Multicore Max-Flow using GraphBLAS: A Usability Study

Zawadi Svela and Anne C. Elster

Norwegian University of Science and Technology

Abstract. Optimizing linear algebra operations has been a research topic for decades. The compact language of mathematics also produce lean, maintainable code. Using linear algebra as a high-level abstraction for graph operations is therefore very attractive.
In this work, we will explore the usability of the GraphBLAS framework, currently the leading standard for graph operations that uses linear algebra as an abstraction. We analyze the usability of GraphBLAS by using it to implement the Edmonds-Karp algorithm for s-t maximum-flow/minimum-cut. To our knowledge, this work represents the first published results of Max-Flow in GraphBLAS. The result of our novel implementation was an algorithm that achieved a speedup of up to 11 over its own baseline, and is surprisingly compact and easy to reason about.

Keywords: Graph Algorithms · Frameworks · HPC · Irregular Algorithms · Linear Algebra

1 Introduction

Graphs are one of the most flexible and most used tools to model data, being flexible enough to model any set of relations between objects. They can represent objects as diverse as building structures, social networks, electrical circuits or road networks. As the sizes of the data processed keeps going up, so does the need for processing power keep increasing. The solution to this has been to increasingly utilize parallel multi-core systems.

Utilizing large systems can be very complex though, as parallel computing comes with unique challenges not present in sequential computing. This is particularly difficult in the case of graph algorithms, as they tend to be data-driven and highly irregular [1], that is, the amount and pattern of work cannot be determined before the algorithm starts. This means sophisticated techniques are needed to divide the work between the computational resources, and to safeguard against unwanted states caused by race conditions.

In order to alleviate the need for users to implement advanced techniques themselves, developers and researchers have made high-level frameworks and libraries. These let the user more generally express the flow of their algorithms. They then don't need to handle threads or implement objects that support parallel access directly.

Recent years have seen a growing interest and development in expressing graphs and graph problems in the language of linear algebra, particularly in the form of the GraphBLAS standard [2]. Utilizing linear algebra has a few promising advantages. One is the consistent mathematical concepts that help reasoning about algorithms, for example identifying multiple mathematically equivalent ways of expressing the same procedure, and reason about which would yield the better performance. Processing linear algebra is also a field that has seen extensive research throughout the years, and expressing graphs in this way unlocks a direct way of utilizing decades worth of research and optimizations.

1.1 Goals and Contributions

The goal of this work is to test the usability and performance of the GraphBLAS standard as a framework for massively parallel graph algorithms. We define usability as the combination of productivity and maintainability, in other words, how easy it is to utilize the framework. The central questions are whether building implementations on GraphBLAS can achieve sufficient performance for the overhead of learning, how quickly new algorithms can be made once learned, and how maintainable the resulting code is.

This is achieved by implementing the Edmonds-Karp algorithm for maximum flow. This algorithm is complex enough that it allowed us to explore large parts of the GraphBLAS tool-set, while being composed of parts that were known to be expressible in terms of linear algebra operations.

2 Background

We define a graph G = (V,E) as a set of vertices $V = \{v_1, v_2, ..., v_n\}$ and edges $E = \{e_1, e_2, ..., e_m\}$. Each edge $e_k = (i, j)$ is a connection from vertex v_i to v_j. If a graph is undirected, then $(i, j) \in E \Rightarrow (j, i) \in E$, otherwise it is a directed graph. Edges can be weighted.

Most real world graphs exhibit a high degree of sparsity, the number of edges $E << V^2$, the upper limit. This means sparse matrix formats like CSR/CSC are utilized for storage. The graphs are also often scale-free, with high variance in degree, which along with sparsity contributes to graph algorithm's irregularity.

Frontier Throughout this paper, we will use the concept of the frontier as short-hand for vertices that are part of a bulk parallel operation. This concept is used under different names in a number of different graph frameworks, like Gunrock[9] and Ligra[8].

2.1 Graphs as linear algebra / GraphBLAS

GraphBLAS is a standard developed by the GraphBLAS Forum[1] for expressing graphs and graph algorithms in terms of linear algebra constructs and operations. The key observation is that if graphs are represented as (sparse) adjacency

[1] https://graphblas.github.io/

matrices, which is already typical, then we can express operations on them in the language of linear algebra. This technique is particularly good at expressing large bulk-operations, which is useful when one wishes to parallelize the algorithms in multi-core computers or even GPUs.

