Seminar Report

Paralleling Maximum Flow Problem

Jonas Hafermas, Zoya Masih

MatNr: 21862404,

Supervisor: Julian Kunkel

Georg-August-Universität Göttingen
Institute of Computer Science, GWDG

October 21, 2023

Abstract

Graph theory's *Maximum Flow Problem* continues to be of great importance throughout a wide range of topics, especially in rapidly expanding fields like telematics or high-performance computing. In the context of the Maximum Flow problem, we aim to compute the maximum possible "flow" (e.g. bits on a wire) which can be sent across all edges of a graph network from a start node to a sink node, with each edge having a specific flow capacity value. In our case, we specifically target computations on very large graphs (millions of nodes), using a newly parallelised version of the tried and tested algorithm named Dinic's algorithm. Standard (and most often sequential solutions) include the algorithm of Ford-Fulkerson, Edmonds-Karp's algorithm as well as Dinic's algorithm, though parallelised solutions seem to be largely unavailable, possibly due to the highly individual nature of parallel programming on high-performance platforms. That is the motivation that led us to implement a parallelised version of Dinic's algorithm to improve scalability on large graphs and thus produce better run-time results over various graph types.

Over a series of scenarios, we observed the execution time of a BFS algorithm, that is a main part of Dinic's algorithm. The first executions of the sequential algorithm highlighted the need for parallel algorithms to address latency in extensive graph computations. However, our findings revealed that parallelization did not consistently improve performance, as it depended on various factors such as graph density and the number of cores used. Notably, using the minimum number of cores necessary often yielded the best results. This study emphasizes the importance of a thoughtful approach to parallelization, considering both the characteristics of the input graphs and available computational resources.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC	High-Performance Computing
PCHPC	Practical Course on High-Performance Computing
BFS	Breadth-First-Search
DFS	Depth-First-Search

1 Introduction

According to the background of one of the authors in Graph theory, we decided to work on a famous problem in Graph field, titled 'Maximum Flow problem'. In this section we mainly focus on the definition, examples and applications of this problem, and we will also add the outlines of the current report and explain how we decided on this title.

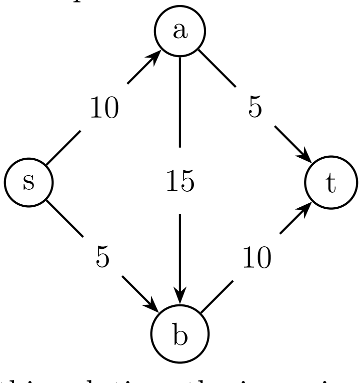
1.1 The Maximum Flow Problem

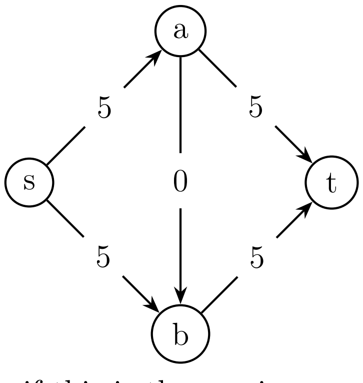
Let $N = (V, E)$ be a network with V and E as the set of vertices and edges of the network. We also call $s, t \in V$ the source and the sink of N respectively.
The capacity of an edge is a map $c : E \to R^+$. In other words, it is the maximum amount of flow that can pass through an edge.
A flow is a map $f : E \to R$ that satisfies the following statements.

- **Capacity constraint** The flow of an edge cannot exceed its capacity, in other words: $f_{uv} \leq c_{uv}$ for all $u, v \in E$.
- **Conservation of flows**. The sum of the flows entering a node must equal the sum of the flows exiting that node, except for the source and the sink.
  $\forall v \in V \setminus \{s,t\} : \quad \sum_{u:(u,v)\in E} f_{uv} = \sum_{u:(v,u)\in E} f_{vu}.$

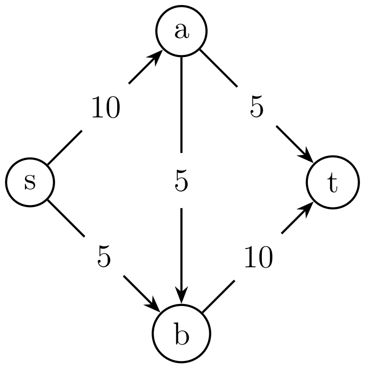
Note that flows are skew symmetric. It means that $f_{uv} = -f_{vu}$ for all $u, v \in E$.
We refer to the total amount of flow value passing from the source to the sink by Network's flow. It is formally shown by $f : E \to R^+$[Wik].

