (3.0)	84 (4.8)	52 (7.8)	37 (10.9)
pld_spam_5000	709	395 (1.8)	218 (3.3)	123 (5.8)	78 (9.1)	46 (15.4)
BL06-camel-lrg	358	172 (2.1)	87 (4.1)	46 (7.8)	25 (14.3)	15 (23.9)

Table 7.4: Parallel benchmark results for *PRSyncNondetWin*. The numbers outside parantheses represent the time in seconds to find a maximum preflow (excluding initialization time). The number in parens represents the self-relative speedup over the run with only one thread. Timings are averaged over multiple runs.

8. Future Work

There are some ideas and problems leftover that we have not followed up on during the research for this thesis due to time constraints. We will document those here in an unordered manner and might investigate them in the future.

8.1 Improvements for *PRSyncDet*

A straightforward observation is that due to the simplicity of the *PRSyncDet* algorithm variant presented in subsection 5.2.5, it lends itself to a MapReduce-based distributed implementation similar to the Ford–Fulkerson implementation from [HYW11]. This might make it possible to solve even larger graphs than the ones we experimented with, where the instances cannot be held in main memory completely. We expect this to yield worse total runtimes due to communication overhead.

With regard to the algorithm itself, we feel that it would be worth looking at the kind of bidirectional approach used by the BK algorithm and [HYW11], where partial augmenting paths are found forward from the source and backward from the sink simultaneously. The analogue in the push–relabel world is to introduce additional labels that approximate the distance to the source and potentially to also allow for node deficits. The pseudoflow algorithm takes this but approach but uses quite complicated data structures that might be hard to parallelize efficiently. We feel that a simpler implementation can still improve our simpler push–relabel algorithm and help increase the number of active vertices to work with.

We noted in subsection 7.1.7 that even for large graphs, a considerable part of the algorithm's execution time is spent in a phase where there is just not enough active vertices for the synchronized approach to be really efficient. It might be worth incorporating the [AS95] or [HH11] approach into our algorithm and swtich to it when the number of active vertices drops beyond a certain threshold. We believe these to be less efficient when many vertices are active but potentially more efficient on small graphs or in the late stages of the algorithm on large graphs.

8.2 Flow Decomposition

Since in most applications only a minimum cut is interesting, the explicit flow assignment function is often not needed. If it is, however, our algorithm performs the second stage of transforming a maximum preflow into a maximum flow sequentially.

The algorithm used by *HIPR* and *HPF* is basically a greedy depth-first search to push the excess flow back to the source. It might be possible to use a modified breadth-first search to replace it, which would be easier to parallelize. We have not looked into the details so far.

9. Conclusion

In this work, we presented a comprehensive overview of current applications of and solutions to the maximum flow problem. We re-evaluated the performance of existing sequential algorithms on a test suite meant to represent real-world applications and were able to show that the often used *HIPR* program is not necessarily the best choice when working with these kinds of graphs. It has been shown in earlier work to outperform most other solvers on the graph families represented in the DIMACS maximum flow challenge, but this superiority does not extend to the range of instances we tested.

Where an implementation was available, we measured the performance of existing parallel solvers and dismissed most of them because they were barely able to outperform efficient sequential implementations.

Based on our observation that push–relabel with FIFO-based selection is a very simple and also efficient solution to the max-flow problem, we chose it as the basis of a first parallel implementation. Seeing as this prototype yielded very promising results, we added non-determinism to increase the work between synchronization points, in an attempt to amortize the relative overhead of synchronization barries and to allow for better overall parallelism. This was only partly successful, the difference in run time is not as significant as we had hoped.

Where applicable, we analyzed the effectiveness of different techniques like gap heuristics and excess scaling. Our experiments indicate that vanilla push–relabel without gap heuristics constitutes a very good compromise between simplicity and performance.

Our final implementation is able to achieve almost linear relative speedups on some graph families, and respectable relative speedups in other cases. Absolute speedups are good, in a lot of cases only two processors are needed to achieve the speed of the best sequential solver. On larger graphs we tend to get better speedups, which we consider to be promising because it shows our algorithms scale well with the size of the data. More often than not, the time to initialize the graph representation in memory, even when massively parallelized, represents a considerable fraction of the total runtime and puts an upper bound on the overall performance.