As an example to illustrate this, we will look at the breadth-first search (BFS), and we will focus on the parent search. That is, for each vertex, we want to note its parents in the search. This version of the BFS is both in a sense the most general and illustrates the expressiveness of GraphBLAS.

Our search will be one level at a time, and given a adjacency matrix A and a vector x representing the current level, the *frontier*, we want to both determine the vertices in the next level outward and note which vertex "discovered" them.

$$x'_i = \sum_{x_j \in x, A_{ij} \in A} x_j * A_{ij} \tag{1}$$

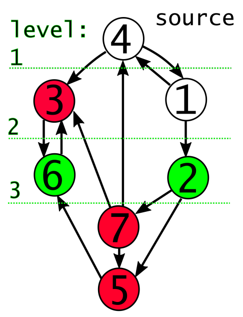
Equation 1 shows the calculation for a given value $x'_i \in x'$ for the vector matrix multiplication $x' = xA$. Already, the individual non-zero multiplications correspond to the edges and the summations correspond to the reduction if a vertex has multiple candidate parents in the search. What remains is simply to change the operators used in order to express traversal rather than arithmetic. We change the multiplicative operator to FIRSTJ, extracting the parent index, and the addition to ANY, as the choice of parent is arbitrary.

The result can be seen in Listing 1.1 and graphically in Figure 1.

Fig. 1: A single step of a breadth-first search as a vector-matrix multiplication, using a (FIRSTJ, ANY)-semiring.

(a) Graph. Figure provided by Professor Tim Davis, author of Suitesparse:GraphBLAS

(b) Linear Algebra operation



$$\boxed{. \;\; 1 \;\; . \;\; . \;\; . \;\; 3 \;\; .} * \begin{array}{c} . \;\; 1 \;\; . \;\; 1 \;\; . \;\; . \;\; . \\ \boxed{. \;\; . \;\; . \;\; . \;\; 1 \;\; . \;\; 1} \\ . \;\; . \;\; . \;\; . \;\; . \;\; 1 \;\; . \\ 1 \;\; . \;\; 1 \;\; . \;\; . \;\; . \;\; . \\ . \;\; . \;\; . \;\; . \;\; . \;\; 1 \;\; . \\ \boxed{. \;\; . \;\; 1 \;\; . \;\; . \;\; . \;\; .} \\ . \;\; . \;\; 1 \;\; 1 \;\; 1 \;\; . \;\; . \\ = \\ \boxed{. \;\; . \;\; 6 \;\; . \;\; 2 \;\; . \;\; 2} \end{array}$$

In GraphBLAS, this changing of operator (and sometimes domain) is done through an algebraic structure called a semiring. By simply changing the semiring used, we can express many algorithms using the same structure as Listing

Listing 1.1: Outline of a breadth-first search in GraphBLAS

```
parent<s> = s
frontier<s> = s
while (frontier not empty):
        frontier<parent'> = x * A
        parent<x> = x
```

1.1. Examples of some semirings and the calculation they infer can be seen in table 1.

Table 1: Semirings and resulting primitive
FIRSTJ: J-index of the first argument
ANY: Any one of the two values

$\bigotimes$	$\bigoplus$	Domain	Primitive
$*$	$+$	$\mathbb{R}$	Traditional linear algebra
AND	OR	$0, 1$	BFS reachable
FIRSTJ	ANY	$\mathbb{R}$	BFS with parents
$+$	MIN	$\mathbb{R}$	Single-source shortest path

An important feature of this formulation is that the three lines in the loop in Listing 1.1 both:

1. Express *all* the calculations performed at that stage. First all the edges needed, then all the recordings of the levels, etc.
2. Does *not* infer any ordering of these calculations.

These two features means that the back-end of a framework that implements this standard is free to order and/or parallelize the operations in the way it finds optimal. This also means algorithms written in this standard are inherently platform-agnostic and portable.

2.2 SuiteSparse:GraphBLAS

Suitesparse:GraphBLAS, by Dr Timothy Davis [7], is the reference implementation of the GraphBLAS standard. It is written in C and also provides a MATLAB interface. It provides both a full implementation of the standard and a test suite to ensure correctness of the underlying algorithms and procedures.

