A Parallel Push-Relabel Maximum Flow Algorithm in LAGraph and GraphBLAS

Darin A. Peries*, Timothy A. Davis†
Dept. of Computer Science and Engineering, Texas A&M University, College Station, TX
Email: vincent2013@tamu.edu*, davis@tamu.edu †

***Abstract*—Maximum flow algorithms hold significant importance in various industries, including communication networks, transportation and logistics, and more. For example, they can find supply chain limitations and identify the maximum number of data packets supported by a network. However, the computation of flow from source to sink is computationally intensive and hard to parallelize. The Push-Relabel algorithm was introduced by Goldberg and Tarjan for better parallelism by pushing flow in a wave-like pattern. We implemented this algorithm in LAGraph using parallel sparse matrix operations provided by GraphBLAS. The resulting method is an elegant and efficient implementation of the Push-Relabel algorithm.**

***Index Terms*—Maximum Flow, Push-Relabel, GraphBLAS, Graph Algorithms, Sparse Matrices, Linear Algebra**

I. Introduction

The Maximum Flow algorithm is a fundamental technique in combinatorial optimization and graph theory. It is used in a variety of industries and research areas such as communication networks, travel and logistics, image segmentation, and network analysis.

A Maximum Flow algorithm typically treats each edge weight of a graph as a capacity and determines the amount of flow that can be pushed throughout the network from a source node to a sink node without exceeding any of the edge capacities. The residual capacity of an edge is defined as $residual := capacity - flow$, which is the amount of additional flow that can be pushed through that edge before reaching its maximum capacity. The residual is used to determine how much flow can be pushed in each iteration of the algorithm. Whenever an edge (u, v) has flow pushed through it to a node v, an edge with residual capacity is added from node v back to u. This property allows any flow pushed forward to be pushed back to the source. In addition, most Maximum Flow algorithms follow the law of flow conservation, which states that the amount of flow entering the node must be equal to the amount exiting the node. By iterating on these principles, an algorithm can calculate the maximum flow from source to sink.

The Push-Relabel algorithm uses two concepts, labels and preflow, to systematically push flow through the network of nodes and edge-capacities. This algorithm differs from most conventional algorithms by temporarily violating the conservation of flow through the concept of preflow. The preflow concept refers to a state in which a node has more units of flow entering than exiting and is treated as an intermediate state in the algorithm. The algorithm is initialized by having the neighbors of the source saturated with preflow. As the algorithm executes, the flow is distributed both forward and backward across the network until there is no node with any preflow left apart from the source and sink nodes. The label concept in the algorithm serves as a control mechanism that dictates which nodes can receive more units of flow. A node u can only push to a node v when $d(u) = d(v)+1$, where the function d returns the height of the node. The terms *height* and *label* are synonyms in the literature for Push-Relabel algorithms, so we use both terms. They are identical.

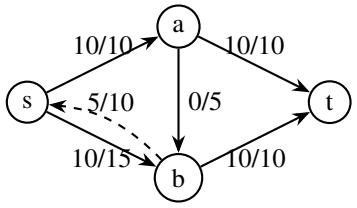

Fig. 1. Example of a Flow Graph with source s and sink t

Additionally, the preflow-push algorithm was made to be easily parallelized compared to other Maximum Flow algorithms. In Baumstark et al. [2], the authors devised a modern and efficient parallel Push-Relabel algorithm, which made use of coarse-grained synchronization and multiple atomic operations [10]. Their algorithm has a wave-like pattern, where the flow is distributed to multiple nodes in the levels of the network, similar to how a parallel breadth-first search would span out from the source node.

By viewing this pattern, the implementation in Baumstark et al. can be expressed mathematically using parallel matrix computations in the SuiteSparse:GraphBLAS library [8], [9], an open source linear algebra library specialized in solving graph algorithms by following the GraphBLAS standard specification [5]. GraphBLAS achieves this through the adjacency matrix of a graph, the abstract concept of a semiring, and the use of sparse matrix algorithms.

II. The Push-Relabel Algorithm

The Push-Relabel algorithm can be seen as a level-by-level push of flow to the sink (Algorithm 1). This incremental movement of flow requires a sense of direction, which the previously described height mechanism provides.