We believe to have shown that parallel maximum flow is not as hard a problem as was conjectured in previous work, if applied to modern, large, real-world graph families as opposed to small, artificial benchmark instances. We further believe that our implementations are of real practical value in many areas where large maximum flow or minimum cut problems

arise as subproblems, for example in graph partitioning and social network analysis. The small-worldness property of a large web graph has turned out to be especially favourable for our parallel approach.

The deterministic algorithm variant presented in subsection 5.2.5 is a surprisingly simple, synchronized parallel algorithm performing well in practice and could be used as an example in academic teaching.

Bibliography

[AO89] R. K. Ahuja and J. B. Orlin, "A Fast and Simple Algorithm for the Maximum Flow Problem," *Operations Research*, vol. 37, pp. 748–759, 1989.

[AS95] R. Anderson and J. Setubal, "A Parallel Implementation of the Push–Relabel Algorithm for the Maximum Flow Problem," *Journal of Parallel and Distributed Computing*, 1995.

[BK04] Y. Boykov and V. Kolmogorov, "An Experimental Comparison of Min-Cut/Max-Flow Algorithms for Energy Minimization in Vision," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 26, no. 9, pp. 1124–37, Sep. 2004.

[BMSW13] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, ser. Contemporary Mathematics, vol. 588. American Mathematical Society, 2013.

[BS06] 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," 2006.

[CG97] B. V. Cherkassky and A. V. Goldberg, "On Implementing Push–Relabel Method for the Maximum Flow Problem," *Algorithmica*, vol. 19, no. 4, pp. 390–410, Dec. 1997.

[CH09] B. G. Chandran and D. S. Hochbaum, "A Computational Study of the Pseudoflow and Push–Relabel Algorithms for the Maximum Flow Problem," *Operations Research*, vol. 57, no. 2, pp. 358–376, Mar. 2009.

[DB08] A. Delong and Y. Boykov, "A Scalable Graph-Cut Algorithm for N-D Grids," in *IEEE Conference on Computer Vision and Pattern Recognition, 2008*, June 2008, pp. 1–8.

[DG08] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Communications of the ACM*, pp. 137–149, 2008.

[Din06] Y. Dinitz, "Theoretical computer science," O. Goldreich, A. L. Rosenberg, and A. L. Selman, Eds. Berlin, Heidelberg: Springer-Verlag, 2006, ch. Dinitz' Algorithm: The Original Version and Even's Version, pp. 218–240.

[DM89] U. Derigs and W. Meier, "Implementing Goldberg's Max-Flow-Algorithm – A Computational Investigation," vol. 33, no. 6, pp. 383–403, 1989.

[FF62] L. Ford and D. Fulkerson, *Flows in Networks*, 1962.

[FHM10] B. Fishbain, D. Hochbaum, and S. Mueller, "Competitive Analysis of Minimum-Cut Maximum Flow Algorithms in Vision Problems," *arXiv preprint arXiv:1007.4531*, pp. 1–16, 2010.

[FHM13] B. Fishbain, D. S. Hochbaum, and S. Mueller, "A Competitive Study of the Pseudoflow Algorithm for the Minimum s–t Cut Problem in Vision Applications," *Journal of Real-Time Image Processing*, Apr. 2013.

[FLG00] G. W. Flake, S. Lawrence, and C. L. Giles, "Efficient Identification of Web Communities," in *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '00. New York, NY, USA: ACM, 2000, pp. 150–160.

[GG87] D. Goldfarb and M. D. Grigoriadis, *A Computational Comparison of the Dinic and Network Simplex Methods for Maximum Flow*, 1987.

[GHK+11] A. V. Goldberg, S. Hed, H. Kaplan, R. E. Tarjan, and R. F. Werneck, "Maximum Flows by Incremental Breadth-first Search," in *Proceedings of the 19th European Conference on Algorithms*, ser. ESA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 457–468.

[Gol09] A. Goldberg, "Two-level Push–Relabel Algorithm for the Maximum Flow Problem," *Algorithmic Aspects in Information and Management*, 2009.

[GSS82] L. Goldschlager, R. Shaw, and J. Staples, "The Maximum Flow Problem is Log Space Complete for P," *Theoretical Computer Science*, vol. 21, 1982.

[GT88] A. Goldberg and R. Tarjan, "A New Approach to the Maximum-Flow Problem," *Journal of the ACM (JACM)*, vol. 35, no. 4, pp. 921–940, 1988.

