Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Zawadi Berg Svela

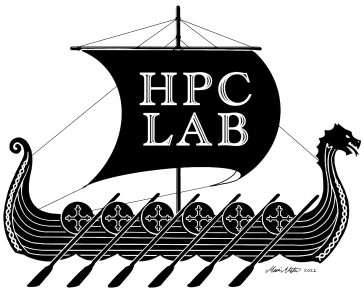
Usability Study of GraphBLAS Through Multicore Max-Flow

Master's thesis in Computer Science
Supervisor: Prof. Anne C. Elster
July 2021

NTNU
Norwegian University of
Science and Technology

Zawadi Berg Svela

Usability Study of GraphBLAS Through Multicore Max-Flow



Master's thesis in Computer Science
Supervisor: Prof. Anne C. Elster
July 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Usability Study of GraphBLAS Through Multicore Max-Flow

Zawadi Berg Svela

Project Description

In this project, the student will evaluate graph libraries. Primarily the focus will be on testing the usability of the GraphBLAS standard, and evaluate the trade-offs it provides between productivity and performance. To do this, the goal is to implement the reasonably complex Edmund-Karp max-flow algorithm in the Suitesparse:GraphBLAS implementation. Benchmarking the implemention will be on one of NTNUs powerful multi-core systems in order to test the scalability of the code.

Acknowledgements

The author would like to thank the HPC-Lab and the Department of Computer Science at NTNU for access and support of equipment throughout the past couple of years, including the Selbu system which was utilized for this work.

They would also like to thank their advisor Professor Anne C. Elster, for continued support throughout this work.

Abstract

Graphs are one of the most general and flexible abstractions for computer science problem. However, as data sets grow, the need for more computing power grows with it. This implies there is a need to adopt graph algorithms to parallel multi-core machines. Since these algorithm generally exhibit a high degree of irregularity, the parallel algorithms needed can be extremely hard to write from scratch. There is therefore a growing effort to develop high-level abstractions in the form of frameworks and libraries to address this challenge.

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 $\sim$11 over its own baseline, and is surprisingly compact and easy to reason about.

We also provide thorough discussions and examples of various ease-of-use aspects of GraphBLAS, through the lens of our max-flow implementation. We found that GraphBLAS delivers an interesting and useful perspective on graph algorithms. We show how the use of linear algebra allowed us to re-use knowledge and observations both from that field and graphs. Our work shows that many operations are *easier* to express in linear algebra than otherwise, some harder, and some impossible, illustrating the abstraction and framework have some clear limitations. Directions for future work are also included.

Sammendrag

Grafer er blant de mest generelle og fleksible abstraksjonene i datavitenskap. Men, etter hvert som størrelsen på datasett vokser trenger vi stadig mer datakraft for å behandle dem, som betyr at vi må ta ibruk grafalgoritmer som kan dra nytte av paralle flerkjernemaskiner. Dette kan være en stor utfordring, da grafalgoritmer ofte har stor grad av irregularitet, som gjør det vanskelig å konstruere slike algoritmer fra bunnen av. Det har derfor blitt utvilket mange bibliotek og rammeverk for å gjøre denne prossessen enklere.

I flere tiår har forskere optimalisert operasjoner for lineær algebra. På grunn av det ekspressive og komptakte språket fra matematikk, kan programmer skrevet som en serie av linear algebra-operasjoner gi kode som er lettere å lese og vedlikeholde. Lineær algebra er derfor en veldig attraktiv abstraksjon å ta ibruk for utrykke grafoperasjoner.

I dette arbeidet skal vi utforske brukbarheten til rammeverket GraphBLAS, den ledende standarden for grafopereasjoner utrykt som lineær algebra. Vi analyserer brukbarheten til GraphBLAS ved å implementere Edmonds-Karps algoritme for s-t maksimal flyt/minimalt kutt. Så vidt oss bekjent er dette arbeidet det første publiserte resultatet av maks-flyt implementert i GraphBLAS. Vår implementasjon ga et program som oppnådde opptil $\sim$11 økt hastighet over egen grunnlinje.

Vi inkluderer også en grundig diskusjon med eksempler for å presentere ulike brukervennlighetsaspekter ved GraphBLAS, med utgangspunkt i vår egen maks-flyt-implementasjon. Våre funn indikerer at GraphBLAS gir en interessant og nyttig måte å se på grafalgoritmer, og vi observerte at å bruke lineær algebra som astraksjon gjorde at vi kunne ta ibruk kunnskap og verktøy både fra dette feltet *og* grafteori. Vi viser at mange operasjoner på grafer faktisk er enklere å utrykke i lineær algebra, men også at noen er vanskelige og andre igjen umulige. Dette illustrerer at abstraksjonen og rammeverket har noen klare begrensninger. Vi beskriver også mulige retninger for fremtidig arbeid.

Contents

Figures

Tables

Code Listings

List of abbreviations

- BFS: Breadth-First Search.
- CSR: Compressed Sparse Row.
- CSC: Compressed Sparse Column.
- DAG: Directed Acyclic Graph.
- GBTL: GraphBLAS Template Library.
- Max-flow: s-t maximum flow.
- Min-cut: s-t minimum cut.

Chapter 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 more 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 safe-guard 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 of research worth of optimizations. All this while keeping a strong separation of concern between the user, who interacts only with high-level linear algebra objects and operations, and the low-level implementations where these, sometimes hard-ware specific, optimizations are employed.

1.1 Goals and Contributions

The goal of this work is to test the usability of the GraphBLAS standard as a framework for massively parallel graph algorithms, both in terms of productivity and performance. 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 to be 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. We will also generate maximum flow problem instances and extracted a minimum cut from the network using GraphBLAS operations. To the author´s knowledge, this thesis contains the first published results in using GraphBLAS for this problem and algorithm.

1.2 Thesis Outline

The rest of this thesis is organized as follows:

- Chapter 2: Background information related to graphs and GraphBLAS.
- Chapter 3: The maximum flow problem, the Edmonds-Karp algorithm, and our implementation.
- Chapter 4: The set-up for our experiments. This includes a description of how our maximum flow instances were generated.
- Chapter 5: We present and discuss the results of our experiments. We also discuss the usability of GraphBLAS in the context of this work through varioius examples.
- Chapter 6: Our conclusions, as well as outline future work.
- Appendix A: Paper presented at NIK 2020, based on earlier work.
- Appendix B: Additional code listings for our program.

Chapter 2

Graphs and GraphBLAS

In this chapter we will present necessary background information related to graphs and GraphBLAS. We will also briefly introduce a few other graph algorithm frameworks. Several parts of this chapter are based on sections from the the author's pre-master fall project, which was an overview of performance and usability of different graph algorithm frameworks. This work was written up as a conference paper that was presented at NIKT 2020, from which we have also borrowed material from.

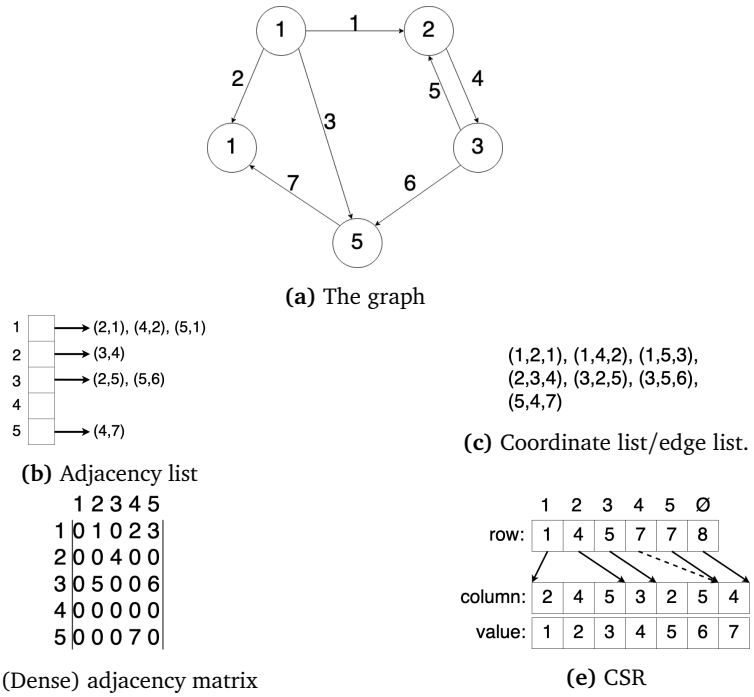
2.1 Graphs

Note: This section is largely taken from the authors own fall project report.

Graphs, in the context of this thesis, are made up of vertices (nodes or points) which are connected by edges (links or lines), which are used to model a number of different networks and relationships/interactions. These include:

- Road networks
- Social networks
- Computational dependencies
- Artificial neural networks
- Protein interactions

Formally, a graph $G = (V, E)$ contains 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, represented as a weight function $w : e \rightarrow \mathbb{R}$, in which case $G = (V, E, w)$, or they can be uniformly weighted/unweighted. The degree $deg(v)$ of a vertex is the number of edges incident to v if G is undirected, otherwise we will differentiate between in-degree $deg_{in}(v)$ and out-degree $deg_{out}(v)$.



(a) The graph

(b) Adjacency list

(c) Coordinate list/edge list.

(d) (Dense) adjacency matrix

(e) CSR

Figure 2.1: Illustrations of graph representations.

2.1.1 Graph representations

In order to be processed by a computer program, a graph needs to be stored in some fashion. The method of storage can have big impacts on both the run-time of the algorithm and/or its spacial requirements, which again will affect what problems can feasibly be solved on what systems. The optimum we want to approach is a storage requirement of $\theta(|E|)$ and an time $\theta(1)$ for inspecting the existence or property of any edge. As with many problems in computer science, there is a space/time conflict, and one usually has to sacrifice one for the other. In addition, different storage methods imply different access patterns that will affect the behaviour of hardware like cache and GPUs. The different storage methods are described in the following sections. We have provided illustrations of them in Figure 2.1 and their storage requirements and access times are summarized in Table 2.1b and 2.1b.

Coordinate list:

Possibly the simplest approach is the edge list/coordinate list (COO), which is just an list of all tuples $(i, j) \in E$, or triples (i, j, w) for weighted graphs. With a spatial requirement of $\theta(|E|)$ this is spatially optimal, and therefore well suited as a storage format. It is very inefficient in terms of access, at $O(|E|)$, so it is not suited as a run-time representation unless $|E| << |V|$, which is rarely the case.

Adjacency list:

Another simple approach is to have a list of edges per vertex, an adjacency list (AL), which could be a linked list or a contiguous array. Weights can be stored with the value of the out-vertex of an edge or in their own list. This is spatially efficient, and in the case of a linked list has the added property of easy insertion and removal of edges while still using minimal space. The access time for a single edge is $O(deg_{out}(v))$, or $O(log(deg_{out}(v)))$ for a sorted array, so this method is best suited for algorithms where each edge of any vertex is explored, or for low-degree graphs.

(Dense) adjacency matrix:

Variations of the adjacency matrix (AM) are the among most common representation of graphs, best illustrated by the dense case, see 2.1d. The matrix A is a $|V|x|V|$ square matrix, where a value a_{ij} implies an edge (i, j) with weight a_{ij}. Among all the representations, this has the best random access time at $\theta(1)$. The cost is explicitly storing all non-edges, and so its spatial requirement is $\theta(|V|^2)$. If vertices are constant, this representation has the benefit of allowing constant time additions or deletions of edges.

Sparse adjacency matrix:

As real-world graphs are often sparse, usually $|E| << |V|^2$, it is common to use sparse matrix formats like compact sparse row format (CSR) instead of dense matrix formats, see 2.1e. In CSR, the non-zero values of the matrix are stored in an array W sorted in a row-major order, with two arrays used determine the coordinates of each value in the original matrix. The array V indicates the row-indices, the values of row i is stored in $W[v_i : v_{i+1}]$ and $E[v_i : v_{i+1}]$ contains the column indices for those values. In a way, CSR can be thought of as in-between an adjacency matrix and a adjacency list.

Both in terms of space and performance CSR is similar to an adjacency list, though it is implicitly sorted, so access time is always $O(log(deg_{out}(v)))$ for a specific edge using binary search. There is also the related compact sparse column (CSC) format, which is identical to CSR except storing in a column major order, which is better for accessing in-edges rather than out-edges.

2.1.2 Characteristics of graphs

As alluded to earlier, a very important characteristic of graphs is whether it is dense or sparse. While a graph with $|E| << |V|^2$ is clearly sparse, there is not a precise definition of sparseness other than "a graph is sparse if it is beneficial to exploit the sparsity". Luckily, most real world graphs exhibit a degree of sparsity where it is clearly profitable to utilize sparse graph techniques.

Table 2.1: Key attributes of different graph representations.

(a) Comparison of space requirement

	Space
COO	$\theta(\|E\|)$
Adjacency list	$\theta(\|E\| + \|V\|)$
Adjacency matrix	$\theta(\|V\|^2)$
CSR	$\theta(\|E\| + \|V\|)$
CSC	$\theta(\|E\| + \|V\|)$

(b) Graph representation comparison
Opt: Optimal
COO: Coordinate list
AL: Adjacency List
AM: Adjacency Matrix
CSR: Compressed Sparse Row
CSC: Compressed Sparse Column

	Single edge (i, j)	All out-edges of v_i	All in-edges of v_i	Memory
Opt	$\theta(1)$	$\theta(deg_{out}(v_i))$	$\theta(deg_{in}(v_i))$	$\theta(\|E\|)$
COO	$O(\|E\|)$	$O(\|E\|)$	$O(\|E\|)$	$\theta(\|E\|)$
AL	$O(deg_{out}(v_i))$	$\theta(deg_{out}(v_i))$	$O(\|E\|)$	$\theta(\|E\| + \|V\|)$
AM	$\theta(1)$	$\theta(deg_{out}(v_i))$	$\theta(deg_{in}(v_i))$	$\theta(\|V\|^2)$
CSR	$O(log(deg_{out}(v_i)))$	$\theta(deg_{out}(v_i))$	$O(\|E\|)$	$\theta(\|E\| + \|V\|)$
CSC	$O(log(deg_{in}(v_j)))$	$O(\|E\|)$	$\theta(deg_{in}(v_i))$	$\theta(\|E\| + \|V\|)$

Many real-world graphs are also characterized as scale-free, where the vertex degrees follow a power-law distribution. This means the fraction of vertices with degree k is $P(k) \approx k^{-\gamma}$, $\gamma > 1$. In practice, this means that for large graphs, there will be a few vertices with much larger degrees than the rest. These graphs also tend to have a low diameter.

Another big category of graphs are road networks. They tend to have very low variance in vertex degrees, sometimes no vertices have degrees larger than 5-6 regardless of the graph size. They also tend to have very large diameters compared to scale-free graphs. Because of this, they present very different challenges when optimizing parallel algorithms.

2.1.3 Irregularity in graph algorithms

Different algorithms exhibit different degrees of irregularity in their computations. That is, the pattern of which data is used when in computations might be strict, loose or non-existent. On the one end, dense linear algebra operations can be categorized as highly regular, every calculation can be clearly identified immediately. Sparse linear algebra operations one the other hand, are somewhat irregular, the necessary computations aren't so readily available. Among the definitely irregular algorithms are the general graph algorithms, the necessary computation not knowable at all before running the algorithm, though, some operations can be translated to sparse linear algebra, which we explore in Section 2.2.