The default underlying structure for the graph and edges is a CSR or CSC matrix, which is the same as most other graph frameworks, which has a near-optimal memory footprint at $O(|V| + |E|)$. Vector-matrix and matrix-matrix multiplication in Suitesparse:GraphBLAS is implemented using Gustavsson's Algorithm, which has a theoretical bound of $O(f)$ in the vector-matrix case, where

f is the number of non-trivial computations, which in our case corresponds to the number of traversed edges.

A GPU-accelerated version is also in progress, and an MPI version is planned in the more distant future [4].

3 Our GraphBLAS Max-Flow Implementation

In order to test both the usability and performance of the SuiteSparse:GraphBLAS implementation, we chose to implement and benchmark an s-t maximum flow algorithm.

This problem was primarily chosen because maximum flow and its algorithms are simple enough to reason about, while still more complex and flexible than a simple primitive like BFS. In practice this meant that implementing an algorithm would force us to utilize large parts of the GraphBLAS tool set, which was a important goal for this work.

At the time of writing, there are also no published results of using GraphBLAS to implement a max-flow algorithm, making this experiment somewhat novel. There is, however, an implementation available by the authors of GraphBLAS Template Library (GBTL) in their Github repository[2]. A line-by-line translation of this implementation from GTBL:GraphBLAS/C++ to SuiteSparse:GraphBLAS/C served as a foundation for our program and also allowed us to try different approaches with the confidence of a working fall-back.

3.1 The maximum flow problem

In the maximum flow problem, the goal is to find the largest potential flow of some kind between two points in a network. This flow could be of electricity, water or some other abstract property we are interested in, and has applications in computer vision [3]. In the abstraction of graphs, what we have is a weighted graph $G = (V, E)$, where the weights represent the flow capacity for each edge. For a set of vertices V, a flow function $f : VxV \to \mathbb{R}$ and capacity function $c : VxV \to \mathbb{R}$, we can express it as a linear program:

$$\begin{aligned} &\textit{maximize} && \sum_{v \in V, v \neq s} f(s,v) \\ &\textit{subject to} && \\ & && f(u,v) \leq c(u,v) \qquad , \forall u,v \in V \\ & && \sum_{u \in V} f(u,v) = \sum_{w \in V} f(v,w) \quad , \forall v \in V - \{s,t\} \end{aligned} \tag{2}$$

That is, our goal is to maximize flow out of the source (or equivalently, the flow into the sink). The flow across an edge can be no greater than it's capacity and the flow into a vertex must equal the flow out of a vertex.

[2] https://github.com/cmu-sei/gbtl/blob/master/src/algorithms/maxflow.hpp

Minimum cut An s-t cut on a graph $G = (V, E)$ is a partitioning of the vertices into to disjoint subsets S and T where $s \in S, t \in T$. The cost of a cut is defined as $\sum_{u \in S} \sum_{v \in T} w(u, v)$, i.e. the sum of the edge weights for any edges going form S to T. A minimum cut, the cut that minimizes this sum, is shown to be equal to the value of the maximum flow [6]. This cut can be found by defining S as the vertices reachable in the residual network after finding the maximum flow, and T as the remaining vertices.

The minimum cut defines a set of bottleneck edges for the maximum flow and as such finding the max flow and min cut are equicalent problems. Some use-cases explicitly call for a minimum cut to be found, for example in image segmentation [3].

The Edmonds-Karp algorithm There are a number of different algorithms solving the maximum flow problem, and our choice fell upon the Ford-Fulkerson method/Edmonds-Karp algorithm [6]. This algorithm is based upon the breadth-first search primitive, which is known to be easily expressible as linear algebra operations.

3.2 Algorithm outline

This section will give an outline of the algorithm benchmarked. We will emphasise the different GraphBLAS-specific techniques used to construct it.

Our algorithm follows the same outline as the Ford-Fulkerson (Edmonds-Karp) procedure:

1. Find an augmenting path (using breadth-first search).
2. Increase flow along the augmenting path, updating capacities.
3. Repeat the previous steps until no augmenting path can be found.

We chose to implement this algorithm because all parts are expressible as linear algebra operations, as proven by the GBTL implementation.

The program is outlined in Listings 1.2 and 1.3. These listings do not include variable initialization, subsequent calculation of the maximum flow, or other code lines that are not necessary to understand the structure of the algorithm. But otherwise it represents a complete program, that given a matrix A calculates the maximum flow.