Flow is distributed from an active node, a node with excess flow, v to any node with a height $d(v) - 1$, ensuring the

algorithm doesn't exit early by pushing to the source or neglecting possible pushes. Only when there are no other possible nodes to push to, the active node will relabel its height to $min(neighbors) + 1$ and then push flow to a node with a valid label. These actions are repeated until there are no more nodes with excess flow. We use the notation d and e to be consistent with Goldberg and Tarjan [10].

Algorithm 1 Push-Relabel Outline

```
Init: d_s = |V|, f_{s,u} = c_{s,u}, e_u = c_{s,u}, others 0
while ∃u ≠ s,t : e_u > 0 do
    if ∃v : d_u = d_v + 1 ∧ r_{u,v} > 0 then
        δ ← min(e_u, r_{u,v})
        f_{u,v} += δ; f_{v,u} −= δ
        e_u −= δ; e_v += δ
    else
        d_u ← 1 + min{d_v | r_{u,v} > 0}
    end if
end while
return Σ_u f_{s,u}
```

III. GraphBLAS and Semirings

SuiteSparse:GraphBLAS [8], [9] is a mathematical software library that implements the GraphBLAS standard [5], [6]. It contains a standard set of matrix and vector operations to solve graph algorithms by allowing a user to define the two binary operations of a semiring: the additive monoid $\oplus$, and the multiplicative binary operator $\otimes$.

In addition to vector and matrix operations, GraphBLAS also uses algebraic masks, accumulators, and descriptors. A mask can either be a vector or a matrix that determines which elements in the resulting algebraic object are affected by the operation. A mask M can be used structurally (denoted as $C\langle \text{struct}(M)\rangle \leftarrow ...$) where only the positions of its entries are used, or its values can be used by typecasting them to Boolean (denoted as $C\langle M\rangle \leftarrow ...$). Accumulators are used as an extra option in any GraphBLAS operation. Lastly, the descriptor is used to modify the behavior of the operation with respect to the mask, inputs, and outputs. For example, the expression $C\langle M\rangle \leftarrow accum(C,\ A * B')$ computes the matrix multiplication $Z \leftarrow A * B'$, the $accum(C,\ Z)$ represents the expression $C \leftarrow C \oplus Z$ for an optional binary accumulator operator $\oplus$, and the mask M allows $C(i,j)$ to be revised if $M(i,j)$ is true; $C(i,j)$ cannot be revised if otherwise. The operator $*$ denotes matrix multiplication with the semiring.

GraphBLAS also has element-wise operations defined over unary and binary operators, which are functions that operate on one or two inputs respectively. It has a variety of pre-built operators, but it also allows the user to create their own user-defined binary and unary operators, which allows the matrix computations to be customized for a variety of use cases. Through the combinations of these operations and matrix/vector/scalar objects, the GraphBLAS API can be used to solve a graph algorithm, where graphs are represented as sparse adjacency matrices.

SuiteSparse:GraphBLAS has its own run-time JIT built into the library, which can compile GraphBLAS operations using arbitrary user-defined types and operators. This is essential for performance, since our implementation makes extensive use of these objects. We use the following methods in GraphBLAS:

- `GrB_apply`: applies a unary or binary operator to all elements of a vector or matrix.
- `GrB_select`: prunes entries from a matrix or vector, keeping or deleting the entry a_{ij} based on the result of a binary operator $f(a_{ij}, i, j, \theta)$, for some scalar θ.
- `GrB_assign`: assigns a vector or matrix to another matrix or vector.
- `GrB_mxv`: a matrix-vector multiply.
- `GrB_eWiseMult`: applies a binary operator to two inputs, returning the set intersection.
- `GrB_eWiseAdd`: applies a binary operator to two inputs, returning the set union.
- `GxB_eWiseUnion`: like `GrB_eWiseAdd`, except that the binary operator is applied to all entries.

IV. The LAGraph Maximum Flow algorithm

The LAGraph Push-Relabel method is described in Algorithm 8. It implements the method of Baumstark, et al. [2], but using GraphBLAS instead of custom parallel methods. In this section, we describe each component of the algorithm and how it is implemented using GraphBLAS. The algorithm is very simple, consisting entirely of calls to GraphBLAS, with a single loop to handle the primary outer iteration. No other loops are needed. All parallelism is handled inside GraphBLAS itself, with no OpenMP pragmas needed in the LAGraph Maximum Flow algorithm itself.