Example. Let us have the graph network bellow with 4 vertices and 5 weighted edges, in which the weights show the max flow that can be passed through the edge. Our goal is to calculate the maximum flow that can be passed from the source (vertex s) to the sink (vertex t). The most left figure below shows the initial status, and the second figure shows a possible flow in the network with total unit 10.





In this solution, the incoming flow to **t** is 10, but the question is if this is the maximum flow achievable? The answer is No. In the figure bellow, we have 15 units of flow, passed from s to t.



In real-world examples of huge graph networks, finding a solution to the problem is by no means trivial, and require complex and sophisticated algorithms.

1.2 Applications

Now that we know enough about the problem, it is the time to speak about the applications, and the necessity of solving such a problem.
The problem has many applications in real world. Here we will limit ourselves to two examples in industry, and some examples in I/O.

- **Circulation with Demands:**
  In this problem, we have a set of supply nodes whose purpose is to deliver a flow, such as products or goods, and a set of demand nodes that wish to receive this flow, such as customers or companies. Each supply node specifies the quantity of products it intends to send, and each demand node specifies the quantity it wishes to receive. Initial querying allows us to efficiently transfer products from supply nodes to demand nodes while adhering to capacity constraints.
  By reducing this problem to a Max Flow problem, we can find the best way of transferring goods from a supplier node to its demanding nodes[Mou].

- **Airline Scheduling:**
  Airline scheduling is a highly complex field, characterized by an extensive web of flights, diverse aircraft types, constraints stemming from limited gate availability, air traffic control rules, environmental compliance, rigorous safety standards, intricate crew regulations, complex compensation systems, and the ever-changing, competitive landscape of uncertain passenger demands and intricate pricing strategies. These intricacies are further exacerbated by the persistent challenges inherent to the airline industry, such as a historical lack of profitability, labor disputes, and outdated and inadequate infrastructure. Operations researchers have been captivated by these intricacies for over half a century, finding fertile ground for developing and applying models and algorithms.[Bar08]

Max Flow also has applications in Input/Output problems and optimization.

- **Task scheduling** Max Flow algorithms can be applied to task scheduling problems. In scenarios where tasks have dependencies and limited resources, finding the maximum flow can help in scheduling tasks efficiently.

- **Data transfer** In computer science and telecommunications, max flow algorithms are used to optimize data transfer. They help in finding the most efficient way to transmit data from one point to another in a network while respecting capacity constraints on various links or channels.

- **Network Routing:** By determining the maximum flow in a network, one can optimize data transmission or resource allocation.

1.3 Report Outline and Goals

In this report, we will center on the parallelization of the Max Flow problem, a very important problem in both the fields of Computer science and Graph theory. Although its history spans many decades of research and development, there have been recent algorithms and studies in this field. The outline of the report is structured as follows. Section two provides some details on the methodology that is used in this project. In section 3 the implementation of the work is explained for sequential and parallel solutions. The output results are analyzed in section 4, and the report will be concluded after a brief discussion on challenges and future works.

1.4 Authors

The authors of this report, Jonas Hafermas, and Zoya Masih, bring diverse academic backgrounds to this project. Currently, in his seventh semester studying for a Bachelor of Science in Computer Science, Jonas brings a strong foundation in Computer Science. In contrast, Zoya is pursuing a PhD in computer science with a rich background in pure mathematics and graph theory, with limited prior experience in computer science.
Given our distinct backgrounds, we were faced with the task of choosing a project that would be both accessible and intellectually stimulating for both of us and also parallelizable. We finally decided to tackle a problem that is firmly rooted in the realm of graph theory, while being equally important in computer science. This is a project that could be parallelized, increasing its feasibility and efficiency in addressing complex computational challenges. While we allocated responsibilities based on our backgrounds, we also collaborated closely on both aspects of the project to ensure a holistic approach.

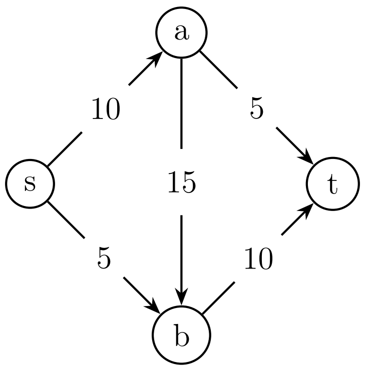
2 Methodology

In this section, we provide a detailed explanation of the algorithm that we have used, and then will provide some details about the environment setup of our project, and the data preprocessing.