As one would imagine, irregularity poses a serious challenge when parallelizing algorithms. One needs to designate ownership of each workload to a executing thread, but when one does not know where work will appear, that can be difficult. A static partitioning, like assigning a set of vertices or edges of a graph to each thread, could function. But as many real-world graphs can have very skewed distribution of vertex degrees, as mentioned in Section 2.1.2, this approach could lead to terrible load-balancing, in the worst case approaching serial performance if a single thread does the majority of the work while the others stay idle.

2.1.4 Breadth-First Search

The Breadth-First Search (BFS) is an important primitive in its own right, but also a cornerstone of the Edmonds-Karp max-flow algorithm. In short, the BFS is a search in a graph G starting from a source vertex s. The simplest definition is that given a graph $G = (V, E)$ and a source vertex s, the BFS should label each vertex reachable from s with the *number of edges* one needs to traverse to reach it, also known as the *level* of a vertex.

In the context of the Edmonds-Karp algorithm, two other definitions of BFS are also useful. One is that it should simply output for each vertex whether or not it is reachable from the vertex, and the other is a labelling where each vertex points to its parent in the search given the ordering outlined above. The former of these two definitions does not infer an ordering and could technically be solved by multiple different searches.

The basic outline of a breadth-first search is as follows:

```
q = {s}
visited = {}
label(s, s)
while(q not empty):
    level = q
    for each u in level:
        remove(u, q)
        insert(u, visited)
        for each v in neighbors(u)
            if v not in visited:
                insert(v, q)
                label(u, v)
```

The label function in the above code is then adjusted to label vertex v either with the current level, a boolean "true" value or a pointer to u, depending on the version of the search needed in the algorithm.

BFS DAG

A directed acyclic graph (DAG) is a directed graph that contain no cycles, that is, there is no path from a vertex back to itself. If we run the "parent-variant" of the BFS outlined above, we produce a BFS *tree*, as each child will only have a single parent. However, if a child had instead kept *all* the parents from the previous level we would have produced a BFS DAG.

Frontier

Throughout this thesis, 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 and Ligra, see Section 2.3. They are tightly connected to the BFS, and in simpler algorithms the frontiers will correspond exactly to the levels in a BFS. The BFS DAG, see above, map out the dependencies between computations many different graph algorithms. This means that if one constructs an algorithm that works one level at a time, there is often no inter-dependency in it, which means we can freely parallelize within the frontier, which again is why it is a useful abstraction utilized by graph algorithm frameworks.

2.2 Graphs as linear algebra / GraphBLAS

GraphBLAS is a standard developed by the GraphBLAS Forum and it seeks to standardize how graphs and graph algorithms can be expressed in terms of linear algebra constructs and operations. The key observation for this goal is that if graphs are represented as (sparse) adjacency matrices, which is already beneficial (see Section 2.1.1), then we can express operations on them using 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.

2.2.1 GraphBLAS BFS

As an example to illustrate this, we will re-visit the breadth-first search (BFS), and we will focus on the parent search. That is, for each vertex, we want to note one of its parents in the BFS DAG 2.1.4. 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 which we will call the *frontier*, we want to both determine the vertices in the next level outward and note which vertex "discovered" them.

$$f_i' = \sum_{f_j \in f, A_{ij} \in A} f_j * A_{ij} \tag{2.1}$$

Equation 2.1 shows the calculation for a given value $f_i' \in f'$ for the vector matrix multiplication $f' = fA$. In our context, f represents the current frontier of vertices, f' the next frontier of vertices and A the edges of the graph. As it is, the individual multiplications correspond to the edges we wish to traverse, and the summation for each vertex corresponds to the reduction we employ if multiple vertices in one frontier are adjacent to the same vertex in the next one.

In order to not just get the pattern we want, but also the result, we introduce is the semiring, an algebraic structure that re-defines the additive and multiplicative operator. In order to extract the index of the parent-vertex in each level of the search, we use FIRSTJ as our multiplicative operator. FIRSTJ is defined as the column/j-index of the first operand. The additive operator simply needs to return a single value out of the candidate parents, so we can set it to the ANY operator to achieve the desired result.

In order to express convergence we need to ignore any vertices we have already found. For this purpose GraphBLAS defines a mask, a structure with the same dimensions as the output that indicates which values we actually want. In this case, we will note visited vertices in a parent-vector and use its structural complement as the mask. That is, only values *not* present in the parent-vector are eligible to be written.

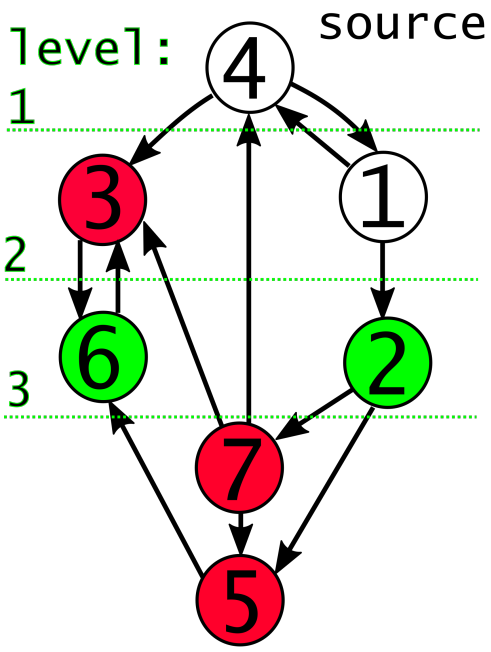
Listing 2.1 outlines this BFS algorithm. A example of a single step of the search can be seen in figure 2.2.

Code listing 2.1: Outline of a breadth-first search in GraphBLAS

```
1 parent<s> = s
2 frontier<s> = s
3 while (frontier not empty):
4         frontier<parent'> = x * A
5         parent<x> = x
```

Figure 2.2: A single step of a breadth-first search as a vector-matrix multiplication.

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



(b) Linear Algebra operation

```
                . 1 . 1 . . .
                . . . . 1 . 1
                . . . . . 1 .
. 1 . . . 3 . * 1 . 1 . . . .
                . . . . . 1 .
                . . 1 . . . .
                . . 1 1 1 . .
                      =
                . . 6 . 2 . 2
```

Table 2.2: Semirings and resulting primitive
FIRSTJ: J-index of the first argument
ANY: Any one of the two values

$\otimes$	$\oplus$	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

Table 2.3: Overview of some GraphBLAS objects. Based on a table in [3]

Object	Description
GrB_Matrix	A sparse matrix.
GrB_Vector	A sparse column vector.
GrB_UnaryOp	A unary scalar operator.
GrB_BinaryOp	A binary scalar operator.
GrB_Monoid	An associative and commutative binary operator and its identity value.
GrB_Semiring	A collection of a monoid and a binary operator, defining "plus" and "multiply" as needed.
GrB_Descriptor	A collection of parameters denoting how inputs and outputs are read in a GraphBLAS operation.

By simply changing the semiring used, we can express other algorithms using the same basic structure. Examples of some semirings and the calculation they infer can be seen in table 2.2.

An important feature of this formulation is that the three lines in the loop in the figure 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.

In sum, 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 means formulating algorithms in this standard makes the algorithms inherently portable across multiple different systems.

2.2.2 GraphBLAS objects and operations

GraphBLAS defines a set of objects and operations, some of which can be seen in table 2.3 and 2.4, some of which have been implied in previous sections. They all have rigorous mathematical definitions outlined in the GraphBLAS API [4].

Table 2.4: Overview of some GraphBLAS operations. Based on a table in [3]. $\odot$ indicates an (optional) accumulator, a binary operator.

Operation name	Description	GraphBLAS math notation
GrB_mxm	Matrix-matrix mult.	$C\langle M\rangle = C \odot AB$
GrB_vxm	Vector-matrix mult.	$\boldsymbol{x}^T\langle m^T\rangle = \boldsymbol{x}^T \odot \boldsymbol{y}^T B$
GrB_mxv	Matrix-vector mult.	$\boldsymbol{x}\langle m\rangle = \boldsymbol{x} \odot A\boldsymbol{y}$
GrB_eWiseMult	Element-wise multiplication.	$C\langle M\rangle = C \odot (A \otimes B)$ $\boldsymbol{x}\langle m\rangle = \boldsymbol{x} \odot (\boldsymbol{y} \otimes \boldsymbol{z})$
GrB_eWiseAdd	Element-wise addition.	$C\langle M\rangle = C \odot A \otimes B$ $\boldsymbol{x}\langle m\rangle = \boldsymbol{x} \odot (\boldsymbol{y} \oplus \boldsymbol{z})$
GrB_extract	Extract sub-matrix/vector.	$C\langle M\rangle = C \odot A(\boldsymbol{x}, \boldsymbol{y})$ $\boldsymbol{x}\langle m\rangle = \boldsymbol{x} \odot y(\boldsymbol{z})$
GrB_assign	Assign sub-matrix/vector.	$C\langle M\rangle(\boldsymbol{x}, \boldsymbol{y}) = C(\boldsymbol{x}, \boldsymbol{y}) \odot A$ $\boldsymbol{x}\langle m\rangle(\boldsymbol{z}) = \boldsymbol{x}(\boldsymbol{z}) \odot \boldsymbol{y}$
GrB_assign_<type>	Assign scalar to all elements in sub-matrix/vector.	$C\langle M\rangle(\boldsymbol{x}, y) = C(\boldsymbol{x}, y) \odot s$ $\boldsymbol{x}\langle m\rangle(\boldsymbol{z}) = \boldsymbol{x}(\boldsymbol{z}) \odot s$
GrB_apply	Apply unary operator.	$C\langle M\rangle = C \odot f(A)$ $\boldsymbol{x}\langle m\rangle = \boldsymbol{x} \odot f(\boldsymbol{y})$
GrB_reduce	Reduce to vector or scalar.	$\boldsymbol{x}\langle m\rangle = \boldsymbol{x} \odot [\bigoplus_j A(:, j)]$ $s = s \odot [\bigoplus_i jA(i, j)]$ $s = s \odot [\bigoplus_i \boldsymbol{x}(i)]$
GrB_transpose	Transpose matrix.	$C\langle M\rangle = C \odot A^T$

Of note is that all objects are opaque. That is, the user cannot interact with them outside of the defined operations. This allows for a number of optimizations as long as the object obeys the mathematical definitions when the user interacts with them. Among these is that GraphBLAS defines a non-blocking mode, where any modification can be postponed until needed, which allows multiple individual changes to be chained together or performed in bulk.

As GraphBLAS is intended to be used for graph processing, it is important that it supports sparse matrices, see Section 2.1.2. Importantly, the value of a non-element in a matrix or vector is user defined. In the case of ordinary linear algebra, non-elements are always zero-valued, but if we were running a single-source shortest path and used a PLUS-MIN semiring, we would want these elements to be treated as "infinity". This is defined by the semiring in addition to the operators, and means that we can run a number of different algorithms on the same graph without explicitly changing any values. This allows GraphBLAS implementations to utilize sparse matrix representations, like the ones outlined in Section 2.1.1.

In addition to the outlined input and output, most GraphBLAS operations accept a set modifier objects. One of these is the mask, which was described in the previous section. Another important one is the descriptor, which defines how the operations reads and writes the inputs, mask and output. Particularly useful is the ability to note that the mask should be evaluated only by structure instead of value, which means explicit zeros (like a parent-pointer to the zero-indexed vertex or a zero-weighted edge) still imply a write in the output object. Another is the ability to consider the masks complement, that is, only the elements *not* implied by the mask.

2.2.3 Parallelism in GraphBLAS

Bulk-operations in GraphBLAS are all defined in such a way to imply a degree of parallelism. Assign and apply operations define fully independent calculations, similarly to filter and compute-functions in the frontier-based framework Gunrock [5]. While the specifics of these calculations can vary between implementations, the Gunrock authors note that these are generally trivially parallelizable. The Gunrock authors note, however that their advance operator, which has near-identical behaviour to the VxM operation in GraphBLAS, is significantly harder to parallelize. This operation has a high degree of irregularity, as vertices can have highly varying in and out-degrees.

As outlined by Pingali et. al [1], problems themselves also express very varying degrees of parallelism. They outline a different pattern of computation, but the principles are the same. In for example a BFS, the beginning of the algorithm will naturally have a low degree of parallelism, as each vertex's computation is dependent on its parent's computation. In the case of GraphBLAS, this simply translates to parallelism being dependent on the size of the frontier vector in the VxM operation. Or generally, the available parallelism varies with the volume of

individual computations, which is dependent on the underlying graph and for example the kind of frontiers it implies.

2.2.4 Suitesparse:GraphBLAS

Note: This section is taken from a paper written by the author for DT8117.

Suitesparse:GraphBLAS, by Dr Timothy Davis [3], 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, that is, the actual traversed edges. This bound assumes a previously allocated buffer with a size linear to the number of vertices, but this would only have to be allocated once for a given algorithm, and is therefore easily amortized.

The main matrix storage formats are CSR and CSC 2.1 with corresponding hypersparse versions for cases where $|E| << |V|$. The implementation supports non-blocking execution, which means any operation can be delayed as long as the objects remain mathematically consistent.

In version 3, they introduced parallel execution of linear algebra operations by utilizing OpenMP [6]. OpenMP is an open-source standard for simple parallel execution of code regions.

A GPU-accelerated version is also in progress, and an MPI version is planned in the more distant future [7].

2.2.5 Other GraphBLAS implementations and project

In addition to Suitesparse:GraphBLAS, several other implementations of the standard exist, and in this section we will introduce a couple of notable ones. We will also introduce the LAGraph project.

GraphBLAST

Note: This section is based on a section from the authors fall project report.

GraphBLAST, Yang et. al [8], is an implementation of GraphBLAS targeting Nvidia- GPUs, and is the GPU implementation with a focus on high performance [9]. It was developed with the goal of making the high performance of frameworks such as Gunrock [5] and D-IrGL (Galois) [10] more accessible by utilizing the GraphBLAS standard.

Among the optimizations it employs to achieve this performance is what the authors refer to as generalized direction-optimization. This novel technique is an

extension of a technique employed in a BFS-implementation by Beamer et al. [11], where it was observed that when the frontier of a breadth-first search is large enough, it is beneficial to pull information from the vertices outside the frontier rather than the frontier vertices pushing the search forward. This technique is referred to as push-pull optimization and translates naturally to linear algebra formulations. The computational pattern from push and pull corresponds to the difference between a sparse frontier vector representation and a dense representation. This sparse/dense view translates beyond BFS search, which is why it is referred to as generalized. Because of this technique, GraphBLAST will sometimes store graphs in both CSR and CSC formats, as CSR has better performance for pull-style computations while CSC is more suited for push operations.

They also identified the potential of exploiting *input sparsity* in their operations. According to the standard, when a mask is provided in a operation, it is applied *last*, that is, the right-hand result is fully determined, and then the mask simply determines which parts of this result is written to the output. If a mask in a GraphBLAS operation is particularly sparse, however, then there can be significant performance gained by using the mask *first* to determine which calculations actually need to be performed.

At the time of writing, GraphBLAST does not fully implement the GraphBLAS standard. Crucially, it does not implement element-wise addition, which is necessary to implement a max-flow algorithms, as one needs to be able to update flow along any edge in the graph. Alternatives would be to store the per-edge values in a $|E|$ long vector or re-build the adjacency matrix for each iteration, neither of which would utilize the linear algebra formulation to the fullest extent.