A. Initialization

The Push-Relabel algorithm in [2], [10] starts with the source-adjacent nodes with preflow equal to the capacity of each out-going edge. We represent the per-node preflow as a sparse double precision vector e of size n. In addition, the labels of each node are represented as a 64-bit or 32-bit integer vector. The input adjacency matrix A (with non-negative weights) is transformed into a matrix R (short for residual) through a series of masked-assign and matrix-apply operations. The entry $R(i,j)$ is set to a tuple, $(capacity, flow)$, which is a user-defined data type. If the edge $A(i,j)$ exists, then this tuple holds $(A(i,j), 0)$, denoting no flow, but with a capacity of $A(i,j)$. If the corresponding edge $A(j,i)$ does not exist, then $R(j,i)$ is set to $(0,0)$. Thus, while A can be unsymmetric, the structure of entries in R is always symmetric. The *structure* of a matrix in GraphBLAS denotes where explicit entries appear. When the flow is revised in R during the algorithm, the structure of R does not change; only the flow components of its tuples change.

During the maximum flow algorithm, the $flow$ component of the tuples can become negative. For example, suppose $A(i,j) = 5$ and $A(j,i) = 3$. This means that up to 5 units can flow from i to j, or up to 3 units can flow from j to i (but not at the same time). Initially, $R(i,j) = (5,0)$ and

$R(j,i) = (3,0)$. If one unit of flow is pushed from i to j, these values become $R(i,j) = (5,1)$ and $R(j,i) = (3,-1)$, respectively.

This initialization first uses `GrB_apply` on the input adjacency matrix. This operation takes each edge value of A and inputs each value into a unary operation. In addition, for each edge $A(i,j)$ there must be an edge $R(j,i)$ created if the edge $A(j,i)$ does not exist. This is achieved by transposing the matrix A and setting the transposed edges in R to the tuple $(0,0)$. To merge these two matrices, a structural complemented-masked assignment is used, where the mask is the original adjacency matrix. The complemented mask ensures that any preexisting edges do not have their values overwritten by the transposed version of A.

The node labels/heights are represented as a dense vector, d, where $d(src) \leftarrow n$ and the other entries are set to zero. We then revise this with a global relabeling using Algorithm 6.

Throughout the execution of the Push-Relabel algorithm, there exists an active set $e = \{e \subset V, preflow(e) > 0\}$. We store this set in a `GrB_Vector` and name it the e vector. We initialize this vector by a `GrB_extract` operation, where we take the column src of matrix A (for the source node) and assign it to e, but exclude any nodes not reachable in A' from the $sink$, as determined by the initial global relabeling.

Lastly, we need to take the values in e and assign them to the row of R, where the values of e are assigned to the flow entry of the tuple. The operation takes the form of $R(i,0).flow \leftarrow e_i$ through a binary accumulator in the assign operation. This sets all source-adjacent edges to their proper values, with the initial preflow from the source node.

B. Iterations

The main loop of the algorithm is divided into four parts: (1) an optional global relabeling, (2) deciding where to push, (3) verifying the pushes, and (4) executing the pushes. These are done through a series of matrix builds, multiplies, and element-wise operations.

1) Part 1: global relabel heuristic: This optional heuristic is described in Section IV-C. We use this heuristic early on, but describe it last, since it is best understood if the remainder of the algorithm is presented first.

2) Part 2: deciding where to push: In the algorithm developed by Baumstark et al. [2], a node i with excess flow can only push to an adjacent node j along an edge (i,j) with positive residual capacity. After determining its adjacent nodes with positive residual capacity, node i must decide which neighbor node j to push its flow to. This decision is based on the height of the nodes i and j, the node ids (for handling ties), and the residual capacity of the edge (i,j).

When mapping to a matrix operation, this decision becomes a masked matrix-vector multiply $push\langle \text{struct}(e), replace\rangle \leftarrow R * d$, where the vector d is dense and the matrix R is sparse, and $push$ is a vector with $push(i)$ containing the best candidate push from node i. The mask e ensures that only active nodes can push to one of their neighbors; if node i is not active, $push_i$ is empty. The semiring for $R*d$ is defined by the additive monoid shown in Algorithm 2 (which selects the best neighbor j for node i to push to), and the multiplicative operator shown in Algorithm 3 (which determines if the edge (i,j) has any residual capacity).