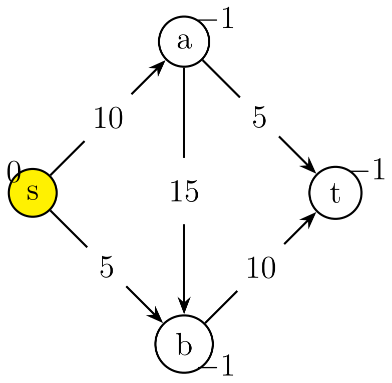
2.1 Dinic's Algorithm

Dinic's algorithm is one of the several algorithms used for solving the max flow problem. This algorithm was invented by Yefim Dinitz in 1970, and is one of the most famous algorithms in the field. Dinic's algorithm has a time complexity of $O(V^2 * E)$ in the worst case, where V is the number of vertices and E is the number of edges in the flow network. This makes this algorithm more efficient than some other algorithms like the Ford-Fulkerson method, which may have unbounded running times in the worst case.
Dinic's algorithm employs a layered approach, where it constructs a level graph in each

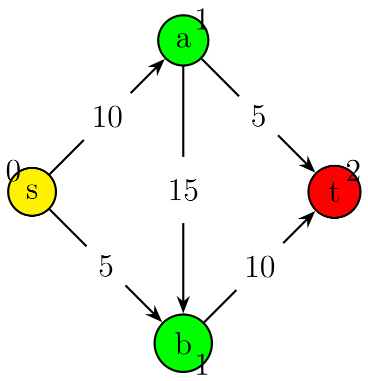
iteration. This approach naturally lends itself to parallelization, as the computation of flow can be performed in parallel for different levels of the graph.
Dinic's algorithm also uses the concept of blocking flows, which allows it to find augmenting paths efficiently. Blocking flows can be identified in parallel, and the computation of blocking flows can be distributed among multiple processors or threads, making it suitable for parallel computing environments. In the following we will explain how this algorithm works with the help of an example[Din70]. Let us consider the example which we had provided previously in 1.1.



1. Initially, the source vertex is labelled 0 and other vertices are labelled -1. The total flow in this point is zero.

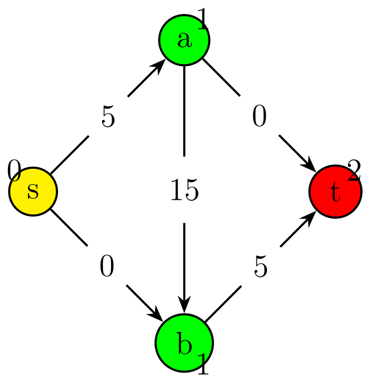


2. In the second step, each unvisited child of an arbitrary vertex u with label i, receives label i+1 (BFS)

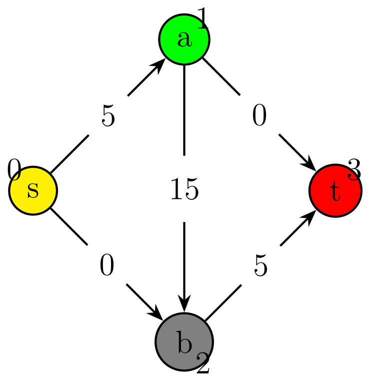


3. For the paths in order of the labels, the algorithm finds the minimum capacity. Finding the paths is executed by using DFS method. Consequently the flow will be added to the previous flow, and all the capacities get updated.

   In this example, the path order from the source to the sink is $0, 1, 2$, and the paths $s - a - t$ and $s - b - t$ are two paths in this order. The minimum capacity among the edges of paths, is 5 for both the paths. Therefore, the total path that we can pass through each of them is 5, and totally we are passing 10 units of flow. In this point the capacities get updated and the updated network graph will be as following.



4. We call the existing graph by 'residual Graph'. The procedure iterates for the residual graph.
   In our example, the path order for the residual graph will be $0, 1, 2, 3$ and $s-a-b-t$ is a path from the source to the sink in the exact order. We can now pass 5 units through this path, and update the total flow to $10 + 5 = 15$.



This ends our example, as the residual graph in the next step has capacity 0 in all the paths from s to t.