GBTL

The GraphBLAS Template Library (GBTL) [12], is an older implementation of GraphBLAS. Written in C++, its goal is primarily to provide an easy to use framework along with example algorithms written in the standard. As such, it is not necessarily designed with high performance as the primary goal. Its contribution through the many algorithms they provide is however, of note.

An older version of GBTL, and what is presented in the cited paper, also provided support for running programs on GPUs using CUDA. While GPU support is no longer available in GBTL, that is an important milestone, as GPUs have much higher peak operations per second. The CUDA-implementation is built on top of Thrust and CUSP (which again is built on top of Thrust), which provide the necessary vector and matrix operations to implement parts of the GraphBLAS standard, including multiplication and element-wise addition/multiplication across two vectors/matrices.

GBTL only supports sequential execution. V2 had a proof of concept GPU support, which was removed in v3.

2.2.6 LAGraph

LAGraph [9] is a growing effort to collect algorithms written in the GraphBLAS standard. It contains a variety of different programs written for GraphBLAS provided by users. At the time of writing, it supports only Suitesparse:GraphBLAS. However, as outlined in the beginning of this section, algorithms in GraphBLAS easily can be ported to most implementations, as they define the same set of operations, even if programming language and signature might differ.

2.3 Other graph frameworks

There are a plethora of different frameworks for parellell graph processing, both for multicore CPU systems and GPUs. What follows is a short summary of some relevant ones, with descriptions taken from the authors own conference paper [13], which is also attached in Appendix A.

2.3.1 Galois

Galois is a framework built around Tao-analysis, [1], and uses the *operator formulation* of algorithms to reveal *amorphous data-parallelism* As such, the key features Galois provides are concurrent data structures and parallel loops that allow generation of new work items/loop iterations as the loop is running. They also provide support for priority and ordering between work items. These parallel loops can also be thought of as unordered set iterators.

D-IrGL is one of the more recent additions to the continually growing family of systems associated with Galois. It is the combination of the communication substrate Gluon [10] with the GPU-targeted intermediate representation IrGL [14], and it facilitates distributed heterogeneous graph applications. These ideas were developed further into the Abelian compiler [15], which produces Gluon+IrGL code from ordinary Galois code written for CPU. The complete system effectively turns Galois into a portable framework for multi-core, distributed and potentially heterogeneous graph algorithms.

2.3.2 Gunrock

Gunrock is a frontier-based framework, see Section 2.1.4, for single-node GPU accelerated systems. In its original implementation, it provided three functions for processing frontiers: *Advance* for traversal, *compute* for label updates and *filter*. A few more specialized operators where added later.

2.3.3 Ligra

Ligra is a light-weight frontier-based framework for CPUs. It provides only two functions, *edgeMap* and *vertexMap* and a *vertexSubset* type for the frontier. EdgeMap is the advancing operator, and applies a given function to all edges out of the

frontier given a condition is met. VertexMap is similar to Gunrock's *filter* function. Both functions have precise mathematical definitions provided in the paper, and as a consequence Ligra is a very compact framework, which is explored further in Section

Chapter 3

Max-Flow in GraphBLAS

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 during this thesis.

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 GBTL. This implementation served as a foundation and also allowed us to try different approaches with the confidence that there was a working fall-back.

In the following sections, we will first give an introduction to the s-t maximum flow problem. We will then present our choice of algorithm, as well as discuss why we chose this particular one.

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 [16]. 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 $f : VxV \rightarrow \mathbb{R}$ and capacity function $c : VxV \rightarrow \mathbb{R}$, we can express it as a linear program:

$$
\begin{aligned}
&\textit{maximize} && \sum_{v \in V, v \neq s} f(s,v) \\
&\textit{subject to} && \\
& && f(u,v) \leq c(u,v) && ,\forall u,v \in V \\
& && \sum_{u \in V} f(u,v) = \sum_{w \in V} f(v,w) && ,\forall v \in V - \{s,t\}
\end{aligned}
\tag{3.1}
$$

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.

3.1.1 Maximum flow algorithms

There are a number of different algorithms sovling the maximum flow problem, and our choice fell upon the Ford-Fulkerson method/Edmonds-Karp algorithm [17]. This algorithm is based upon the breadth-first search primitive, which is known to be easily expressible as in terms of linear algebra operations. There is also an open source implementation available on the GitHub repository of GBTL 2.2.5, which served as the basis for our own implementation and will be discussed further in Section 3.2.3.

Other algorithms were considered. In a survey paper outlining the relative performance of maximum-flow algorithms, [18], Ahuja et. al presents a selection of different solutions and discuss their relative merits. A recurring problem, however, was that none of these could be represented practically as a series of linear algebra operations.

Variants of Push-Relabel [19] were considered, as they are not particularly complex algorithms to express sequentially. However, common for all these algorithms is that they label each with a distance/height and filter viable edges (u,v) based the difference $l(u) - l(v)$. This turned out to be quite complex to express in the GraphBLAS standard. Our suspicions that this would not be a viable approach were validated by the implementation available in the GBTL repository, as it hardly utilized any linear algebra bulk operations, instead implementing a essentially sequential algorithm using random accesses.

Dinic's algorithm [20] and its improvement, the MPM algorithm [21], have better theoretical run-time bounds than Edmonds-Karp and Push-Relabel. They both however, utilize depth-first searches to construct their augmenting path. While it is possible to express depth-first searches as linear algebra operations [22], this was deemed too complex for our program, as it requires us to construct larger more complex matrices and vectors other than the basic adjacency matrix and frontier vector. As depth-first search is a P-complete problem [23], it also is not suited for work primarily focused on multi-core processing.

3.1.2 The Edmonds-Karp algorithm

The Edmonds-Karp algorithm follows the simple outline given by the Ford-Fulkerson method:

1. Find an augmenting path.
2. Increase flow along the augmenting path, updating capacities.
3. Repeat the previous steps until no augmenting path can be found.

The specification that makes it the Edmonds-Karp algorithm is that the augmenting path is found through a breadth-first search, and the shortest path in terms of number of edges is chosen during each iteration. The increase in flow is the minimum residual capacity along this path. This gives the algorithm an $O(VE^2)$ upper bound in run-time [17].

This algorithm does not work on the graph directly, but rather on a *residual graph*. This residual graph represents the available capacity between all pairs of vertices, and starts of as a copy of the original flow network. Whenever flow is increased by a factor d along a path P, all capacities $c(u, v)$ in the residual graph along the path is lowered by d. Also, crucially, capacities along the parallel opposite edges $c(v, u)$ are *increased* by a factor d to represent flow that could be re-directed later. This generally means edges are both added and deleted in the residual graph at every iteration of the algorithm, as their capacities go from zero to non-zero or from non-zero to zero.

3.1.3 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 [17]. 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. As such, it can be useful to find explicitly in many maximum flow use-cases. In particular, it has applications in image segmentation [16].

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.

The program is outlined in Listings 3.1 and 3.2. 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.

Code listing 3.1: GraphBLAS maximum flow algorithm outline.

```
while(M = get_augmenting_path(R, s))
    P = M ⊗ R
    delta = min(P)
    P<M> = -delta
    P = P ⊕ (-P^T)
    R = R ⊕ P
    R<R> = R

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

Code listing 3.2: 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]
```

Code listing 3.3: 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
```

The corresponding C-code to Listings 3.1 and 3.2, as well as the minimum-cut procedure in Listing 3.3, see Section 3.2.2, can be found in the appendix as Listings B.1, B.2 and B.3 respectively.

Listing 3.1 outlines the overall procedure of the algorithm, which we will give an thorough explanation of here.

1. We obtain the augmenting path, if there is one, as a boolean matrix M outlining the path.
2. Extract the corresponding capacities from the residual graph R. This is done by performing an element-wise multiplication between M and R. We used SECOND as our binary function, which returns the second operand (the capacity), but since M is boolean, TIMES could also have been used.
3. Perform a reduction on the extracted value to obtain the minimum capacity along the path.
4. We assign this minimum value on every edge in the path negated to represent the negative change in capacity, using the boolean path M as the mask.
5. To insert the positive changes to the capacities along the parallel opposite edges, we use a additive inverse unary function on all elements of P, transpose them, and add them back into P.
6. The changes stored in P is then applied to the residual graph by performing an element-wise addition between the two matrices.
7. We need to remove explicit zeros from R, which is done by applying a unary identity function to all elements in R and writing them back to R but using R as a mask. That is, for each value, we write that value back, but only if it does *not* equal zero, as those values are ignored because of the (same) zeros in the mask.
8. After the core loop terminates, we calculate the maximum flow by accumulating the difference between the capacities and residual on all edges going out of the source.

Listing 3.2 represents the first step. 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 Section 2.2.1. We produce a vector where for each vertex reached in the search we note its parent. The only difference from the algorithm outlined in the Background chapter is that we terminate as soon as we find the sink. Lines 7-10 is a sequential backtrack through the parent-pointers, constructing the boolean mask M which outlines the path.

3.2.1 Assign

There are two instances were we could have used masked assign instead of element-wise multiplication and a unary function apply, respectively, which we believe would have made for a more readable algorithm. Some informal experiments indicated that this was no slower than the version outlined above. Correspondence with the developer of the Suitesparse:GraphBLAS code also confirmed this, and they indicated that using assign in these instances could actually be faster, at least

Code listing 3.4: Outline of the algorithm, replacing element-wise multiplication and apply with assign.

```
1  while(M = get_augmenting_path(R, s))
2      P<M> = R
3      delta = min(P)
4      P<M> = -delta
5      P<M^T> = delta
6      R = R ⊕ P
7      R<R> = R
8
9  max_flow = 0
10 for i: 1 to n:
11     max_flow += A[s,i] - R[s,i]
```

in an upcoming version of the framework.

The first one is for extracting capacities from R given the boolean path M, step 2 above. We could have simply assigned the capacities of R directly to P and used M as the mask.

The second one corresponds to step 5 above. Instead of applying a unary onto a transposed P and adding the result back to P itself, we could have transposed M and then assigned the delta as a scalar assign. This would make significantly more sense mathematically and also reduce the time of the operation to $O(1)$ according to the developer.

Applying these changes would lead to a new algorithm outline seen in Listing 3.4. In the appendix, these alternatives are also presented within the C-code listing outlining the program B.1, indicating where and how it would change. This improvement was discovered too late to be part of the benchmarks.

3.2.2 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 3.3. The minimum cut is defined in Section 3.1.3. We find it by running a simple BFS in the residual graph to find the vertices reachable from the source (line 1-5). To find the vertices incident to the T-side of the cut, we then perform a single traversal step from the reachable vertices into the original graph. Similarly, we find the vertices on the S-side of the cut by traversing a single step from the "T-vertices" back towards the reachable ones.

Lastly, we use the indices from these vertices to find the edges in the minimum cut. This last step is the only that is notably more complex when written out in the GraphBLAS standard. We first extract the indices of the vertices, then use them to extract the edge capacities. We then translate the local indices of the extracted edges back to global indices in the graph, which is lastly used to construct a new graph containing only the minimum cut edges.

3.2.3 GBTL maximum flow algorithm

Our algorithm borrows from one written by the developers of GBTL 2.2.5, which was made publicly available in their Github repository [1]. The two implementations are structurally very similar, and our follows the GBTL program quite closely. Where applicable, we have however changed GraphBLAS operations to increase readability of the program. This includes:

1. We use a masked assigns to insert the negative capacity changes into the path matrix, instead of a unary function apply.
2. The GBTL impementation constructs a vector called index_ramp, which they use to associate vertices with their parents in the breadth-first search. We instead employ the FIRSTJ multiplicative operator for the same result.
3. We use ANY as our additive operator in the BFS rather than MIN, as this more clearly communicates the intent of the algorithm, and leaves more room for back-end optimizations.
4. We explicitly extract the minimum cut, this is not done in the GBTL implementation.
5. We changed the order of some operations to make the program easier to parse and understand.

Our goal with these changes was to shorten and/or simplify the program. In general, changes that lengthened the program were not implemented, even if if they would lead to better clarity. Otherwise, it was assumed that changes that would make the code more readable would not make the program slower.

3.3 Parallelism

As the program is based on GraphBLAS, it is inherently a bulk-synchronous approach. Each GraphBLAS function call functions as a superstep where calculations can be performed in parallel, and changes are not assumed visible until the call returns. We are using visible in the sense that updates are noted within the GraphBLAS environment. All GraphBLAS objects are opaque, which means none of the updates are visible to the user until explicitly queried.

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

Chapter 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 [24] by Beamer et. al. This suite is a collection of algorithms and data sets used to benchmark and compared 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.

As can be seen in Table 4.1, the data sets provided in the GAP benchmark suite span a few key graph characteristics. This includes both directed and undirected graphs, and varying edge distributions and diameters. Three of the five graphs are constructed from real-world data, so it also provides a realistic view of how a program would actually fare "in the wild".

Table 4.1: Attributes of data sets used in our experiments.
All data-sets are from the GAP benchmark suite [24].
Approximate diameter taken from Azad et. al [25].

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

All data sets have integer weighted edges, even if the original data set it is based on was unweighted.

Matrices were provided in the Matrix Market file format. This is a coordinated list (COO) format, where each value in the matrix is provided as a triple (i, j, v). GrB_Matrix_build is a function provided in GraphBLAS that constructs matrices with COO formatted input. It does, however, assume three separate arrays for i, j and v, instead of a stream of triples. This meant a custom file reading program was written to read mtx-files and construct the arrays.

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 Edmund-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.

The GAP datasets each provide a set of 64 sources for use with algorithms such as BFS and SSSP, but they did not also come with a corresponding sink. As such, processing them were considered wasted work, as they were essentially no better candidates than random vertices for our purposes.

In order to find candidates the source and sink, we ran a breadth-first search starting from an arbitrary source vertex. We first opted for a fully automated process which tried new random source vertices until it found one that could reach at least half the graph, and then sat the sink vertex to the one that maximized the depth. The reasoning behind this was that this would create the largest possible flow network. However, experiments found that maximizing depth also near-guaranteed that the sink vertex would be placed at a periphery of the graph with only a small number of edges incident which would be filled very early in the algorithm, resulting in runs with very low iteration counts.

In order to counter-act this, we adjusted the breadth-first search. We started by giving the source vertex a weight of 1. Then, as we ran the BFS, 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 BFS DAG 2.1.4. As this would again favour 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.

Code listing 4.1: Source-sink search. Weights and frontier are $|V|$-length vectors, A is the adjacency matrix.