Algorithm 2 Additive monoid $\oplus$ for the $R * d$ semiring

```
Inputs: x and y, both of type result_tuple
Output: z of type result_tuple
► return z as the best of x and y
if x.d < y.d then
    z ← x                                  ▷ take tuple with smaller d
else if x.d > y.d then
    z ← y
else                                       ▷ tuples x and y have the same height d
    if x.residual > y.residual then
        z ← x                              ▷ take tuple with larger residual
    else if x.residual < y.residual then
        z ← y
    else                                   ▷ tie: take tuple with the larger node id
        if x.j > y.j then z ← x else z ← y
    end if
end if
```

Algorithm 3 Multiplicative operator $\otimes$ for the $R * d$ semiring

```
Inputs: x (which is R_{i,j}), y (which is d_j), and j
Output: z of type result_tuple
residual ← x.capacity − x.flow
if residual > 0 then
    z.residual ← residual
    z.d ← d_j
    z.j ← j
else
    z ← (0, ∞, −1)                         ▷ the empty tuple
end if
```

The result $push\langle ... \rangle \leftarrow R * d$ has type `result_tuple`, a vector of tuples in the form $(residual, d_j, j)$. For the entry $push_i$, $residual$ is the residual capacity of the edge (i,j), d_j is the height of node j, and j is the node id j. If no flow can be pushed from node i, then y_i is set to an "empty tuple," with the values $(0, \infty, -1)$. Empty tuples are pruned from $push$ using a `GrB_select` operation. Some of the candidate pushes may be invalid; these are removed in Part 3 of the algorithm.

3) Part 3: verifying the pushes: This step decides which of the tentative pushes in the $push$ vector are valid. We create a new data type that contains the height of the two nodes involved in the push operation as well as the residual capacity and j index. This is achieved using a `GrB_eWiseMult` operation on the $push$ and d vectors, where we define the binary operation to combine the inputs into a 4-tuple of the form: $z \leftarrow (residual, d_i, d_j, j)$, where the entry $y_i = (residual, d_j, j)$ and d_i comes from the d vector. The resulting vector is called yd. Its sole purpose is to construct a matrix C with all of the candidate pushes (C for Candidate) using `GrB_Matrix_build`. The entry $C(i,j)$ is defined as the

tuple $(residual, d_i, d_j, j)$, if y_i contains the candidate push of node i to j (namely, the tuple $(residual, d_j, j)$), and d_j is the height/label of node j. This matrix, C, contains all the edges that flow can be pushed upon, as selected in Part 2.

We make e dense via $e\langle\neg\text{struct}(e)\rangle = 0$, and then compute $push \leftarrow C * e$ on a semiring defined by the additive monoid and multiplicative operator in Algorithms 4 and 5. The multiplicative operator of the semiring compares the heights, excess flows, residual capacities, and positions and determines which edges will have flow pushed upon them to ensure no conflicts happen between pushes. The multiplicative operator (Algorithm 5) includes the rules used by Baumstark et al. to determine which pushes are valid [2].

Algorithm 4 Additive monoid $\oplus$ for the $C * e$ semiring

Inputs: x and y, both of type `result_tuple`
Output: y of type `result_tuple`
▷ This is not a true monoid since is not commutative. However, it is not used since each row of C has at most one entry.
$z \leftarrow y$

Algorithm 5 Multiplicative operator $\otimes$ for $C * e$ semiring

Inputs: x (which is Map_{ij}) and y (which is e_j)
Output: z of type `result_tuple`
$jactive \leftarrow (y > 0)$
if $(x.d_i < x.d_j - 1)$
$\vee\ (x.d_i = x.d_j - 1 \wedge \neg jactive)$
$\vee\ (x.d_i = x.d_j \wedge (\neg jactive \vee (jactive \wedge i < j)))$
$\vee\ (x.d_i = x.d_j + 1)$ **then**
  $z \leftarrow (x.residual, x.d_j, x.j)$ ▷ node i can push to j
else
  $z \leftarrow (0, \infty, -1)$ ▷ the empty tuple; i cannot push to j
end if

The $push$ vector now contains all validated pushes. Pushes which may cause conflict are replaced with empty tuples, which are pruned from the $push$ vector via a call to `GrB_select`. The relabel operation is done via call to `GrB_eWiseMult`, with d and $push$, to revise the height/label vector d.