Listing 1.3 represents the first step in the Edmund-Karp algorithm. Given a residual graph R, it produces a boolean matrix M representing a shortest augmenting path. Lines 1-5 is essentially the GraphBLAS BFS outlined in Listing 1.1. Lines 7-10 is a sequential backtrack through the parent-pointers, constructing the boolean mask M which outlines the path.

Listing 1.2 outlines the overall structure of the algorithm. Given the matrix given by the *get_augmenting_path()* function, it extracts the corresponding capacities, finds the minimum, and then adjusts the residual capacities accordingly.

Listing 1.2: GraphBLAS maximum flow algorithm outline.

```
while(M = get_augmenting_path(R, s))
    P = M $\bigotimes$ R
    delta = min(P)
    P<M> = -delta
    P = P $\bigoplus$ (-P$^T$)
    R = R $\bigoplus$ P
    R<R> = R

max_flow = 0
for i: 1 to n:
    max_flow += A[s,i] - R[s,i]
```

Listing 1.3: get_augmenting_path()

```
frontier[source] = true
parent[source] = source
while(frontier not empty AND parent[sink] == NULL):
    frontier<parent'> = frontier * R //ANY_FIRSTJ semiring
    parent<frontier> = frontier

current_vertex = sink
while(current_vertex != source):
    M[current_vertex] = parent[current_vertex]
    current_vertex = parent[current_vertex]
```

Listing 1.4: get_mincut()

```
reachable[source] = true
frontier[source] = true
while(frontier not empty):
    frontier<reachable'> = frontier * R //OR_AND semiring
    parent<frontier> = frontier

t_cut<reachable'> = reachable * A //OR_AND semiring
s_cut<reachable> = t_cut * A //OR_AND semiring

min_cut = A<s_cut.indices, t_cut.indices> //All edges from s to t
```

Min-Cut In addition to calculating the maximum flow, we also explicitly produce the minimum cut. This procedure highlighted further interesting GraphBLAS use-cases and limitations. The process is outlined in Listing 1.4.

It is essentially a three part traversal. First to find the vertices reachable from the source (using BFS), then to find the vertices *not quite* reachable by the source, and then to find the source-side vertices incident with that second set. The indices of the two vertex-sets on either sides of the cut is then used to extract the corresponding edges.

This procedure is notable in that it requires to translate the opaque objects (matrices) into transparent ones (index arrays) and then back into opaque ones. This is generally not the intended use of GraphBLAS, and as such this is skirting along the edge of GraphBLAS's capabilities.

4 Benchmarking

The program was benchmarked with scalability in mind, as this is one of the clearer indicators of the efficiency of a parallel program. As such, it was important to both have a sufficiently powerful shared-memory system, to test with a large amount of threads, and with large representative data sets.

4.1 Data sets

The data sets utilized were the ones provided in the GAP Benchmark suite [2] by Beamer et. al. This suite is a collection of algorithms and data sets used to benchmark and compare different graph frameworks and solutions. It specifies six algorithmic kernels with provided reference implementations, five data sets and measurement methodologies. Because max-flow is not one of the algorithmic kernels, we have are only utilizing the data sets, but we have also utilized the measurement methodologies as the baseline for our own. An overview of the data sets can be seen in Table 2.

Table 2: Attributes of data sets used in our experiments.
Approximate diameter taken from Azad et. al [1].

Name	n/Vertices	Non-zeros/Edges	Edge distribution	Approx. diameter
Road	24M	58M	road-network	6,304
Twitter	62M	1468M	power-law	14
Web	51M	1930M	power-law	135
Kron	134M	4,223M	power-law	6
Urand	134M	4,295M	uniform	7

4.2 Source-sink selection

Because we worked with large datasets, and Edmonds-Karp is an algorithm that can involve hundreds of iterations of breadth-first searches, we considered it infeasible to run benchmarks using multiple sink-source pairs for each graph. As such, our goal was to find a single pair of vertices for each graph that produced a large workload, as this would give us better data of the performance and scalability of the algorithm. Large work-loads in the case of the Edmonds-Karp algorithm is determined by:

1. The length of each iterations breadth-first search.
2. The number of vertices reached in the breadth-first search.
3. The number of iterations.