```
bfs_source = 0;  //Current source of breadth-first search
can_s = 0;          //Candidate s/source
can_t = NULL;       //Candidate t/sink
prev_s = NULL;      //Previous s/source
prev_t = NULL;      //Previous t/sink

while (can_s != prev_s OR can_t != prev_t):
    weights[bfs_source] = 1
    frontier[bfs_source] = 1

    while (frontier not empty):
        frontier<weights'> = frontier * A                  //PLUS_FIRST semiring
        frontier = frontier ⊗ dampening_factor //Element-wise multiplication
        weights = weights ⊕ frontier            //Add new weights

    //Find candidate s/t and flip the search direction
    if(bfs_source == can_s):
        can_t = max(weights)
        bfs_source = can_t
    else:
        can_s = max(weights)
        bfs_source = can_s

    A = A^T
    clear(weights)
    clear(frontier)
```

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. Pseudo-code for the complete procedure can be seen in listing 4.1.

As outlined in section 2.1.2, different graphs can have very different distribution of edges. This in turn meant that it was impossible to fully automate the source-sink search. On the road network, with a low average vertex degree and a high diameter, we needed a low dampening factor. Otherwise the search would favor sink vertices very close to the source, whih would exclude large parts of the graph from the problem instance. On power-law graphs we found the opposite to be true, where higher d was needed to keep the sink from being placed in the periphery.

Table 4.2: Software versions

Software	Branch/version	Dependencies
Suitesparse:GraphBLAS	stable/0618e95108	OpenMP
Docker	20.10.7, build f0df350	
gcc	7.5.0	
Python	3.6.9	
Ubuntu Linux	18.04.5	

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 hyperthreading, and as a result we had 80 threads available for the experiments.

Number of utilized threads/cores was controlled using the OpenMP environment variable OMP_NUM_THREADS, which was set to selected numbers between 1 and 80.

Table 4.2 provides an overview of the software utilized in these experiments, along with version numbers.

4.4 Measurements

Utilizing the best practices outlined in the GAP benchmark suite, all graphs are considered loaded before any time measurements are done. Within this we have also included any variable initialization that would be done only once across multiple trials. Any initialization that has to be done in a per-trial basis however, like allocating and copying the graph to construct a residual graph, is timed.

We ran each graph with a number of different thread configurations, which is discussed further in 5. For each configuration, the maximum-flow algorithm was run 5 times on the fastest instances to ensure low variance between measurements. This exceeds the lowest number of iterations found in the GAP benchmarking standard, which is 3. 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.

For each run, the primary measurement was the total time from duplicating the graph to create a residual graph to outputting the maximum flow. In addition, we measured how much of the time was spent finding new augmenting paths and how much time was spent adjusting the flow along said paths. Lastly we measured the time it took to determine the minimum cut as well as the time spent loading

the matrix, to ensure neither became a bottleneck in the program. These measurements were done using the Linux's built-in gettimeofday() function, which was inserted around relevant sections of the code.

Chapter 5

Results and Discussion

In this chapter we will discuss the results of our work. We will begin by discussing the results from our benchmarks, and then provide a discussion and evaluation of the usability of GraphBLAS.

5.1 Benchmark results

In this section we present and discuss the results from our benchmarks, which followed the process outlined in Chapter 4.

5.1.1 Speedup and Scalability

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.

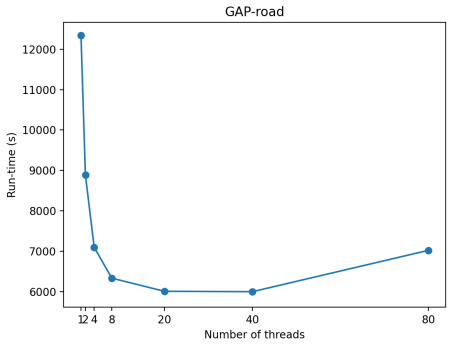
Using our single-threaded runs as the baseline, we achieved the average speedups seen in Table 5.3.

Strong scaling

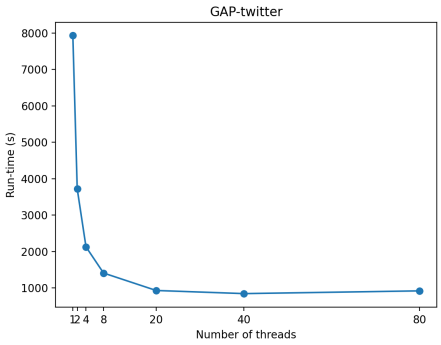
We will mostly consider strong scaling, that is, keeping the problem size constant as resources are introduced. As is discussed further in Section 5.1.3, this is not something we can measure completely directly, as iteration count is part of

Table 5.1: Attributes of the problem instances.

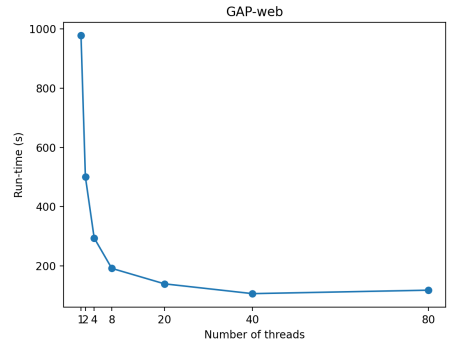
Name	Source-sink dist.	Reachable vertices	Iterations	Min-cut size
Road	5744	100%	1274-1290	4
Twitter	6	56.9%	95-103	23
Web	22	99.8%	11-14	2
Kron	4	47.0 %	22-35	8
Urand	7	100%	197-216	48



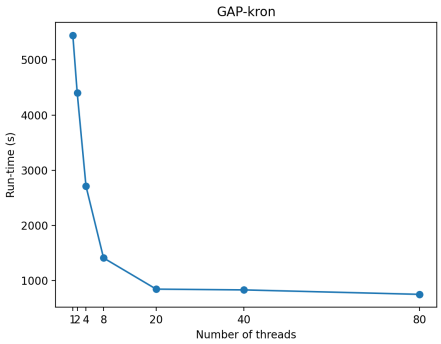
(a) GAP-road run-times.



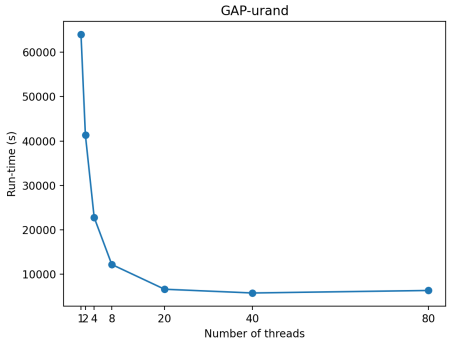
(b) GAP-twitter run-times.



(c) GAP-web run-times.

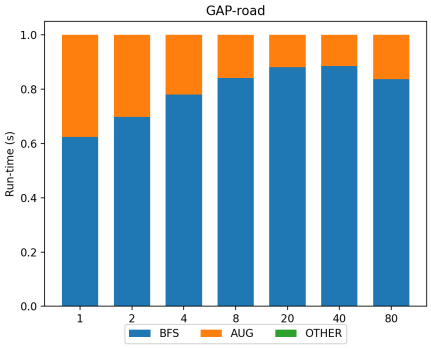


(d) GAP-kron run-times.

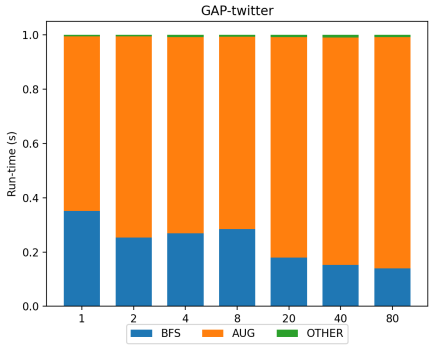


(e) GAP-urand run-times.

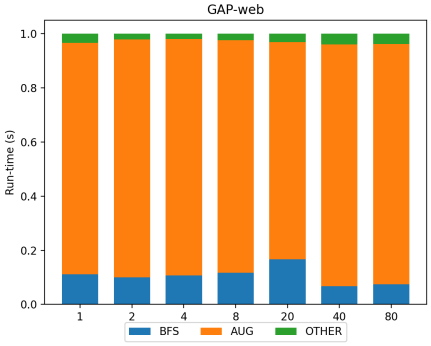
Figure 5.1: Average run-time for each problem instance and thread configuration.



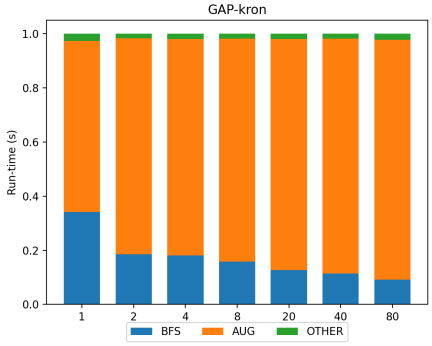
(a) GAP-road percentage time.



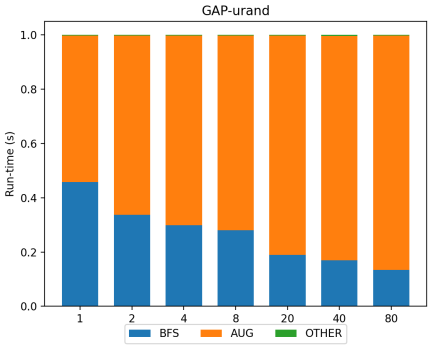
(b) GAP-twitter percentage time.



(c) GAP-web percentage time.



(d) GAP-kron percentage time.



(e) GAP-urand percentage time.

Figure 5.2: BFS vs. Augment profiling. Percentage-wise time spent performing the Breadth-First Searches, Augmenting the flow along the found path, or in other sections of the code, for different thread counts between 1 and 80.

Table 5.2: GAP-kron non-deterministic vs. deterministic. Comparing iteration count and run-time for different thread counts when using the non-deterministic ANY-operator or the deterministic MIN-operator for BFS.

Table 5.3: 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

the problem size and not consistent across runs. However, the variance is small enough, especially in the instances with iteration counts > 100.

Looking at Figure 5.1, the pattern overall is quite consistent, and as expected for strong scaling. Assuming a problem instance with some proportion of the program that can be run in parallel and some that is inherently sequential, as we introduce more and more threads, the time spent in the parallel sections drops to a minimum and the sequential run-time stays constants. This is most certainly what is observed here, as we can see the sequential minimum run-time is also dependent on problem size. This follows the pattern known as Amdahl's law.

What is not consistent is the size of the bottleneck, as we saw in Table 5.3. It is clear that GAP-road has a much larger sequential bottleneck than the other graphs, as it never achieves a speedup higher than ~ 2.

If we further look at Figure 5.2, we can where the different instances spent most of their time. These figures show estimates of time spent in the two main portions of the code, namely finding the augmenting path and then augmenting flow along that path. The GAP-road problem instance infers much greater time spent in the breadth-first search than the others, even at higher thread counts. This seems to indicate that the search is where we find the bottleneck for the GAP-road instance, while the others have their bottleneck in the capacity adjustment.

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 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. This is not unlikely, as the search has a sequential loop that punishes long paths, and the augmentation has an edge deletion section that punishes the largest graphs. 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.

GAP-kron, 80 threads

ANY			MIN		
Time	Iterations	Time/iteration	Time	Iterations	Time/iteration
819.18	31	26.43	611.61	22	27.80
696.96	26	26.81	614.55	22	27.93
825.43	31	26.63	604.53	22	27.48
717.29	27	26.57	609.40	22	27.70
686.48	26	26.40	609.04	22	27.68

GAP-kron, 40 threads

ANY			MIN		
Time	Iterations	Time/iteration	Time	Iterations	Time/iteration
807.67	32	25.24	579.35	22	26.33
826.34	32	25.82	590.97	22	26.86
813.86	32	25.43	582.35	22	26.47
841.91	33	25.51	588.3	22	26.74
858.88	34	25.26	592.51	22	26.93

GAP-kron, 20 threads

ANY			MIN		
Time	Iterations	Time/iteration	Time	Iterations	Time/iteration
784.28	28	28.00	647.88	22	29.45
861.21	32	26.91	672.24	22	30.56
837.80	31	27.03	645.1	22	29.32
829.33	30	27.64	662.77	22	30.13
905.12	34	26.62	644.74	22	29.31

GAP-kron, 8 threads

ANY			MIN		
Time	Iterations	Time/iteration	Time	Iterations	Time/iteration
1,408.79	27	52.18	1221.01	22	55.5

GAP-kron, 4 threads

ANY			MIN		
Time	Iterations	Time/iteration	Time	Iterations	Time/iteration
2709.61	29	93.43	2134.82	22	97.04

GAP-kron, 2 threads

ANY			MIN		
Time	Iterations	Time/iteration	Time	Iterations	Time/iteration
4399.55	26	169.21	3921.57	22	178.25

GAP-kron, 1 threads

ANY			MIN		
Time	Iterations	Time/iteration	Time	Iterations	Time/iteration
5444.35	22	247.47	6972.92	22	316.95

5.1.2 Weak scaling

In addition to strong scaling, weak scaling is another way of measuring a parallel programs efficiency. In this case the execution time is kept constant rather than the problem size. In our work however, our instances not only vary in size, i.e. edges and vertices in the graph, but also iteration count and minimum cut size. As such, this is not a metric we can measure with a degree of confidence.

5.1.3 Iteration count variance

Iteration count varied somewhat within the run of each data set. This is expected, as we have built in non-determinism in the algorithm through the use of the ANY-operator when we perform the BFS 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 parent in the BFS-DAG does not affect correctness. The original GBTL implementation's use of a MIN-operator instead (minimum of vertex indices) 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.

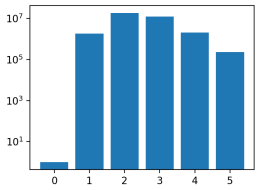
Lower iteration count is, naturally, tied to a lower total run-time, which means that a problem-specific heuristic could benefit the algorithm. Heuristics are utilized in other maximum-flow algorithms, and 'shortest path' is itself a heuristic that implies a known theoretical run-time bound, so it is not unthinkable.

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 can be seen in Table **??**, and the results are 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 each iteration consistently took longer than when using the ANY-operator, though this could also be a consequence of the blocking mode. The reduced number of iterations makes up for this though, and results overall in a reduced run-time.

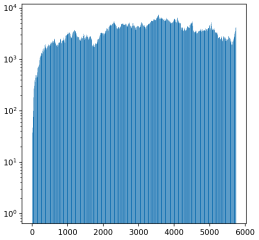
5.1.4 GAP-road

GAP-road is the same dataset as the US road set from the DIMACS implementation challenge [26], but with added edges weights. As a road network, it is a high-diameter graph with low variance in vertex degree. This means for a frontier-based/bulk-synchronous parallel approach, like the linear algebra approach is, the amount of parallelism at each level of a search will be low.

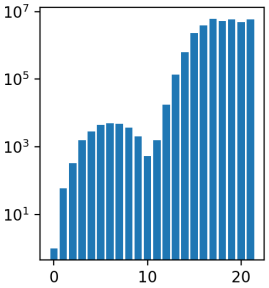
We can clearly see this manifesting in the results. While GAP-road is a smaller graph in number of edges by orders of magnitude than the other graphs, the time



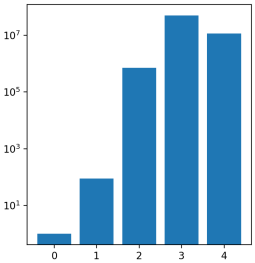
(a) GAP-twitter with log scale.



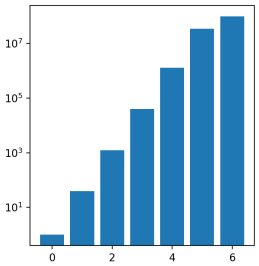
(b) GAP-road with log scale.



(c) GAP-web with log scale.



(d) GAP-kron with log scale.



(e) GAP-urand with log scale.

Figure 5.3: Frontier sizes. These are all taken from the first iteration of their respective run.

needed to process the max-flow problem instance is still considerable.

5.1.5 Profiling

In addition to measuring overall run-time for each configuration of dataset+thread count, we also measured specifically how much time the program spent in different sections of the code, as was seen in 5.2. As we can see from the results on the GAP-twitter and GAP-web datasets, the program spent significantly more time augmenting the flow along a given path than it did actually calculating that path, regardless of thread count.

The amount of work done in a single VxM operation, the basis for finding the augmenting path, is dependent on the number of multiplications between the frontier vector and the adjacency matrix. While we cannot observe this number directly, we can get an estimate by looking at the frontier sizes, as the size of the next level frontier is a lower bound on the number of multiplications. In Figure 5.3 we can see the frontier sizes for the first iteration for each problem instance.

The amount of work done when augmenting flow along a path is generally dependent on the length of the path. However, the graph with the longest paths, GAP-road, spent significantly more time in its runs in traversal than it did augmenting the flow. While pin-pointing why this is, it can be a consequence of how edges are deleted, which is to filter the residual graph through a mask, as seen in Section 5.3.2. If the mask is applied after rather than before the application of the unary function, then this operation would naturally infer very large amounts of computations on larger graphs. This optimization has been identified by the authors of another GraphBLAS implementation, GraphBLAST [8], and was described in Section 2.2.5.

Another consequence of this observation is how graph properties affect choice of algorithm. Dinic's algorithm and algorithm in that style generally have significantly lower theoretical runtime bounds than Edmonds-Karp. However, the way they achieve this is by essentially re-using the breadth-first search and augmenting the flow multiple times before searching again. As we see from our benchmarks, traversal was not the bottleneck for the largest datasets, and as such, it is doubtful we would have gained much implementing one of the more complicated algorithms, even if one assumes they could be parallelized efficiently.

The radical difference in where the bottleneck exists in the program between the GAP-road network and the power-law graphs and the GAP-urand graph suggests that there is not necessarily a single approach that can optimize this implementation.

5.2 Source-sink search

Our method for selecting the sink in the graphs, as outlined in section 4.2, produced sufficiently hard problem instances for the maximum flow problem. They did however, consistently produce instances with the distinct feature that the

placement of the sink was hardly affected by the dampening factor. This is likely consequence of the breadth-first search itself. If a vertex is visited, it will never be visited again. So, if we find a high in-degree vertex candidate for the sink and give it a high weight. Then, if it does not have any out-edges to un-visited vertices, no matter how we set the dampening factor, this vertex will likely stay the heaviest, as the weight accumulated will never propagate outward. In other words, it might be that accumulating weights, even with a dampening factor, lead to us favoring high-degree "dead ends" in the BFS DAG 2.1.4. This does not necessarily lead to bad problem instances, but indicates that a more robust method might find better ones.

Early experiments with depth instead of our current method consistently produced iteration counts

5.3 Usability evaluation

In addition to evaluating the performance of GraphBLAS, an important part of this project was to test the usability of the standard and specifically Suitesparse:GraphBLAS. In this section we will first discuss a few somewhat unintuitive behaviors in the standard. We will then give some concrete examples of how "intended behaviour" (in the form of pseudo-code) and GraphBLAS code look side by side. Finally, we will provide some thoughts on the portability aspect of GraphBLAS and the resources available, especially as they relate to the work done for this thesis.

5.3.1 Unintuitive behaviors

There are some unintuitive behaviours. Particularly, only functions that take a monoid or semiring will assign a value to the non-elements in the matrix, i.e the elements that would be treated as implicit zeros in normal arithmetic. For all other functions, including those that take binary operator, any calculation between an explicit value and an non-value will simply return the explicit value, bypassing the binary operator entirely. An example of this can be seen in the breadth-first search for augmenting paths in our algorithm, see Listing 5.1, where in a call to *GrB_apply*, a binary operator is provided. As the two vectors combined in this operation are non-overlapping, any binary operator could be provided, as it will only ever return a single value from either vector. The only function of this binary operator is to override that GraphBLAS operations by default replaces all values in the output vector/matrix if accumulation is not defined.

Code listing 5.1: Example of unintuitive use of GrB_apply()