In the original paper presented by Goldberg and Tarjan [10], there is a height invariant that must be maintained between pushes. It states that if flow is pushed along edge (v, w), then $d(v) = d(w)+1$ must hold, which means that the height of the node must be one more than the height of the node it is pushing flow to. Our algorithm always maintains this invariant (some Push-Relabel methods temporarily violate this invariant).

4) Part 4: executing the pushes: To calculate the amount of flow pushed for an iteration, the vector e is put through a min operation with the vector e and the residual values from $push$, with a call to `GrB_eWiseMult`. The operation yields the δ vector. This vector, in turn, is used to create the Δ matrix, where $push.j$, $\delta.i$, and the values of δ are passed to the `GrB_Matrix_build` operation. From there, we duplicate the matrix and subtract it by its transpose. This gives us the final Δ matrix. We then use a masked `GrB_eWiseMult` with a user-defined binary operator in the form $R(i,j).flow$+=$\Delta(i,j)$ to update the R matrix.

The changes to the preflow resulting from Δ are found by summing each row of Δ, resulting in another δ vector. The vector e is then updated with δ via a masked assignment, to obtain the revised preflow of each node. The maximum flow is augmented (using Algorithm 7 with the preflow that has reached the sink, and then a `GrB_select` operation removes any explicit zeros in the e vector to get a new set of active nodes with excess flow, preserving the e vector's sparse nature. The algorithm terminates if the e vector is empty.

C. Global relabel heuristic

An optional optimization step can be applied intermittently throughout the execution of the algorithm. In the original Push-Relabel paper by Goldberg et al. [10], the authors noted an important observation about the performance of the algorithm. When the node labels accurately reflect the distance to the sink node, the algorithm terminates much earlier. Cherkassky et al. [7] expand on this idea and describe two heuristics to improve the algorithm: the global and gap relabel heuristics. Baumstark et al. [2] do not use the gap relabel heuristic, since it has little impact on the performance of their algorithm. They use the global relabeling heuristic only. We do likewise.

The global relabel heuristic involves running a breadth-first search algorithm starting from the sink node. The heuristic reverses the edges and ignores any saturated edges. The result is a graph labeled with the distance to the sink for each node. Any node u that cannot reach the sink is labeled with the value $d(u) = n$, where n is the number of nodes in the graph. This ensures that node u is removed from any further computation and creates optimized paths to the sink for the flow to traverse.

The global relabeling in Algorithm 6 uses LAGraph's pre-existing Breadth-First Search (BFS) algorithm [12]. It uses push-pull optimizations to achieve a significant performance increase [4]. The BFS computes a level vector with the distance of each node from the starting node.

Algorithm 6 Global Relabel algorithm

Input: matrix R, vector mask z with just src and $sink$
Input/Output: vector d
Output: vector lvl
► **function definition**
$GetResidual(R_{ij}) = R_{ij}.capacity - R_{ij}.flow$
► **construct $\widehat{R}$ and perform a BFS from $sink$**
$\widehat{R} \leftarrow GetResidual(R)$ using `GrB_apply`
remove saturated edges from $\widehat{R}$ (with $\widehat{R}_{ij} \leq 0$)
$lvl \leftarrow$ breadth-first search on graph of $\widehat{R}'$ starting at $sink$
► **assign level accurate heights**
$d\langle\neg\text{struct}(z)\rangle \leftarrow lvl$ with `GrB_assign`
► **assign any unreachable nodes the height of n**
$d\langle\neg\text{struct}(lvl)\rangle \leftarrow n$ with `GrB_assign`

The matrix $\widehat{R}$ is constructed from R, extracting the residual flow of each edge. `GrB_apply` is used with a user-defined binary operator ($GetResidual$) to compute $\widehat{R} = R.capacity - R.flow$. Then, using `GrB_select`, we delete any saturated edges from $\widehat{R}$. A saturated edge is any edge v whose flow is equal to the capacity. Passing $\widehat{R}'$ into the LAGraph BFS algorithm, we get a vector that has all the distances of each node to the sink. The last step is to assign these updated distances to the d vector with two calls to `GrB_assign`.