We started by giving the source vertex a weight of 1. Then, we ran a BFS, and at each new level of the search, the weight of the child vertices would be set to the sum of its parents weights. This meant at level N of the BFS, each vertex would have a weight equal to the number of unique paths of length N reaching it in the breadth-first search. As this would favor vertices in the periphery of the graph, after accumulating the incoming weights at a vertex we multiplied it by a dampening factor $1/d, d >= 1$, penalizing longer paths. This dampening would ensure that a vertex at the periphery would not "inherit" the weight of an ancestor in the search. Our goal was to set this factor such that the sink would be at the last available vertex before we reach the sparse periphery of the graph, essentially at the last high in-degree vertex of the search.

After finding a potential sink candidate, we decided the source-vertex in a similar way. By performing the same search in the transposed graph, i.e. where all edges have their edges reversed, we could weight source candidates by how many paths they had *to* the sink. We repeated this process iteratively until a source and sink candidate both "agreed" on one another as the best candidate.

4.3 Experimental set-up

All experiments were performed on a Supermicro SuperServer 6049GP-TRT with 2 CPUs, Intel Xeon Gold 6230 SP - 20-Core. All cores were capable of hyper-threading, and as a result we had 80 threads available for the experiments.

4.4 Measurements

For each configuration, the maximum-flow algorithm was run 5 times on the fastest instances to ensure low variance between measurements. This was considered sufficient in these experiments too as a single run of our Edmund-Karp implementation involves multiple iterations of BFS, which is considered a single algorithm in the GAP standard.

The heaviest runs, with a thread count of 1-8 were only run once, as some took several hours to complete. Multiple runs help minimize variance imposed by the system, such as other processes running at the same time. In the cases were a run took multiple hours, this variance was assumed to be naturally evened out, and multiple runs were considered infeasible because of time constraints.

Utilizing the best practices outlined in the GAP benchmark suite, all graphs are considered loaded before any time measurements are done.

5 Benchmark results

Our goal with this work was to construct a parallel program, and as such scalabiltiy is an important aspect to look at. Scalability are metrics that measure how well the run-time scales as computing resources are introduced.

Using our single-threaded runs as the baseline, we achieved the average speedups seen in Table 4.

In sum, while the speedup differs the pattern of the strong scaling seems to be independent of graph type, graph size and which section of the code is acting

Table 3: Attributes of the problem instances.

Name	Source-sink dist.	Reachable vertices	Iterations	Min-cut size
Road	5744	100%	1274-1290	4
Twitter	6	56.9%	95-103	23
Web	22	99.8%	11-14	2
Kron	4	47.0 %	22-35	8
Urand	7	100%	197-216	48

Table 4: Average speedup for each graph with different thread counts.

	Threads					
Graph	2	4	8	20	40	80
GAP-road	1.39	1.74	1.95	2.05	2.06	1.76
GAP-twitter	2.13	3.73	5.62	8.51	9.37	8.60
GAP-web	1.95	3.32	5.09	6.99	9.15	8.25
GAP-kron	1.24	2.01	3.86	6.45	6.56	7.27
GAP-urand	1.55	2.81	5.24	9.65	11.06	10.08

as a bottleneck. This could indicate that both of the code sections have their own sequential bottlenecks. In this case, it would make perfect sense that GAP-road, which has long paths, emphasizes the search bottleneck at higher thread counts, while the power-law networks, which have a huge number of non-zeros but short paths, would emphasize the bottleneck in the flow augmentation part of the code. Through more fine-grained profiling, this was confirmed, where the GAP-road was shown to spend upwards of 85% of the runtime in the BFS-section while the scale-free sets spent around 20% of their time there. There is also possibility that there are bottlenecks inherent with the framework itself, either as a result of the underlying linear algebra algorithms or how threads are organized through OpenMP.

5.1 Iteration count variance and design choices

Iteration count varied somewhat within the run of each data set, which can be seen in Table 3. This is expected, as we have built in non-determinism in the algorithm through the use of the ANY-operator when we perform the breadth-first search that finds a new augmenting path. We believe that this is the correct choice, as it expresses an important aspect of the correctness of the algorithm. That is, the choice of parents at each level in the BFS does not affect correctness. The original GBTL implementation uses a MIN-operator instead (minimum of vertex indices), which makes their algorithm deterministic, but infers more overhead than ours. There is no reason to believe that this deterministic algorithm would consistently achieve a lower iteration count, as using the minimum vertex index is not a problem-specific heuristic.