```
1 //parent<frontier> = frontier
2 GrB_Vector_apply(parent_list, NO_MASK, GrB_PLUS_INT32,
3     GrB_IDENTITY_INT32, frontier, DEFAULT_DESC);
```

Another sometimes unintuitive, but powerful feature is that of the mask. The descriptor can be set to make the mask be either treated explicitly as a boolean

mask or structurally. It can also be set to use the patterns complement as a mask instead of the mask itself. All these features were utilized in our program. Of note is that after a single iteration of the Edmonds-Karp algorithm, there are at least one zero-vaue edge that needs to be removed for the algorihtm to behave correctly. What we did, which was borrowed from the original GBTL algorithm, was to overwrite the residual graph with its own values, but using itself as the mask. That is, an explicit zero in the residual would prevent that same edge from being saved.

We also made heavy use of different masks and accompanying descriptors when extracting the min-cut, as described in section 3.2.2. This routine is however also a good example of some limitations of the standard, which is discussed further in Section 5.3.2.

5.3.2 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 5.4, 5.5 and 5.7.

Traversal, 5.4, 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.

Deletion, 5.5, is also relatively compact. This does, however, change the structure of the algorithm. In order to express all the deletions at once, one needs to bundle them together in a mask and *then* call GrB_apply() (GrB_assign() would also work). It also demands that the user have a keen understanding of how the mask is evaluated, that implicit *and* explicit zeros in the mask indicates a non-write, and that by using a matrix with explicit zeros as a write mask on itself those explicit zeros would be deleted. It could also infer a run-time dependent on the number of values in the graph rather than the number of deleted edges. And if that isn't the case, it has the unfortunate consequence of obfuscating the performance of the operation, making the program harder to reason about.

The last example is the least intuitive, which is taken from the min-cut extraction part of our program, 5.7. From a graph perspective, what we want to express is that the min-cut is the set of edges that point "out" of the set of vertices that are reachable from the source in the residual graph. After determining the set of reachable vertices, we need to do two steps of traversal. Then we extract the indices, so we can extract the corresponding edge values. Finally we combine the extracted indices and values to construct a new graph containing only the min-cut edges. While the process is not much longer, at only two more lines of instructions, it is significantly harder to parse. What makes it especially difficult is the need to translate between vectors, matrices and their indices, where GraphBLAS normally

```
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);
```

Figure 5.4: 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);
```

Figure 5.5: Edge deletion. Pseudo-code and C/GraphBLAS code.

allows us to only care about these containers as a whole.

5.3.3 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, 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.

All GraphBLAS implementations also have a emphasis on separation of concerns [9] between the top-level interface and the back-end. This means that as long as the low-level implementations of the linear algebra algorithms are efficient across the different platforms, one can feasibly trust that a high-performing algorithm written in one GraphBLAS implementation will still perform well in other implementations, even on other platforms.

Figure 5.6: Extract min-cut

```
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);
```

Figure 5.7: Min-cut extraction. Pseudo-code and C/GraphBLAS code.

5.3.4 GraphBLAS resources

An incredibly useful resource when working on this project was the GraphBLAS reference API [4]. This is a comprehensive overview of the behaviour of all functions and object defined in the standard. As they provide mathematical definitions along with written descriptions, it reinforces the way the linear algebra abstraction works. This is what allowed us to make the changes between the GTBL code and our own, as well as implement our own algorithm for extracting the minimum cut.

In addition, as was mentioned in the background chapter, the LAGraph project [9] seeks to compile a variety of different algorithms written in the GraphBLAS standard. In addition to being useful in and of themselves, these programs can be a great asset when learning different techniques for how to translate a graph problem into a set of linear algebra operations.

5.3.5 Comparison to other methods of implementation

In terms of performance, there has been done research in comparing different graph algorithm frameworks, including by Azad et. al [25]. Here Suitesparse:GraphBLAS does not perform favorably when compared to other frameworks. They focus on performance, however, and not on ease of use.

In all our core program spans ~500 lines, including comments and debug

functionalitym as well as functionality for extracting the minimum cut. Excluding min-cut extraction, which as far as we are aware is not standard for other implementation, the total program spans a little less than 400 lines of code. The functionality for reading a matrix-market formatted graph adds another ∼100 lines. By comparison, the max-flow implementation present in the Lonestar/Galois repository is 870 lines long and Gunrock's is ∼1500 lines.

Another interesting point of comparison would be how easy to write and maintainable our maximum flow implementation is compared to one written "from scratch". While we will abstain from speculating too much in that regard, there are a couple of important points. Such an implementation would potentially be able to exhibit less "fork and join" behaviour. That is, if we have more fine-grained control over threads, we can keep them busy for longer before we need to explicitly wait for the other threads to finish, which could lead to more concurrent computations. However, in addition to the algorithm itself, we would have needed to build a system to queue up and manage the work and divide it between the threads. Considering the irregularity present in the algorithm, for example the wildly varying frontier-size, this would not be easy.

Chapter 6

Conclusions and Future Work

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 [2] also bore out.

The new functionality we implemented, in particular the minimum-cut extraction, demonstrated that the framework is clearly usable to build entirely new algorithms too, once the core logic is understood. This functionality did however push towards the limitations of the standard though, as this part of the program translated less neatly to linear algebra and as such was less intuitive.

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 this thesis. 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 constructed maximum-flow problem instances out of the GAP Benchmark Suite's set of reference data sets. This was done to expose the program to a representative selection of graph types, while keeping the number of experiments to a minimum. This strategy turned out to be essential, as some experiments were incredibly long-running.

Our experimental results were mixed. 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.

6.1 Future work

As we saw in the results, iteration count varied because of our non-deterministic algorithm. In the case of GAP-kron, running a deterministic variant of our algorithm meant the algorithm found the maximum flow in a minimal number of iterations. This warrants exploring further, whether this generalizes across other graphs and problem instances.

Where source and sink is placed in the graph can have huge consequences for the hardness of the problem instance. Maximum flow algorithms seem less viable to test in the same way other graph algorithms are tested. The GAP benchmark suite standardized 64 *sources* of its algorithms, but random source-sink pairs don't imply any sort of consistent workload. One solution could be to test enough pairs to even out the variance, similarly to how GAP provides a variety of sources, but this could quickly lead to infeasible run-times for experiments. Alternatively, one could exclusively look at domain specific problem instances, though this limits the validity of the experiment results to those domains.

It would be interesting to see whether the scalability pattern presented in our experiments would repeat across other data sets, instances and systems. This would require the logistical problem outlined above to be solved, of how to conduct such large scale experiments in a feasible manner. Though, as we saw the same pattern emerge across all our experiments, one could likely use smaller problem instances than we did in this work. Preferably one would compare the results to the scalability of other maximum-flow algorithms across the same instances, to show whether or not the pattern is a artifact of the algorithm, the system or the problem itself.

Gunrock [5] has been used to implement a push-relabel algorithm for maximum flow, an algorithm with a better theoretical run-time bound than Edmonds-Karp. While Gunrock is not based on linear algebra, it implies a very similar pattern of computation, which could indicate that such an algorithm *is* feasible in GraphBLAS too.

Bibliography

[1] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Mendez-Lojo, D. Prountzos and X. Sui, 'The Tao of Parallelism in Algorithms,' en, *PLDI'11*, 2011.

[2] A. Buluc, T. Mattson, S. McMillan, J. Moreira and C. Yang, 'Design of the GraphBLAS API for C,' en, in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Orlando / Buena Vista, FL, USA: IEEE, May 2017, pp. 643–652, ISBN: 978-1-5386-3408-0. DOI: 10.1109/IPDPSW.2017.117. [Online]. Available: http://ieeexplore.ieee.org/document/7965104/ (visited on 15/05/2020).

[3] T. A. Davis, 'SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra,' en, *ACM Transactions on Mathematical Software*, vol. 45, no. 4, pp. 1–25, Dec. 2019, ISSN: 0098-3500, 1557-7295. DOI: 10.1145/3322125. [Online]. Available: https://dl.acm.org/doi/10.1145/3322125 (visited on 25/03/2021).

[4] A. Buluc, T. Mattson, S. McMillan, J. Moreira and C. Yang, *The GraphBLAS C API Specification*, 2019. [Online]. Available: http://people.eecs.berkeley.edu/~aydin/GraphBLAS_API_C_v13.pdf (visited on 18/03/2021).

[5] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel and J. D. Owens, 'Gunrock: A high-performance graph processing library on the GPU,' en, in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming - PPoPP '16*, Barcelona, Spain: ACM Press, 2016, pp. 1–12, ISBN: 978-1-4503-4092-2. DOI: 10.1145/2851141.2851145. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2851141.2851145 (visited on 23/09/2019).

[6] H. M. Bücker and X. S. Li, Eds., *Parallel GraphBLAS with OpenMP*, en. Philadelphia, PA: Society for Industrial and Applied Mathematics, Jan. 2020, ISBN: 978-1-61197-622-9. DOI: 10.1137/1.9781611976229. [Online]. Available: https://epubs.siam.org/doi/book/10.1137/1.9781611976229 (visited on 25/03/2021).

[7] B. Brock, A. Buluc, T. G. Mattson, S. McMillan, J. E. Moreira, R. Pearce, O. Selvitopi and T. Steil, 'Considerations for a Distributed GraphBLAS API,' en, in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, New Orleans, LA, USA: IEEE, May 2020, pp. 215–

218, ISBN: 978-1-72817-445-7. DOI: 10.1109/IPDPSW50202.2020.00048. [Online]. Available: https://ieeexplore.ieee.org/document/9150368/ (visited on 05/05/2021).

[8] C. Yang, A. Buluc and J. D. Owens, 'GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU,' en, *arXiv:1908.01407 [cs]*, Sep. 2019, arXiv: 1908.01407. [Online]. Available: http://arxiv.org/abs/1908.01407 (visited on 06/05/2020).

[9] T. Mattson, T. A. Davis, M. Kumar, A. Buluc, S. McMillan, J. Moreira and C. Yang, 'LAGraph: A Community Effort to Collect Graph Algorithms Built on Top of the GraphBLAS,' en, in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Rio de Janeiro, Brazil: IEEE, May 2019, pp. 276–284, ISBN: 978-1-72813-510-6. DOI: 10.1109/IPDPSW.2019.00053. [Online]. Available: https://ieeexplore.ieee.org/document/8778338/ (visited on 22/03/2021).

[10] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir and K. Pingali, 'Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics,' en, in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2018*, Philadelphia, PA, USA: ACM Press, 2018, pp. 752–768, ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192404. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3192366.3192404 (visited on 15/01/2020).

[11] S. Beamer, K. Asanovic and D. Patterson, 'Direction-optimizing Breadth-First Search,' in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ISSN: 2167-4337, Nov. 2012, pp. 1–10. DOI: 10.1109/SC.2012.50.

[12] P. Zhang, M. Zalewski, A. Lumsdaine, S. Misurda and S. McMillan, 'GBTL-CUDA: Graph Algorithms and Primitives for GPUs,' en, in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, Chicago, IL, USA: IEEE, May 2016, pp. 912–920, ISBN: 978-1-5090-3682-0. DOI: 10.1109/IPDPSW.2016.185. [Online]. Available: http://ieeexplore.ieee.org/document/7529957/ (visited on 11/05/2021).