The global relabel algorithm is costly, so it is used only during the initializations and every 12 iterations thereafter. The costly part is not the LAGraph BFS itself, but the transpose required to compute the graph it operates on.

Once the nodes have been relabeled, we filter out any disconnected nodes. This is defined in [2] as any nodes that can no longer reach the sink from the source by following non-saturated edges in R. The remainder of the Push-Relabel iterations are sped up by ignoring any disconnected nodes.

D. Flow value and Flow graph

The primary result of our algorithm is the total maximum flow from source to sink as a single scalar. If this is the only result required, the global relabel optimization can be modified to remove any disconnected nodes from the active set for a better runtime. However, our algorithm can optionally return the flow graph as well, which is an entire graph with edge (u, v) containing the flow from node u to v to obtain the maximum flow from the source node to the sink node.

When computing the maximum flow using the global relabeling heuristic, the algorithm may terminate while leaving preflow on nodes that cannot be reached by the sink from a breadth-first search of the transposed graph. This results in having an incorrect flow graph being returned, as the law of flow conservation is not followed for each node.

To return a valid flow graph, this excess flow must eventually be pushed back to the source. If the flow graph is a required output of our algorithm, the removal of unreachable nodes from the excess vector e is skipped in Algorithm 8. The flow graph can then be constructed by extracting the flows from the R matrix when the iterations finish. This allows our algorithm to compute a valid flow graph, at the cost of extra time, which can be substantial. We are exploring alternative methods for computing the flow graph.

V. Performance Results

Table I lists the graphs used in our benchmarks. For each problem, we selected a source and sink node such that there was at least some possible flow from source to sink (node ids are zero-based). We compare our performance with the parallel Push-Relabel maximum flow method in Galois [13], a high-performance parallel graph library not based on GraphBLAS. Galois does not have an option to return the flow graph, so our LAGraph benchmarks exclude this option.

We tested our method on three of the GAP matrices [1], [3] and three matrices from the SNAP dataset [11] on a system with an Intel Xeon E5-2695 v2 (2.40GHz), 12 physical cores, one socket, 24 threads, and 3/4 TB of RAM. Table II shows the timings for running the Galois [13] and our method on the three largest graphs: GAP-twitter, GAP-web, and GAP-road. Galois failed on the GAP-road matrix. Our method is about two to six times faster than Galois. We also benchmarked our method on a system with an Intel Xeon 8352Y (2.2GHz), 64 physical cores, two sockets, 128 threads, and 256GB of RAM. Galois ran out of memory on this system for the larger matrices, so it is excluded from Table III.

TABLE I
GRAPH BENCHMARK DATA

Graph name	**# edges**	**# nodes**	**source**	**sink**
GAP-Twitter	1468.4M	61.6M	13	17
GAP-Web	1930.3M	50.6M	64712	500
GAP-road	57.7M	23.9M	4	1075
com-Youtube	6.0M	1.1M	4	275
com-LiveJournal	69.4M	4.0M	2	88
com-Orkut	234.4M	3.1M	43	75

TABLE II
RUNTIME (SECONDS) OF LAGRAPH AND GALOIS WITH DIFFERENT # OF THREADS, ON A SYSTEM WITH 12 HARDWARE CORES AND 3/4TB RAM

Method and graph	4 threads	8 thr.	12 thr.	24 thr.
Galois GAP-twitter	3749.6	2627.8	1438.8	1803.3
Galois GAP-web	997.1	709.9	699.9	546.7
Galois GAP-road	-	-	-	-
LAGraph GAP-twitter	1792.6	944.4	782.2	769.0
LAGraph GAP-web	334.9	194.4	108.6	86.5
LAGraph GAP-road	1059.9	831.7	899.1	298.6
Galois/LAGraph relative run times (higher is better for LAGraph):				
GAP-twitter	2.1	2.8	1.8	2.3
GAP-web	3.0	3.7	6.4	6.3