2.2 Experimental Setup

This project was conducted on the computing cluster of GWDG. The key components of the hardware environment are as follows. The cluster is powered by Intel(R) Xeon(R) Silver 4214 CPUs with 48 CPU cores, supporting virtualization (VT-x) and organized into two NUMA nodes for efficient memory management. Memory (RAM) capacity: 187 GB (23 GB in use, 131 GB available), with 10 GB allocated for shared memory and a

32GB buffer/cache.
The software environment for this project is based on Scientific Linux 7.9. In this project, we wrote algorithms in C, and we employed NetworkX library of Python 3 to create and manipulate the input data used in our algorithms. For the parallelization of our algorithms, we utilized the Message Passing Interface (MPI). MPI proved to be a robust choice for distributing the computational workload across multiple processors, facilitating parallel processing effectively. In terms of debugging and code quality assurance, we employed Valgrind, a powerful tool for memory analysis and error detection. Valgrind played a vital role in identifying memory leaks and ensuring the stability of our codebase. It allowed us to maintain code quality and reliability throughout the development process."

2.3 Data Preprocessing

Solving the maximum flow problem on big graphs is an active area of research, and new techniques continue to be developed to handle ever-larger and more complex graph structures efficiently. An average graph with 1 million nodes can have 300 billion(3×10^{11}) edges. Such vast datasets exceed the capacity of individual CPUs to hold in local memory. As a result, efficient partitioning and memory management become paramount. Distributing computational tasks across multiple processors or machines holds the promise of faster problem-solving, but it also presents formidable challenges when dealing with graphs of this magnitude. Effectively parallelizing graph algorithms under these conditions is far from straightforward.

In the initial stages of our project, we explored various methods for generating and handling graph data. One approach involved utilizing popular graph generation tools such as Graph500 and GraphX. These tools are well-established and have been widely used in the research community. Although This took a significant portion of our time, later, when we delved deeper into the requirements of our project, it became apparent that these off-the-shelf solutions were not well-suited to our specific needs.
One of the primary challenges we encountered was the need to work with exceptionally large graphs, often comprising millions of nodes and edges. Many existing tools and libraries struggled to efficiently generate and manage graphs of this scale. Moreover, the ability to incorporate custom graph properties and characteristics relevant to our study was limited within these pre-built solutions.
As a result of these challenges, we opted for an alternative approach. We transitioned to using JSON files to store the essential information about our graphs. This transition allowed us to have full control over the generation and representation of our graph data. JSON's flexibility and human-readable format made it a suitable choice for creating custom graph descriptions.
To seamlessly integrate these JSON-based graphs into our parallelized Dinic's algorithm, we developed a dedicated JSON parser. This parser was instrumental in extracting the graph's structure, edge capacities, and other pertinent information required for the algorithm's execution. The customizability of this approach not only improved our ability to tailor the graph data to our research needs but also facilitated the integration of real-world data into our experiments.
The transition to JSON-based graph representations and the development of the corresponding parser proved to be a pivotal step in our project. It enabled us to work with large,

custom graphs effectively, aligning our data preprocessing pipeline with the demands of our parallelized algorithm and the specific requirements of our research objectives.

3 Implementation

In this section *Note: In the following, the terms "node" and "vertex" are used interchangeably as both describe fundamental objects in graph theory which are connected by edges as described in section 1.*

3.1 Sequential Solution

When starting out with developing a sequential implementation of Dinic's algorithm, a programming language of choice had to be determined. Owing to existing proficiency, C was quickly chosen as the working language, but there were other factors to consider as well: Firstly, C offers native bindings for the openMPI implementation of the MPI standard, making for a robust code base and enabling the use of the latest version of openMPI. The standardisation of C furthermore allowed great portability of code from local debugging to large-scale testing on the SCC, and finally the wide-spread usage of C in operating systems development (especially POSIX conformity) meant that a range of options for architectural tailoring existed (and also threading via `<pthreads.h>`), should there still be time for further optimisation later on.

3.1.1 Sequential BFS

The core of our sequential BFS algorithm is fairly straightforward: A tail queue (borrowed from the BSD library `<sys\queue.h>`) serves as the data structure holding unvisited vertices (children), and during each loop another vertex is dequeued and all of its children are themselves enqueued. In our implementation, the `list_start` pointer has to be reset according to the length of the list each time a new vertex is dequeued and examined, which was changed in a beta draft where the length is an additional parameter of every series of the graphs adjacency list `graph->adj`, saving the loop traversal.