[13] Z. Svela, 'Evaluating multi-core graph algorithm frameworks,' en, *NIKT 2020*, p. 12, 2020. [Online]. Available: https://ojs.bibsys.no/index.php/NIK/article/view/829.

[14] S. Pai and K. Pingali, 'A compiler for throughput optimization of graph algorithms on GPUs,' en, in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2016*, Amsterdam, Netherlands: ACM Press, 2016, pp. 1–19, ISBN: 978-1-4503-4444-9. DOI: 10.1145/2983990.2984015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2983990.2984015 (visited on 18/06/2020).

[15] G. Gill, R. Dathathri, L. Hoang, A. Lenharth and K. Pingali, 'Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms,' en, in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani and M. Torquati, Eds., vol. 11014, Cham: Springer International Publishing, 2018, pp. 249–264, ISBN: 978-3-319-96982-4 978-3-319-96983-1. DOI: 10.1007/978-3-319-96983-1_18. [Online]. Available: http://link.springer.com/10.1007/978-3-319-96983-1%5C_18 (visited on 04/03/2020).

[16] 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. 1124–1137, Sep. 2004, Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, ISSN: 1939-3539. DOI: 10.1109/TPAMI.2004.60.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd. The MIT Press, 2009, ISBN: 0-262-03384-4.

[18] R. K. Ahuja, M. Kodialam, A. K. Mishra and J. B. Orlin, 'Computational investigations of maximum flow algorithms,' en, *European Journal of Operational Research*, vol. 97, no. 3, pp. 509–542, Mar. 1997, ISSN: 03772217. DOI: 10.1016/S0377-2217(96)00269-X. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S037722179600269X (visited on 11/06/2021).

[19] A. V. Goldberg, 'A New Approach to the Maximum-Flow Problem,' en, *Journal of the Association for Computing Machinery*, p. 20, Oct. 1988.

[20] E. A. Dinic, 'Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation,' 1970.

[21] V. Malhotra, M. Kumar and S. Maheshwari, 'An O(|V|3) algorithm for finding maximum flows in networks,' en, *Information Processing Letters*, vol. 7, no. 6, pp. 277–278, Oct. 1978, ISSN: 00200190. DOI: 10.1016/0020-0190(78)90016-9. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0020019078900169 (visited on 01/03/2021).

[22] D. G. Spampinato, U. Sridhar and T. M. Low, 'Linear algebraic depth-first search,' en, in *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming - ARRAY 2019*, Phoenix, AZ, USA: ACM Press, 2019, pp. 93–104, ISBN: 978-1-4503-6717-2. DOI: 10.1145/3315454.3329962. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3315454.3329962 (visited on 09/06/2021).

[23] J. H. Reif, 'Depth-first search is inherently sequential,' en, *Information Processing Letters*, vol. 20, no. 5, pp. 229–234, Jun. 1985, ISSN: 00200190. DOI: 10.1016/0020-0190(85)90024-9. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0020019085900249 (visited on 09/06/2021).

[24] S. Beamer, K. Asanović and D. Patterson, 'The GAP Benchmark Suite,' en, *arXiv:1508.03619 [cs]*, May 2017, arXiv: 1508.03619. [Online]. Available: http://arxiv.org/abs/1508.03619 (visited on 03/05/2021).

[25] A. Azad, M. M. Aznaveh, S. Beamer, M. Blanco, J. Chen, L. D'Alessandro, R. Dathathri, T. Davis, K. Deweese, J. Firoz, H. A. Gabb, G. Gill, B. Hegyi, S. Kolodziej, T. M. Low, A. Lumsdaine, T. Manlaibaatar, T. G. Mattson, S. McMillan, R. Peri, K. Pingali, U. Sridhar, G. Szarnyas, Y. Zhang and Y. Zhang, 'Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite,' en, in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, Beijing, China: IEEE, Oct. 2020, pp. 216–227, ISBN: 978-1-72817-645-1. DOI: 10.1109/IISWC50251.2020.00029. [Online]. Available: https://ieeexplore.ieee.org/document/9251247/ (visited on 03/05/2021).

[26] *10th DIMACS Implementation Challenge - Graph Partitioning and Graph Clustering*, 2012. [Online]. Available: https://www.cc.gatech.edu/dimacs10/index.shtml.

Appendix A

NIKT2020 Paper By Author

The following paper, "Evaluating multi-core graph algorithm frameworks", was presented at the NIKT2020 conference by the author. This was based on the fall pre-project which this thesis is an extension of. It can also be accessed on NIK 2020's web pages: `https://ojs.bibsys.no/index.php/NIK/article/view/829`

Evaluating multi-core graph algorithm frameworks

Zawadi Svela
Department of Computer Science
Norwegian University of Science and Technology
z.b.svela@gmail.com

Abstract

Multi-core and GPU-based systems offer unprecedented computational power. They are, however, challenging to utilize effectively, especially when processing irregular data such as graphs. Graphs are of great interest, as they are now used to model geographic-, social- and neural networks. Several interesting programming frameworks for graph processing have therefore been developed these past few years.

In this work, we highlight the strengths and weaknesses of the Galois, GraphBLAST, Gunrock and Ligra graph frameworks through benchmarking their single source shortest path (SSSP) implementations using the SuiteSparse Matrix Collection. Tests were done on an Nvidia DGX2 system, except for Ligra, which only provides a multi-core framework. D-IrGL, built on Galois, also provided a multi-GPU option for SSSP. We also look at program size, documentation and overall ease of use.

High performance generally comes at the price of high complexity. D-IrGL shows its strength on the very largest graphs, where it achieved the best run-time, while Gunrock processed most other large sets the fastest. However, GraphBLAST, with a relatively low-complexity interface, achieves the greatest median throughput across all our test cases. This despite that its SSSP implementation size is only 1/10th of Gunrock, which for our tests has the highest peak throughput and the fastest run-time in most cases. Ligra had less computational resources available, and consequently performed worse in most cases, but it is also a very compact and easy to use framework. Futher analyses and some suggestions for future work are also included.

1 Introduction

Graphs are an ever useful modelling tool for modelling data in countless of situations, including geographic-, social-, and neural networks. The demand for processing is increasing, but Moore's Law is not keeping up. Thus, highly parallel programs are needed for large-scale graph processing. GPUs (Graphical Processing Units) are also growing in popularity, as they have a much higher throughput potential than CPU-only systems.

Exposing parallelism in irregular algorithms, such as graph algorithms, can be difficult, and this challenge is increased when working with GPUs, which favors highly regular computations. Specialized libraries and frameworks can help in this regard.

In this work, we study four different graph algorithm frameworks, using their SSSP implementations as a reference point. Galois [1], Gunrock [2] and Ligra [3] were

This paper was presented at the NIK-2020 conference; see http://www.nik.no/.

chosen because they are well-known frameworks in the area of graph algorithms, and the GraphBLAS [4] implementation GraphBLAST [5] was added because of its novel programming model. In addition, they also represent a variety of complexity levels. From the larger Galois project, we looked at the D-IrGL [6] system specifically, which targets heterogeneous and distributed systems. Here, "Galois" will refer to the framework and the programming model, while "D-IrGL" will refer to the code that was run and tested.

SSSP was chosen for our benchmarks since it is both an important graph primitive which is simple to outline and because it has room for many implementation differences. The specific algorithm can thus play to the strengths of frameworks, regardless of the programming model they use.

The goal of this work is to give a comparison of four different frameworks, in order to help evaluate the trade-offs between both system complexity (CPU/GPU/multi-GPU) and the frameworks that facilitate programming for these devices.

Related works include a taxonomy and categorization of different parallel graph algorithm frameworks by Heidari et al. [7]. Petterson's [8] survey is similar to our own, but with a a different primitive and selection of frameworks. Both of these works included Gunrock and Galois in their comparisons, but not D-IrGL, Ligra or GraphBLAST.

2 Background

The following sections introduce the four main frameworks benchmarked in this study: Galois [1], Gunrock [2], Ligra [3] and GraphBLAS [4]. We also include some of the abstractions used by the frameworks we evaluated, including the operator formulation used by Galois, the breath-first-based Frontier-based view associated with Gunrock and Ligra, which is also implicitly used in GraphBLAS, as well as some of the basic linear algebra terminology associated with graph representations.

Galois

Galois is a framework built around Tao-analysis, [1], and uses the *operator formulation* of algorithms to reveal *amorphous data-parallelism*, see Section 3. As such, the key features Galois provides are concurrent data structures and parallel loops that allow generation of new work items/loop iterations as the loop is running. They also provide support for priority and ordering between work items.

D-IrGL is one of the more recent additions to the continually growing family of systems associated with Galois. It is the combination of the communication substrate Gluon [6] with the GPU-targeted intermediate representation IrGL [9], and it facilitates distributed heterogeneous graph applications. These ideas were developed further into the Abelian compiler [10], which produces Gluon+IrGL code from ordinary Galois code written for CPU. The complete system effectively turns Galois into a portable framework for multi-core, distributed and potentially heterogeneous graph algorithms.

SSSP implementation. D-IrGL uses the *Chaotic Relaxation* algorithm for SSSP [11]. The source vertex begins as active, relaxes the labels of its neighbours and activates them. Now, in any order, the same procedure is performed on active vertices, relaxing labels and activating new vertices as necessary, until the algorithm converges. Because D-IrGL computes in a bulk-synchronous parallel manner, the pattern of computations would resemble a parallel Bellman-Ford implementation like in Ligra or GraphBLAST.

Gunrock

Gunrock is a frontier-based framework (see Section 3) for single-node GPU accelerated systems. In its original implementation, it provided three functions for processing frontiers: *Advance* for traversal, *compute* for label updates and *filter*. A few more specialized operators where added later.

SSSP Implementation. Gunrock uses two operators for its SSSP implementation, *advance* and *filter*. When advancing, they use an atomic minimum function to update the target vertex distance, and *filter* removes any redundant vertices from the frontier. They also utilize a two-level priority queue to process more relevant vertices will be processed first when the frontier size grows large. An older version of Gunrock has multipe-GPUs support for SSSP, but we decided to focus on its recent version which has a new programming model, but only has single-GPU support for SSSP.

Ligra

Ligra is a light-weight frontier-based framework for CPUs. It provides only two functions, *edgeMap* and *vertexMap* and a *vertexSubset* type for the frontier. EdgeMap is the advancing operator, and applies a given function to all edges out of the frontier given a condition is met. VertexMap is similar to Gunrock's *filter* function. Both functions have precise mathematical definitions provided in the paper, and as a consequence Ligra is a very compact framework, which is explored further in Section 6.

SSSP implementation. Ligra implements a straight-forward Bellman-Ford algorithm. As with any frontier-based framework, it will only relax edges incident to the vertex frontier, in contrast all edges as in the original Bellman-Ford algorithm.

GraphBLAS

GraphBLAS [4] is an open standard for sparse linear algebra operations targeting graph algorithms. Inspired by the BLAS standard, it seeks to be portable and specifies the semiring structure, see Section 3, as well as different linear algebra operations such as matrix-vector multiplication and element-wise addition. There is a reference implementation in SuiteSparse, as well as mutliple other implementations.

GraphBLAST

GraphBLAST [5] is a implementation of GraphBLAS targeting Nvidia-GPUs. It uses an extension of a technique employed in a BFS-implementation by Beamer *et al.* [12], who observed that when the frontier of a breadth-first search (BFS) is large enough, it is beneficial to pull information from the vertices outside the frontier rather than the frontier vertices pushing the search forward. The dichotomy between pull and push-style computation matches dense and sparse vector-matrix multiplication, and not just for BFS. This is exploited in GraphBLAST, and the authors call this technique *generalized direction-optimization*.

SSSP Implementaion. GraphBLAST's SSSP algorithm is similar to that of Ligra, as linear algebra exhibits a very similar compuitational pattern. In addition to direction-optimization, they employ *sparsification* of the frontier. An element-wise minimum-operation is performed between the the frontier and the distance label vector to rid the frontier of any vertices that did not see any improvement in their distance.

3 Abstractions

This section will present the different abstractions utilized by the frameworks. There are many different abstractions for exposing parallelism in graph programs. They are given high importance here as they are how the user conceptualizes and models their algorithm. They are also interesting to look at separately because they might be implemented differently by different frameworks.

Operator formulation

Used by Galois and key in Tao-analysis [1], the operator formulation views an algorithm as actions on a data structure. The formulation is general enough to apply to any structure, but the primary goal is to reveal parallelism in irregular computations such as graph algorithms, so that will be the focus here.

An element in a graph ready for computation is called *active*. The elements read or written to by an activity is called the *neighbourhood*, and is key to performing activities in parallel, as they are the potential source of race conditions. Algorithms were activities can activate new elements are called *data-driven*.

A benefit to this model is how it decouples the generation of activities from any ordering or dependencies they might have. It exposes what Pingali *et al.* refers to as *amorphous data-parallelism*, and if one ensures that no computation is committed until a safe state is reached, possibly by issuing roll-backs, activities can be performed in parallel and in any order. This leaves room for many optimizations "behind the scenes" that the user of this kind of system does not need to interact with directly.

Frontier-based

One of the most popular abstraction is the frontier, which is based on the pattern of a breadth-first search. The frontier represents the active elements in the graph, usually vertices, and algorithms are constructed by iteratively applying operators on the frontier until it empties/converges. The most important operator is one that advances the frontier, producing a new frontier with neighbours from the previous one. In addition, one needs the possibility of updating internal values of vertices and filter out frontier elements. For many algorithms, this would yield a procedure like in Figure 1, the pattern of a breadth-first-like search. Among those frameworks studied in this work, this abstraction is used by Gunrock, Ligra and implicitly in GraphBLAST.

```
Initialize frontier
while frontier not empty do
    Advance frontier to neighbours
    Update label on newly discovered vertices
    Filter out previously discovered vertices
end while
```

Figure 1: Outline of frontier-based algorithms.

Parallelism in this abstraction is available in all operators. As noted by Wang *et al.* [2], the advance operator presents the most challenges, as it is irregular, some vertices can have vastly different degrees than others.

Linear Algebra

If a graph is represented as a matrix, many graph traversal operations similar to the ones mentioned in Section 3 can be described in the language of linear algebra. Given an adjacency matrix A and a vector representing a frontier x, the matrix-vector product $A^T x$ produces the same pattern of operations as advancing through the graph in a frontier-based graph framework. One does, however, need to adjust the actual operators used, which is expressed through semirings.

A semiring is an algebraic structure consisting of an additive operator, a multiplicative operator and a domain, as well as "zero" and identity ("one") values. Examples of different semi-rings can be found in Table 1, along with what kind of primitive a matrix-vector multiplication would imply. This is perhaps the most important feature of this abstraction. By using semirings, it allows for re-use of patterns and optimizations from linear algebra when constructing graph algorithms.

Table 1: A few example semirings.

Result	operators $\bigoplus$	operators $\bigotimes$	domain	0	1
Standard arithmetic	$+$	$\times$	R	0	1
Breadth-first search	or	and	$\{0,1\}$	0	1
Single source shortest path	*min*	$+$	$\{-\infty, R\}$	$-\infty$	0

4 Benchmarking the Frameworks

In order to evaluate and compare the performance of the different frameworks, we ran their SSSP implementations with a suite of different data sets ranging from 64 to 4,294,966,740 edges. These sets span a variety of different domains with different graph characteristics. This section describes our test set-up, the methodology behind the experiments as well as the results we obtained. Arbitrarily, the results are ordered Gunrock, D-IrGL, GraphBLAST, then Ligra, an artifact of how the tests were run.