TABLE III
RUNTIME (SECONDS) OF LAGRAPH WITH DIFFERENT # OF THREADS ON A SYSTEM WITH 64 HARDWARE CORES AND 256GB OF RAM

Graph	4 threads	16 thr.	32 thr.	64 thr.	128 thr.
GAP-twitter	379.78	108.67	62.24	40.11	37.96
GAP-web	204.43	59.97	40.88	36.70	31.10
GAP-road	108.13	70.48	63.98	84.49	70.64
com-Youtube	0.56	0.23	0.19	0.22	0.28
com-LiveJournal	3.11	2.60	0.51	0.68	0.63
com-Orkut	35.60	9.55	5.23	4.59	4.46

VI. Conclusions

The maximum flow from source to sink in a graph is expensive to compute. The Push-Relabel method [2], [10] is able to compute the maximum flow in parallel. Its implementation in GraphBLAS is a simple and elegant way to express this algorithm without the use of any compiler or hardware directives. All operations of the algorithm are computed with high-performance parallel sparse matrix operations implemented by SuiteSparse:GraphBLAS. We maintain the height invariant introduced in [10], where each push can only occur from node u to v where $d(u) = d(v) + 1$. This verifies the correctness of our algorithm and shows that it upholds the basic principles outlined in that paper.

The primary motivation for our work lies in expressing a parallel Push-Relabel algorithm through simple matrix operations while maintaining reasonable performance. Our work demonstrates that such an algorithm can be expressed without knowledge of advanced C/C++ hardware primitives. To obtain high performance, we rely on the GraphBLAS JIT to compile kernels; the operations used by our method would be much slower in earlier versions of GraphBLAS. Without the JIT, each call to an operator would require a call through a function pointer. Our method achieves high performance and is 2 to 6 times faster than the parallel Push-Relabel method in Galois.

Our code is in LAGraph v1.2.1 on GitHub[1]. Performance results were obtained with GraphBLAS v10.1.0.[2]

VII. Acknowledgments

This work was supported by MIT Lincoln Lab & FalkorDB.

References

[1] A. Azad et al., "Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite," 2020 IEEE Intl. Symp. Workload Characterization (IISWC), Beijing, China, 2020, pp. 216-227.

[2] N. Baumstark, G. E. Blelloch, and J. Shun, "Efficient Implementation of a Synchronous Parallel Push-Relabel Algorithm." Algorithms - ESA 2015. Lecture Notes in CS, vol 9294. Springer, Berlin.

[3] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," vol. arXiv: 1508.03619, 2015.

[4] –, "Direction-optimizing Breadth-First Search," SC12: Proc. Intl. Conf. High Performance Computing, Networking, Storage and Analysis, Salt Lake City, 2012, pp. 1-10.

[5] B. Brock, A. Buluç, T. Mattson, S. McMillan, and J. E. Moreira. "The GraphBLAS C API specification," v2.0, 2021. http://graphblas.org/.

[6] A. Buluç, T. Mattson, S. McMillan, J. E. Moreira, and C. Yang. "Design of the GraphBLAS API for C." IEEE Intl. Parallel and Distributed Processing Symp., Orlando, pages 643–652, 2017.

[7] B. V. Cherkassky and A. V. Goldberg, "On implementing Push-Relabel method for the maximum flow problem," Integer Programming and Combinatorial Optimization, pp. 157–171, Springer Berlin, 1995.

[8] T. A. Davis, "Algorithm 1000: SuiteSparse:GraphBLAS: graph algorithms in the language of sparse linear algebra," ACM Trans. Mathematical Software, vol 45, no 4, Dec. 2019.

[9] T. A. Davis, "Algorithm 1037: SuiteSparse:GraphBLAS: Parallel graph algorithms in the language of sparse linear algebra," vol. 49, ACM Trans. Mathematical Software, Sept. 2023.

[10] A. V. Goldberg and R. E. Tarjan, "A new approach to the maximum flow problem," in Proc. 18th Annual ACM Symp. Theory of Computing, STOC '86, p. 136–146, ACM, 1986.

[11] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection." http://snap.stanford.edu/data, 2014.

[12] T. Mattson, T. A. Davis, M. Kumar, A. Buluç, S. McMillan, J. Moreira, and C. Yang, "LAGraph: A community effort to collect graph algorithms built on top of the GraphBLAS," 2019 IEEE Intl. Parallel & Distributed Processing Symp. Workshops (IPDPSW), pp. 276–284, 2019.