We decided to run the data sets with the highest variance in terms of iterations again, namely the GAP-kron graph. This time, we ran used the MIN-operator instead of ANY, and we also sat GraphBLAS to blocking-mode to ensure everything was completely deterministic. The results were quite interesting. Not only did all runs have the exact same number of iterations, which was expected, they also hit the smallest known iteration count for that problem instance. Our suspicions on the increased overhead of the MIN-operator seems to have been correct, however, as the MIN-version spent more time per iteration than when using the ANY-operator. The reduced number of iterations made up for this though, and resulted in an overall reduced run-time.

6 Usability results

Usability of GraphBLAS was a another important aspect we wanted to explore in this work, which we will discuss in the following sections.

6.1 Translation examples

In order to give some hands-on examples of how GraphBLAS code looks like, we have provided a few examples of pseduo-code and Suitesparse:GraphBLAS C-code side by side in Figure 2, 3 and 4.

Traversal, Figure 2, is definitely a feature where GraphBLAS excels. While one needs to grasp the relationship between linear algebra and the graph, and understand the function signatures, the vxm itself manages to capture a double "for all" loop in a single line of code. This is an incredibly compact way of expressing this and at the same time through the semiring capture what kind of dependency exists between the in-edges of an element in the output.

Edge deletion, Figure 3, is also compact, but requires restructuring of the program, as one has to delete edges after the core loop rather than in it.

Lastly we re-visit the minimum-cut extraction, Figure 4. The resulting GraphBLAS-code here is both quite complex, and also, as discussed in Section 3.2, not consistent, as it requires the user translate back and forth between matrices and index sets.

```
for all u in frontier:
    for all (u,v) in A:
        visit(v)
```

```
GrB_vxm(_f,         //New frontier
    visited, NULL,
    f, A,
    GrB_<PLUS>_<MULT>_<type>,
    desc); //Complement mask
GrB_assign(visited,
    NULL, NULL,
    _f,
    GrB_all, n,
    NULL);
```

Fig. 2: Frontier traversal. Pseudo-code and C/GraphBLAS code.

```
//In core loop
    if (w(u,v) == 0):
        delete(u,v)
```

```
//After core loop
GrB_apply(_A,         //New graph
    A, NULL,
//Non-zero value edges as mask
    A,
    GrB_Identity_<type>,
    NULL);
```

Fig. 3: Edge deletion. Pseudo-code and C/GraphBLAS code.

```
reachable = bfs(s, R)
for all u in reachable:
    for all (u,v) in A:
        if v not in reachable:
            min_cut.add(u,v)
```

```
bfs(&reachable, s, R);

//Complemented mask
GrB_vxm(t_cut, reachable, NO_ACCUM, GxB_ANY_FIRSTJ_INT32, reachable, A, desc);

//Transpose A
GrB_vxm(s_cut, reachable, NO_ACCUM, GxB_ANY_FIRSTJ_INT32, t_cut, A, transpose_1);

GrB_Vector_extractTuples(s_indices, s_vals, &s_len, s_cut);

GrB_Vector_extractTuples(t_indices, t_vals, &t_len, t_cut);

GrB_extract(cut_extract, NO_MASK, NO_ACCUM, A, s_indices, s_len, t_indices,
    t_len, DEFAULT_DESC);

GrB_assign(min_cut, NO_MASK, NO_ACCUM, cut_extract, s_indices, s_len, t_indices,
    t_len, DEFAULT_DESC);
```

Fig. 4: Min-cut extraction. Pseudo-code and C/GraphBLAS code.

6.2 Portability

As mentioned in Chapter 3, our algorithm was originally based on a implementation written in GBTL. In fact, our baseline program was a line-by-line translation of their code from their C++-based framework to Suitesparse:GraphBLAS, which again is identical to the signatures provided in the GraphBLAS reference API.

The original plan for this project was to port the GBTL algorithm to the GPU based GraphBLAS:GraphBLAST implementation. Had GraphBLAST been a complete implementation of the standard, and in particular, if it had implemented element-wise operations on matrices, this would have been relatively painless. In other words, with little effort, we could have translated the single-threaded GBTL implementation both to a multi-core program *and* a GPU program. This speaks very favorably of the potential GraphBLAS has as a portable standard.