Testbed Systems

All GPU based frameworks were tested on a Nvidia DGX-2 node. This node contains two boards of 8 V100 GPUs each. Each board also contains 6 NVSwitch interconnects.

Ligra, the only CPU-based framework tested, was tested on a Supermicro SuperServer 6049GP-TRT with 2 CPUs, Intel Xeon Gold 6230 SP - 20-Core. All cores were capable of hyperthreading, and all of the 80 available threads were utilized for the experiments.

Price comparison. As a point of reference, the CPU resources used by Ligra has a standalone price of ~4.000USD at the time of writing and a single V100 GPU, which most configurations of the other programs used, has a price of ~8.000USD.

Bechmarking Methodology

One SSSP implementation was compiled from each framework and used to process 128 different graphs. Run-time reported from each program excluded data-movement. Each program was run at least 3 times for each graph to overcome variance and the best time was considered for comparisons and evaluation.

A single configuration was used for each program, except for D-IrGL which was tested on multiple GPUs. We indicated to the programs whether the graph was undirected, but

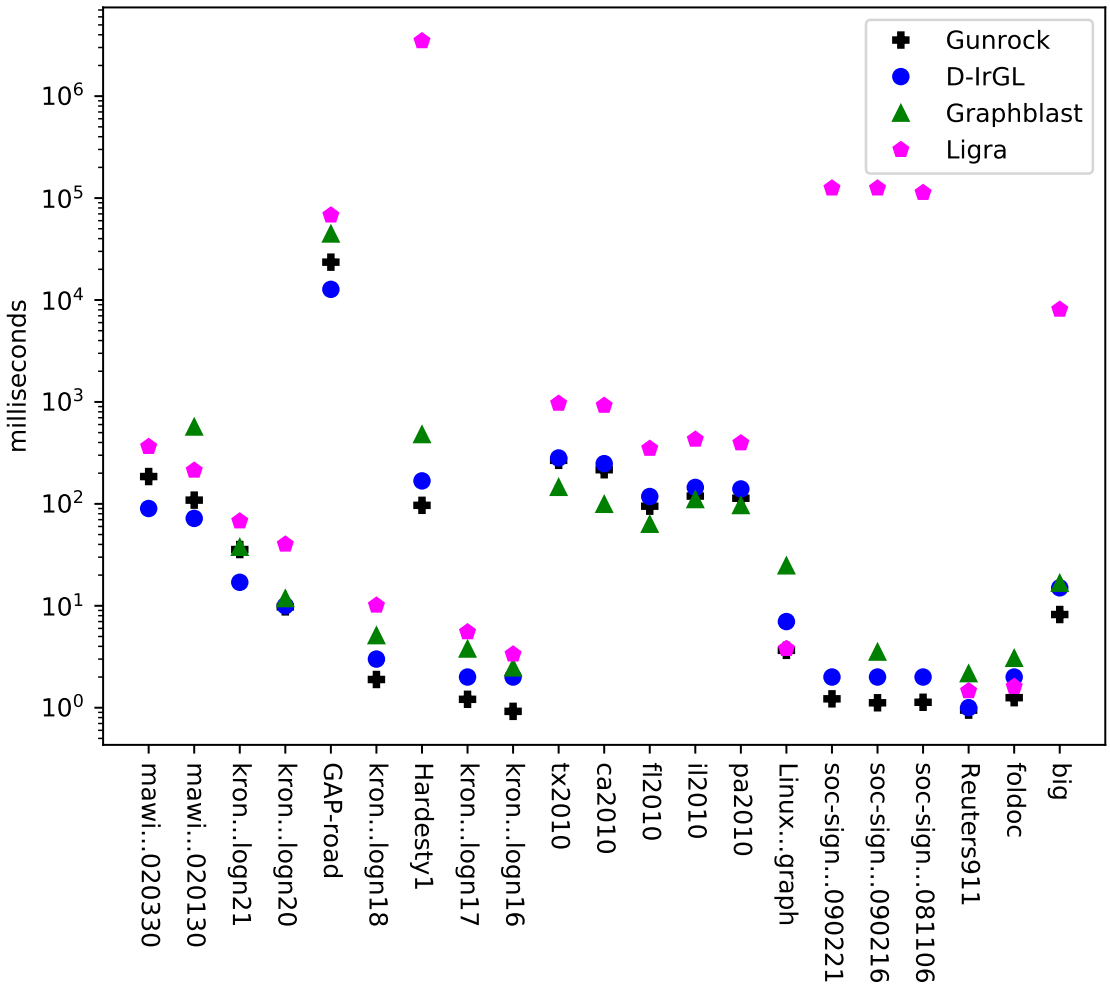


Figure 2: Runtime in milliseconds for the largest graphs with $> 10,000$ visited vertices. Only the 5 largest "xx2010" are shown.

Table 2: Number of best times per framework.

Gunrock	56	D-IrGL (Galois)	4
Graphblast	47	Ligra	5

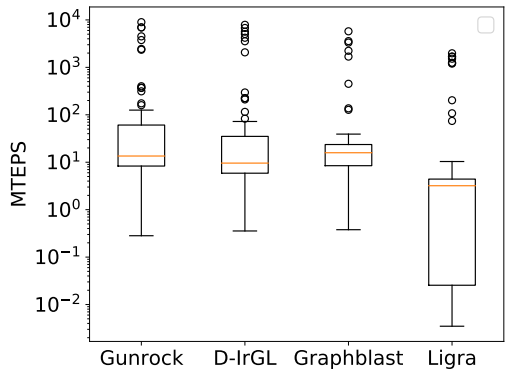
not any other graph characteristics. Gunrock reports the number of visited vertices for each graph, and this was used to filter results and estimate throughput.

Data sets

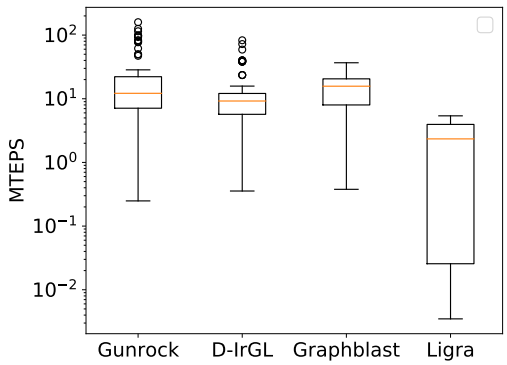
All data sets were pulled from the SuiteSparse Matrix collection [13]. Only square integer matrices with the keyword "weighted" were considered. Ligra does not support floating point values, and matrices representing unweighted graphs would not be suited for comparing SSSP implementations. This left 128 graphs.

5 Benchmarking results

This section presents and evaluates the results of the experiments. Two metrics were deemed the most relevant, run-time on large graphs, and average throughput.



(a) Visited vertices >= 30. 110 out of 128 graphs.



(b) Complete traversal only. 83 out of 128 graphs.

Figure 3: Millions of traversed edges per second estimates for different subsets of the results. Graphs where Gunrock crashed are excluded.

Table 3: Peak throughput for each program.

Program	Peak Throughput
Gunrock	8969
Galois	7885
GraphBLAST	5754
Ligra	1980

Relative run-time

Figure 2 shows a selection of the running time for the different frameworks on largest graphs with more than 10,000 visited vertices. The general trend ranking the relative performance is Gunrock, D-IrGL, GraphBLAST and then Ligra. This is not wholly unexpected. Gunrock is specifically developed specifically for high performance and for single-node systems. D-IrGL is also developed for high-performance, but for potentially distributed systems. GraphBLAST and Ligra are light-weight frameworks with a stronger focus on accessibility, and additionally Ligra only runs on CPU and therefore had less computational resources available than the GPU-based programs.

However, as can be seen in Table 2, the aforementioned trend is not universal. D-IrGL outperforms Gunrock in four of the five largest graphs. Among these there are both scale-free graphs and a road network, so results does not seem to be tied to specific properties of the graphs. Ligra also outperformed Gunrock on some smaller graphs (less than 10,000 edges), as well as the "geom" data set with approximately 20,000 edges, which for context is roughly 1/4th the size of the smallest graph present in Figure 2. Lastly, GraphBLAST outperformed the other programs in a significant number of cases, which is further discussed in Section 5.

Throughput

In addition to runtime on specific graphs, we wanted to measure the throughput efficiency of each framework to get a somewhat normalized metric to compare the programs by. We used an estimate of millions of edges traversed per second (MTEPS) as our throughput metric, visualized in the boxplots of Figure 3. The number of edges processed by each program is estimated based on the number of visited vertices reported by Gunrock, which

would be the same for any SSSP algorithm, and the average degree of the graph. This is not a perfect estimate, as different programs process different amounts of edges, and the average degree might not reflect the connected component explored, but it still gives a sense of how efficient the programs are in relation to the graph size.

Using this metric, one can see that GraphBLAST actually outperforms the rest in terms of its median MTEPS across all graphs, Figure 3a. This is even clearer when looking at only complete traversals 3b. If we only look at peak throughput, however, see Table 3, Gunrock and Galois again take the lead, with peaks that are respectively 56 and 37 percent better than that of GraphBLAST. Ligra has a very high variance in its throughput. It is unclear what causes this variance, and it might either be hardware/CPU specific or caused by Ligra'a execution model.

Graph characteristics

In general, the graphs represented in the data sets seem to span a few staple categories of graph analysis, which allows us to test the generality of the programs. There are many scale-free graphs, both synthetic and real, and there are road-networks and mesh-like graphs. Scale-free graphs or power-law graph are frequently used as test cases and cited as challenges, [2,3,5,8], which makes sense as they have highly skewed degree distribution.

Of the 110 graphs with valid results, 50 are "xx2010" graphs from the DIMACS10 collection. These are based on US census data, and vertices and edges model respectively census blocks and their borders. These graphs have a high diameter and an even distribution of degrees, in contrast to scale-free networks that generally have a highly skewed distribution and low diameter. Interestingly, removing the "xx2010" graphs increased the median throughput of both Gunrock and D-IrGL, but the opposite was the case for GraphBLAST and Ligra. The reason for this is unknown. GraphBLAST achieved the best run-time on 47 of these graphs, but not on any other sets, which might indicate these graphs have attributes especially suited for this particular framework.

D-IrGL and multi-GPU

D-IrGL is developed for potentially distributed systems with multiple GPUs, and was tested on both 1, 2, 4 and 8 GPUs. The performance worsened with multiple GPUs compared to one, but it did manage to load the GAP-kron graph with 4 and 8 GPUs, a graph that caused all other programs and configurations to run out of memory. Because of this, large graphs is a use-case for these configurations, even if the throughput is lowered.

6 Evaluation

A framework is a toolkit for constructing new programs. As such, its quality is not just dependent on its efficiency, but also its usability. This section contains a comparison of code sizes as well as a subjective assessment of the ease of use of the different frameworks. The focus is on the available documentation, including papers published in relation to the frameworks. A summary of this section can be found in Table, 4.

Framework code sizes

The total lines of code for the different SSSP implementations are provided in Table 5. This is an important metric, as large code bases are both harder to understand, and also harder to maintain. In all cases the programs have additional features such as support for multiple runs and timing, but they still give an indication of how easy it is to understand

Table 4: Accessibility summary

	Code size	Documenation	System
D-IrGL	Very large	N/A	Multi-GPU
Gunrock	Very large	Fair	Single node, multi-GPU
GraphBLAST	Small	Good	Single GPU
Ligra	Very small	Good	Multicore CPU

Table 5: Lines of code per SSSP implementation, including per file, by code size.

Ligra	87	BellmanFord.C
GraphBLAST	138	gsssp.cu
Gunrock	293	sssp_app.cu
	361	sssp_enactor.cuh
	408	sssp_problem.cuh
Total	1062	
D-IrGL	944	sssp_push_cuda.cu
	216	sssp_push_cuda.cuh
	147	sssp_push_cuda.h
Total	1307	
Galois	494	SSSP.cpp

the given program. From these counts, it is clear that Ligra and GraphBLAST are by far the most compact frameworks, which should make them easier to understand than the others. Gunrock lies as the other extreme with large file sizes and also multiple files. D-IrGL code is constructed by the Abelian compiler and as such isn't necessarily meant to be read or interacted with directly, but the Galois CPU-implementation which D-IrGL is based on, still has a fair number of lines in its code.

Documentation

Documentation is important especially frameworks. The perspective of this assessment is after a thorough reading of relevant papers describing the frameworks, any tutorials available, and a scan of available manuals. The developers of all 4 frameworks were a great help clearing up any confusion throughout the process of this evaluation, which goes a long way for compensating for ways in which documentation might be lacking.

Galois

The online documentation of Galois contains both a comprehensive manual and a step-by-step tutorial on how to create a simple program. As Galois is a pretty complex framework, this helps in easing into the necessary concepts.

It does not incorporate all features of Tao-analysis, most notably all operators are assumed cautious. This means that ethey can not write to a data structure before executing all necessary reads. This allows Galois to use locks to detect conflicts before an operation is committed, which again means it does not need to execute any rollbacks.

D-IrGL The documentation of how to run the D-IrGL programs is good, and there were no major issues encountered. However, there is no documentation available on how to write D-IrGL programs.

GraphBLAST

GraphBLAST has minimal documentation in comparison to Gunrock and Galois. There is a single text-file in their Git repository, which contains an example SSSP implementation with comments, and a short explanation of their linear algebra operations, as well as semirings and how they relate to graph algorithms.

The GraphBLAST documentation assumes the user is comfortable with linear algebra operations and how it relates to graph algorithms. Reading the GraphBLAST paper is thus recommended to understand all the sample algorithms and make one's own programs.

Gunrock

In general, the presentation of Gunrock is quite excellent. They have an easily navigable website that provides an overview of the programming model, performance analysis, and links to publications and their GitHub code repository with notes on the state of the current build. They also provide a quick start guide for building and running the algorithm implementation they provide. Also beneficial is a thorough description of exactly what is measured and reported for the different programs, which aids benchmarking.

At the time of writing, Gunrock has recently transitioned from release 0.5 to release 1.0, which saw a great simplifications of the interface. It does however, also mean that the online documentation isn't up to date yet. Notably, there is no tutorial available on how to write new primitives with Gunrock. As Gunrock is a quite complex system, it can be quite difficult to make wholly new programs from scratch at this point. A guide for porting 0.5 application is provided, which can help, and they also provide steps to generate the newest documentation locally. The developers are fully aware of these features currently lacking. They are all described in their online roadmap under "Documentation" [1].

Ligra

Similarly to GraphBLAST, Ligra's light interface allows it to have minimal documentation. It simply presents the input format, how to get the programs up and running, and provides a short description of the functions and data structures. It is very beneficial to have read the paper presenting Ligra first, [3], because the mathematical function definitions in the paper match up quite well with the actual code. There is no tutorial-specific code provided, but the code for BFS and SSSP (Bellman-Ford) are both quite short.

7 Conclusions & Future Work

Graphs are used to model a wide variety of different data. With the increasing computational demand and system complexity, recent graph frameworks provide attractive tools for implementing graph-based methods that can be very beneficial for productivity. In this paper we presented four of the most well-known frameworks for graph algorithms, with SSSP used as the reference point to evaluate both the ease of use and their performance. All frameworks showed strengths in different use-cases. A summary of our results is followed by suggestions for future work.