[13] K. Pingali, "High-speed graph analytics with the Galois system," in Proc. 1st Workshop on Parallel Programming for Analytics Applications, PPAA '14, p. 41–42, ACM, 2014.

Algorithm 7 Augment the maximum flow

Inputs: src and $sink$, mask vector z
Input/Output: maximum flow f, excess vector e
$f \leftarrow f + e(sink)$ ▷ augment the maximum flow
► prune zeros from e, and delete $e(src)$ and $e(sink)$
$e\langle\neg\text{struct}(z), replace\rangle \leftarrow$ `select`(e where $e > 0$)

[1]https://github.com/GraphBLAS/LAGraph

[2]https://github.com/DrTimothyAldenDavis/GraphBLAS

Algorithm 8 LAGraph Maximum Flow

Inputs: adjacency matrix A and its transpose; src and $sink$
Output: maximum flow f
► **function definitions**
$ResidualForward(A_{ij}) = (A_{ij}, 0)$
$ResidualBackward(A_{ij}) = (0, 0)$
$MakeFlow(e_i) = (0, e_i)$
$InitForw(x, y) = (x.capacity, x.flow + y.flow)$
$InitBack(x, y) = (x.capacity, x.flow - y.flow)$
$ResidualFlow(y_i) = y_i.residual$
$UpdateFlow(R_{ij}, \Delta_{ij}) = (R_{ij}.capacity, R_{ij}.flow + \Delta_{ij})$
$Relabel(d_i, push_i) = push.d_i + 1$ if $d_i < push.d_i + 1$, else d_i
► **initializations**
$f \leftarrow 0$ ▷ initial maximum flow
$d \leftarrow$ all zero vector of size n, $d(src) \leftarrow n$
► create R matrix, where $R_{ij} = (capacity, flow)$
$R \leftarrow ResidualForward(A)$ with `GrB_apply`
$R\langle A\rangle \leftarrow ResidualBackward(A')$ with `GrB_apply`
initial global relabeling (Algorithm 6)
► initialize the excess flow, e
$e\langle\text{struct}(lvl)\rangle \leftarrow A(src, :)'$
► push preflow to all neighbors of src
$t = MakeFlow(e)$ with `GrB_apply`
$R(src, :) \leftarrow InitForw(R(src, :), t')$ with `GrB_assign`
$R(:, src) \leftarrow InitBack(R(:, src), t)$ with `GrB_assign`
$z \leftarrow$ empty vector of size n, $z(src) = 1$, $z(sink) = 1$
augment the maximum flow (Algorithm 7)
► **compute the maximum flow**
while e is not empty **do**
  ► **Part 1: global relabeling**
  if every 12th iteration **then**
    global relabeling (Algorithm 6)
    ▷ remove unreachable nodes from excess vector e
    $e\langle\neg\text{struct}(lvl)\rangle \leftarrow$ (empty) with `GrB_assign`
  end if
  ► **Part 2: deciding where to push**
  $push\langle\text{struct}(e), replace\rangle \leftarrow R * d$ with `GrB_mxv`
  prune empty tuples from $push$ with `GrB_select`
  ► **Part 3: verifying the pushes**
  create C matrix from structure & values of $push$ and d
  ▷ pad e with zeros to ensure it is dense
  $e\langle\neg\text{struct}(e)\rangle = 0$ with `GrB_assign`
  $push \leftarrow C * e$ with `GrB_mxv`
  prune empty tuples from $push$ with `GrB_select`
  $d\langle\text{struct}(push)\rangle \leftarrow Relabel(d, push)$,
    with `GrB_eWiseMult`
  ► **Part 4: executing the pushes**
  $\delta \leftarrow ResidualFlow(push)$ with `GrB_apply`
  $\delta \leftarrow \min(\delta, e)$ with `GrB_eWiseMult`
  create Δ matrix from structure and values of δ and $push$
  $\Delta \leftarrow \Delta - \Delta'$ using `GxB_eWiseUnion`
  $R\langle\text{struct}(\Delta)\rangle \leftarrow UpdateFlow(R, \Delta)$
  $\delta \leftarrow sum(\Delta')$ with `GrB_reduce`
  $e\langle\text{struct}(\delta)\rangle \mathrel{+}= \delta$ with `GrB_assign`
  augment the maximum flow (Algorithm 7)
end while