7 Conclusions

Graphs model a number of different networks and relationships/interactions related to building structures, social networks, electrical circuits or road networks. As these data sets grow, finding and implementing efficient graph algorithms for analyzing large graphs utilizing parallel multi-core systems is becoming more important.

The use of linear algebra, and the standardization in the form of GraphBLAS, is a very exiting development in the field of graph algorithm frameworks. It allows algorithms to be expressed in a way that is concise and consistent. The benefits of this is algorithms that are easier to reason about, which facilitates iterations and optimizations, and that are portable across different systems, which makes it incredibly flexible.

We explored this by implementing the Edmonds-Karp algorithm for maximum flow in Suitesparse:GraphBLAS, the standard's reference implementation, as well as one with support for multi-thread execution. In addition, we implemented a minimum cut extraction algorithm. Neither of these algorithms have previously published results, and as far as we are aware, this is the first implementation of minimum cut using GraphBLAS. Implementing these algorithms provided a more interesting exploration of the standard than a simple algorithm such as BFS (breath-first search).

Our usability results were shown to be very promising. By building on the GraphBLAS framework, we were able to write a program that is significantly more compact than maximum-flow algorithm implementations written in other frameworks. Our work demonstrated the portability of GraphBLAS programs by porting the GraphBLAS Template Library implementation of the algorithm. By improving the algorithm and changing certain parts of the program flow, the claim that GraphBLAS programs are composable [5] also bore out.

Utilizing linear algebra to reason about and discuss the algorithm turned out to be a great asset. This was shown when we presented different solutions for implementing the same behaviour in our work. It was also helpful when discussing solutions with the Suitesparse:GraphBLAS developer and other users, as this short-hand facilitated concise and precise discussion.

We achieved a peak speedup of 11 for the GAP-urand problem instance, but only 2 for GAP-road. None of the instances showed significant speedup beyond 20 cores. Since some experiments ran for nearly two hours even with 40 cores available, this is not ideal. It is unclear where the bottleneck exists, whether it is our implementation, Suitesparse:GraphBLAS or OpenMP, which is used to manage threads. Due to time limitations, analyzing this further is left as future work.

References

1. Azad, Ariful, e.a.: Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite. In: 2020 IEEE International Symposium on Workload Characterization (IISWC). pp. 216–227. IEEE, Beijing, China (Oct 2020). https://doi.org/10.1109/IISWC50251.2020.00029
2. Beamer, S., Asanović, K., Patterson, D.: The GAP Benchmark Suite. arXiv:1508.03619 [cs] (May 2017), http://arxiv.org/abs/1508.03619, arXiv: 1508.03619
3. Boykov, Y., Kolmogorov, V.: An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. IEEE Transactions on Pattern Analysis and Machine Intelligence **26**(9), 1124–1137 (Sep 2004). https://doi.org/10.1109/TPAMI.2004.60, conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence
4. Brock, B., Buluc, A., Mattson, T.G., McMillan, S., Moreira, J.E., Pearce, R., Selvitopi, O., Steil, T.: Considerations for a Distributed GraphBLAS API. In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 215–218. IEEE, New Orleans, LA, USA (May 2020). https://doi.org/10.1109/IPDPSW50202.2020.00048
5. Buluc, A., Mattson, T., McMillan, S., Moreira, J., Yang, C.: Design of the GraphBLAS API for C. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 643–652. IEEE, Orlando / Buena Vista, FL, USA (May 2017). https://doi.org/10.1109/IPDPSW.2017.117
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. The MIT Press, 3rd edn. (2009)
7. Davis, T.A.: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. ACM Transactions on Mathematical Software **45**(4), 1–25 (Dec 2019). https://doi.org/10.1145/3322125, https://dl.acm.org/doi/10.1145/3322125
8. Shun, J., Blelloch, G.E.: Ligra: A Lightweight Graph Processing Framework for Shared Memory. Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP) pp. 135–146 (2013), https://people.csail.mit.edu/jshun/ligra.pdf
9. Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., Owens, J.D.: Gunrock: a high-performance graph processing library on the GPU. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP '16. pp. 1–12. ACM Press, Barcelona, Spain (2016). https://doi.org/10.1145/2851141.2851145, http://dl.acm.org/citation.cfm?doid=2851141.2851145