If high-performance is of top priority, Gunrock had the most reliable performance, and achieved the best run-time on 56 out of 128 graphs, which included multiple different graph types. D-IrGL was generally slightly slower than Gunrock and had the best run-time on only 4 graph. However, these graphs were among the largest we tested, and D-IrGL

[1] https://github.com/gunrock/gunrock/projects/3

does in theory support arbitrarily large graphs when run on multi-device systems. D-IrGL code is not as easily re-usable, a definite weakness compared to the other programs. Gunrock also achieved the best peak throughput across all frameworks, but it is also the most complex framework out of the four. It is also currently lacking in terms of documentation of version 1.0. In addition, it's SSSP implementation indicates both high complexity and low maintainability compared to the lighter frameworks.

GraphBLAST performed surprisingly well for a relatively light-weight framework, even ignoring a large set of similar graphs where it actually outperformed all other frameworks. It achieved the best run-time on 47 graphs and had the overall best median throughput. Compared to Gunrock, it is also significantly less complex, with easier to understand documentation and an SSSP implementation of approximately 1/8th the size. It and Gunrock did however crash on a few data sets, which is a definite negative.

Ligra used ~100,000 the time of Gunrock on certain graphs, which makes it less appealing as a general case high-performance framework. Its ease of use and compactness, with SSSP code size 1/12th the size of Gunrock's, does however make appealing for pipelines with high maintainability demands or for smaller graphs.

Future work

For a more thorough comparison, one could write the same primitive in different frameworks, preferably solving a new problem rather than those already provided. Writing the exact same algorithm would not necessarily be the correct approach, as different frameworks excel at expressing different behaviours, like there were four different algorithms implemented for SSSP. Future comparisons should also include floating data sets, as this could dramatically increase the variety of the data sets.

It would have been beneficial to have a standardized suite with detailed descriptions of relevant attributes of all graphs. Two such attributes would be degree distribution and diameter, as they can greatly affect the run-time of a program.

It would also be interesting to explore exactly why GraphBLAST performed so well specifically on the "xx2010" graphs. As discussed in Section 5, this is probably related to some common attributes across these graphs, and finding the exact ones might reveal new optimization strategies for both GraphBLAST and other graph algorithms.

Acknowledgements: The author wishes to thank their advisor Prof. Anne C. Elster for helping with this write-up, and NTNU and the IDI HPC-Lab for computer resources.

References

[1] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Mendez-Lojo, D. Prountzos, and X. Sui, "The Tao of Parallelism in Algorithms," *PLDI'11*, 2011.

[2] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "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*. Barcelona, Spain: ACM Press, 2016, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2851141.2851145

[3] J. Shun and G. E. Blelloch, "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. [Online]. Available: https://people.csail.mit.edu/jshun/ligra.pdf

[4] A. Buluc, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the GraphBLAS API for C," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Orlando / Buena Vista, FL, USA: IEEE, May 2017, pp. 643–652. [Online]. Available: http://ieeexplore.ieee.org/document/7965104/

[5] C. Yang, A. Buluc, and J. D. Owens, "GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU," *arXiv:1908.01407 [cs]*, Sep. 2019, arXiv: 1908.01407. [Online]. Available: http://arxiv.org/abs/1908.01407

[6] R. Dathathri, G. Gill, L. Hoang, H.-V. Dang, A. Brooks, N. Dryden, M. Snir, and K. Pingali, "Gluon: a communication-optimizing substrate for distributed heterogeneous graph analytics," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2018*. Philadelphia, PA, USA: ACM Press, 2018, pp. 752–768. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3192366.3192404

[7] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges," *ACM Computing Surveys*, vol. 51, no. 3, pp. 1–53, Jul. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3199523

[8] H. Pettersson, "A Survey of Parallel Breadth-First Search Frameworks for CPUs and GPUs," *Master Thesis at NTNU*, p. 41, 2018.

[9] S. Pai and K. Pingali, "A compiler for throughput optimization of graph algorithms on GPUs," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2016*. Amsterdam, Netherlands: ACM Press, 2016, pp. 1–19. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2983990.2984015

[10] G. Gill, R. Dathathri, L. Hoang, A. Lenharth, and K. Pingali, "Abelian: A Compiler for Graph Analytics on Distributed, Heterogeneous Platforms," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, vol. 11014, pp. 249–264. [Online]. Available: http://link.springer.com/10.1007/978-3-319-96983-1_18

[11] D. Chazan and W. Miranker, "Chaotic relaxation," *Linear Algebra and its Applications*, vol. 2, no. 2, pp. 199–222, Apr. 1969. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0024379569900287

[12] S. Beamer, K. Asanovic, and D. Patterson, "Direction-optimizing Breadth-First Search," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2012, pp. 1–10, iSSN: 2167-4337.

[13] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011, place: New York, NY, USA Publisher: Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/2049662.2049663

Appendix B

Additional Code Listings

This section contains the listings with code written in C. In all cases, to save space, variable instantiation and memory allocation has been removed. Mathematical pseudo-code notation has been inserted to correspond with the respective listings in 3, where the concrete behaviour of these code snippets are more thoroughly explained. NO_MASK, NO_ACCUM and DEFAULT_DESC all correspond to GrB_NULL, which indicate default behaviour. A null mask or descriptor means output and input are read as is. No accumulative operator means all values written replace the respective values present in the output object beforehand.

Code listing B.1: The implementation of the Edmonds-Karp algorithm. Mathematical notation corresponds to the outline in Listing 3.1.

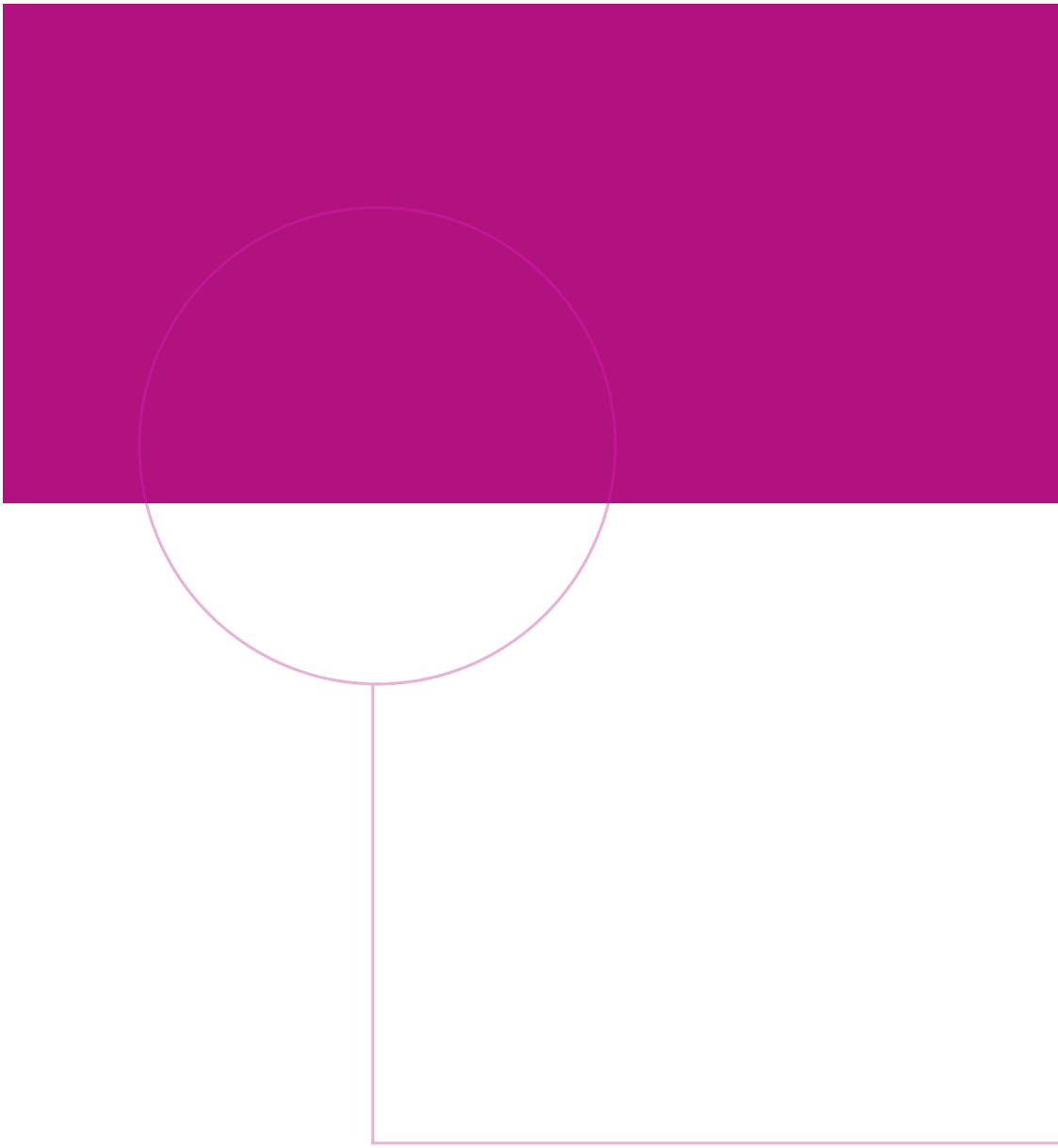
```
//while(M = get_augmenting_path(R, s))
while(get_augmenting_path(R, source, sink, M))
{
    //P = M ⊗ R
    GrB_eWiseMult(P, NO_MASK, NO_ACCUM, GrB_SECOND_FP64, M, R, DEFAULT_DESC);
    /* Alternative:
    //P<M> = R
    GrB_assign(P, M, NO_ACCUM, R, GrB_ALL, n, GrB_ALL, n, replace);
    */

    //delta = min(P)
    GrB_reduce(&delta_f, NO_ACCUM, GrB_MIN_MONOID_FP64, P, DEFAULT_DESC);

    //P<M> = -delta
    GrB_assign(P, M, NO_ACCUM, -delta_f, GrB_ALL, n, GrB_ALL, n, DEFAULT_DESC);

    //P = P ⊕ (-P^T)
    GrB_Matrix_apply(P, NO_MASK, GrB_PLUS_FP64, GrB_AINV_FP64, P, transpose_a);
    /* Alternative:
    //P<M^T> = delta
    GrB_transpose(M, NO_MASK, NO_ACCUM, M, DEFAULT_DESC) );
    GrB_Matrix_assign_FP64(P, M, NO_ACCUM,
        delta_f, GrB_ALL, n, GrB_ALL, n, DEFAULT_DESC);
    */

    //AR = R ⊕ P
    GrB_eWiseAdd(R, NO_MASK, NO_ACCUM, GxB_PLUS_FP64_MONOID, R, P, DEFAULT_DESC);

    //R<R> = R
    GrB_apply(R, R, NO_ACCUM, GrB_IDENTITY_FP64, R, replace_output);
}


//max_flow = 0
double total_flow = 0;

//for i: 1 to n:
for (GrB_Index i = 0; i < n; i++) {

    //max_flow += A[s,i] - R[s,i]
    if(GrB_Matrix_extractElement(&capacity, A, source, i) == GrB_NO_VALUE){
        capacity = 0;
    }
    if(GrB_Matrix_extractElement(&residual, R, source, i) == GrB_NO_VALUE){
        residual = 0;
    }
    if(capacity != 0){
        total_flow += capacity-residual;
    }
```

Code listing B.2: The get_augmenting_path() function.
Mathematical notation corresponds to the outline in Listing 3.2.

```
GrB_Descriptor_new (&vxm_desc);
GrB_Descriptor_set (vxm_desc, GrB_MASK, GrB_COMP);
GrB_Descriptor_set (vxm_desc, GrB_MASK, GrB_STRUCTURE);
GrB_Descriptor_set (vxm_desc, GrB_OUTP, GrB_REPLACE);

//frontier[source] = true
GrB_Vector_setElement(frontier, 1, source);

parent[source] = source
GrB_Vector_setElement(parent_list, source, source);

//while(frontier not empty AND parent[sink] == NULL):
GrB_Vector_nvals(&frontier_nvals, frontier);
while ((frontier_nvals > 0) &&
    (GrB_Vector_extractElement(&sink_parent, parent_list, sink) == GrB_NO_VALUE))
{
    //frontier<parent'> = frontier * R
    GrB_vxm(frontier, parent_list, NO_ACCUM,
        GxB_ANY_FIRSTJ_INT32, frontier, R, vxm_desc);

    //parent<frontier> = frontier
    GrB_Vector_apply(parent_list, NO_MASK, GrB_PLUS_INT32,
        GrB_IDENTITY_INT32, frontier, DEFAULT_DESC);

    GrB_Vector_nvals(&frontier_nvals, frontier);
}

if ((GrB_Vector_extractElement(&sink_parent, parent_list, sink) == GrB_NO_VALUE))
{
    found_path = false;
} else {
    found_path = true;

    //current_vertex = sink
    GrB_Index curr_vertex = sink;

    //while(current_vertex != source):
    while (curr_vertex != source)
    {
        //M[current_vertex] = parent[current_vertex]
        GrB_Index parent;
        GrB_Vector_extractElement(&parent, parent_list, curr_vertex);
        GrB_Matrix_setElement(M, true, parent, curr_vertex);

        //current_vertex = parent[current_vertex]
        curr_vertex = parent;
    }
}

return found_path;
```

Code listing B.3: The get_mincut() function.
*C is the the matrix that containing the min-cut.
Mathematical notation corresponds to the outline in Listing 3.3.

```
//reachable[source] = true
GrB_Vector_setElement(reachable, 1, s);

//frontier[source] = true
GrB_Vector_setElement(frontier, true, s);

//while(frontier not empty):
bool successor = true;
while (successor) {

    //frontier<reachable'> = frontier * R
    GrB_vxm(frontier, reachable, NO_ACCUM, GxB_LOR_LAND_BOOL, frontier, R, desc);

    //parent<frontier> = frontier
    GrB_assign(reachable, frontier, NULL, true, GrB_ALL, n, NULL);

    GrB_reduce(&successor, NO_ACCUM, GrB_LOR_MONOID_BOOL, frontier, DEFAULT_DESC);
}

//t_cut<reachable'> = reachable * A
GrB_vxm(t_cut, reachable, NO_ACCUM, GxB_LOR_LAND_BOOL, reachable, A, desc);

//s_cut<reachable> = t_cut * A
GrB_vxm(s_cut, reachable, NO_ACCUM, GxB_LOR_LAND_BOOL, t_cut, A, transpose_b);

//START min_cut = A<s_cut.indices, t_cut.indices>
GrB_Vector_nvals(&t_len, t_cut);
GrB_Vector_nvals(&s_len, s_cut);

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_Matrix_nvals(&cut_len, cut_extract);

GrB_Matrix_extractTuples(cut_s, cut_t, cut_val, &cut_len, cut_extract);

GrB_assign(*C, NO_MASK, NO_ACCUM,
    cut_extract, s_indices, s_len, t_indices, t_len, DEFAULT_DESC);
//END min_cut = A<s_cut.indices, t_cut.indices>
```



NTNU
Norwegian University of
Science and Technology