[HH10] Z. He and B. Hong, "Dynamically Tuned Push–Relabel Algorithm for the Maximum Flow Problem on CPU-GPU-Hybrid Platforms," *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1–10, Apr. 2010.

[HH11] B. Hong and Z. He, "An Asynchronous Multithreaded Algorithm for the Maximum Network Flow Problem with Nonblocking Global Relabeling Heuristic," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, pp. 1025–1033, June 2011.

[Hoc08] D. S. Hochbaum, "The Pseudoflow Algorithm: A New Algorithm for the Maximum-Flow Problem," *Operations Research*, vol. 56, no. 4, pp. 992–1009, Aug. 2008.

[Hon08] B. Hong, "A Lock-Free Multi-Threaded Algorithm for the Maximum Flow Problem," *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, Apr. 2008.

[HW08] M. Haklay and P. Weber, "Openstreetmap: User-generated street maps," *Pervasive Computing, IEEE*, vol. 7, no. 4, pp. 12–18, Oct 2008.

[HYW11] F. Halim, R. Yap, and Y. Wu, "A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs," in *2011 31st International Conference on Distributed Computing Systems (ICDCS)*, June 2011, pp. 192–202.

[JM93] D. S. Johnson and C. C. McGeoch, Eds., *Network Flows and Matching: First DIMACS Implementation Challenge*. Boston, MA, USA: American Mathematical Society, 1993.

[JW98] J. Jiang and L. Wu, "A MPI Parallel Algorithm for the Maximum Flow Problem," 1998.

[LS10a] C. E. Leiserson and T. B. Schardl, "A Work-efficient Parallel Breadth-first Search Algorithm (or How to Cope with the Nondeterminism of Reducers)," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10. New York, NY, USA: ACM, 2010, pp. 303–314.

[LS10b] J. Liu and J. Sun, "Parallel Graph-Cuts by Adaptive Bottom-Up Merging," in *2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2010, pp. 2181–2188.

[MVLB14] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "Graph Structure in the Web — Revisited: A Trick of the Heavy Tail," in *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*, ser. WWW Companion '14. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2014, pp. 427–432.

[PMLP+11] K. Pingali, M. Méndez-Lojo, D. Prountzos, X. Sui, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, and R. Manevich, "The Tao of Parallelism in Algorithms," *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '11*, p. 12, 2011.

[SBF+12] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief Announcement: The Problem Based Benchmark Suite," in *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '12. New York, NY, USA: ACM, 2012, pp. 68–70.

[SH13] A. Shekhovtsov and V. Hlaváč, "A Distributed Mincut/Maxflow Algorithm Combining Path Augmentation and Push–Relabel," *International Journal of Computer Vision*, p. 40, Sep. 2013.

[Sib97] J. Sibeyn, "The Parallel Maxflow Problem Is Easy for Almost All Graphs," vol. 3075, no. 3075, pp. 1–16, 1997.

[SK10] P. Strandmark and F. Kahl, "Parallel and Distributed Graph Cuts by Dual Decomposition," in *2010 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2010, pp. 2085–2092.

[SO13] S. Soner and C. Ozturan, "Experiences with Parallel Multi-threaded Network Maximum Flow Algorithm," pp. 1–10, 2013.

[SS11] P. Sanders and C. Schulz, "Engineering Multilevel Graph Partitioning Algorithms," *Algorithms–ESA 2011*, 2011.

[SS13] ——, "Think Locally, Act Globally: Highly Balanced Graph Partitioning," in *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, ser. LNCS, vol. 7933. Springer, 2013, pp. 164–175.

[STKA07] H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara, "A Large-Scale Study of Link Spam Detection by Graph Algorithms," *Proceedings of the 3rd international workshop on Adversarial information retrieval on the web - AIRWeb '07*, p. 45, 2007.

[VB12] T. Verma and D. Batra, "MaxFlow Revisited: An Empirical Comparison of Maxflow Algorithms for Dense Vision Problems." *BMVC*, 2012.

[YKGF06] H. Yu, M. Kaminsky, P. B. Gibbons, and A. Flaxman, "SybilGuard: Defending Against Sybil Attacks via Social Networks," *SIGCOMM Computer Communication Review*, vol. 36, no. 4, pp. 267–278, Aug. 2006.