Listing 1 Relevant part of the sequential BFS algorithm in C

```c
while (!TAILQ_EMPTY(&head)) {
    struct BFSQelem *u = TAILQ_FIRST(&head);
    TAILQ_REMOVE(&head, TAILQ_FIRST(&head), entries);
    struct AdLst *list_start = *(graph->adj+(u->val));
    // determine length of our list
    for (i = 0; list_start->next != NULL; ++i) {
        list_start = list_start->next;
    }
    // reset start pointer
    for (int j = 0; j < i; j++) {
        list_start = (*(graph->adj+(u->val)))->next;

        for (int k = 0; k < j; k++) {
            list_start = list_start->next;
        }
        // select edge (we ignore paths with blocking flow)
        e = list_start->value;

        if ((*(graph->levels+(e->v)) == -1) && (e->flow < e->c)) {
            // node is child, increase level by one
            *(graph->levels+(e->v)) = *(graph->levels+(u->val))+1;
            TAILQ_INSERT_TAIL(&head, &elems[e->v], entries);
        }
    }
}
```

3.1.2 Sequential DFS

The sequential depth-first-search implementation recursively traverses our graph from left to right. We decided for a recursive implementation because this version doesn't suffer from a risk of duplicate vertices, as would be the case with most iterative solutions using stacks. In practice, this DFS-implementation has been modified somewhat to work with flow values on graph edges. `node_visit` denotes an array of all vertices of the graph and is used to track how many neighbours of a given node's adjacency list of length `i` have been visited already. Starting at the source node `s`, the length of its associated adjacency list is saved to `i` and the loop is entered. First, the `node_visit[u]`'th node from the adjacency list is computed and it is checked if there are any outgoing edges which also have flow capacity left. One of these edges is traversed and the current flow is updated according to said flow capacity. Now, temporary flow is computed recursively and negative flow is added to the reverse edges as per the rules of Dinic's algorithm. This is repeated until the temporary flow reaches zero and the flow is *stabilised*. Unfortunately, this part of the algorithm, even though developed and functional, has been excluded from our result due to our focus newly shifting to comparing runtimes of sequential versus parallel implementations of BFS.

Listing 2 Relevant part of the sequential DFS algorithm `send_flow` in C

```c
while (node_visit[u] < i) { // i denotes length of adjacency list of vertex u
    list_start = (*(graph->adj+u))->next;

    for (int j = 0; j < node_visit[u]; j++) {
        list_start = list_start->next;
    }

    // currently examined edge
    e = list_start->value;

    // check if vertex v is neighbour (u has outgoing edge to v)
    // and if flow capacity is sufficient
    if ((*(graph->levels+(e->v)) == *(graph->levels+u)+1) && (e->flow < e->c)) {

        curr_flow = (int) floor(fmin(flow, e->c-e->flow));
        // accumulate temporary flow recursively
        temp_flow = send_flow(graph, e->v, curr_flow, t, node_visit);

        if (temp_flow > 0) {

            e->flow += temp_flow;
            list_start = (*(graph->adj+(e->v)));

            for (int j = 0; j <= e->rev; j++) {
                list_start = list_start->next;
            }
            // add negative temporary flow to reverse edges
            list_start->value->flow -= temp_flow;
            return temp_flow;
        }
    }
    node_visit[u] += 1;
}
```

3.2 Parallel Solution

Paralleling Dinic's algorithm was more of a challenge than was initially thought. Even though the algorithm essentially decomposes into one part BFS and one part DFS with a little overhead due to function logic of Dinic's concept of finding the blocking flow, parallelising even those basic graph theory algorithms proved quite difficult. For this reason, previous work by Beamer et al. served as the basis for our parallelisation attempts with BFS, where some wiggle room for performance improvements later on was kept. DFS proved to be a more difficult to be parallel meaningfully, though some pseudocode was developed, which, due to time constraints, unfortunately remained out of the final implementation.

3.2.1 Parallel BFS

For parallel BFS, the 1D decomposition of the adjacency matrix of a given graph was used with the following approach by Buluç and Madduri [BM11], and with the 2D variant planned as an additional feature if there is still time. The illustrations used in the project presentation still hold true and offer a nice glimpse at the inner workings of the parallel

BFS algorithm:

INPUT: n subgraphs G_n of $G(V, E)$, source vertex s, sink vertex t

OUTPUT: Truth value: Is t reachable from s? ($\overrightarrow{st} \neq -1$)

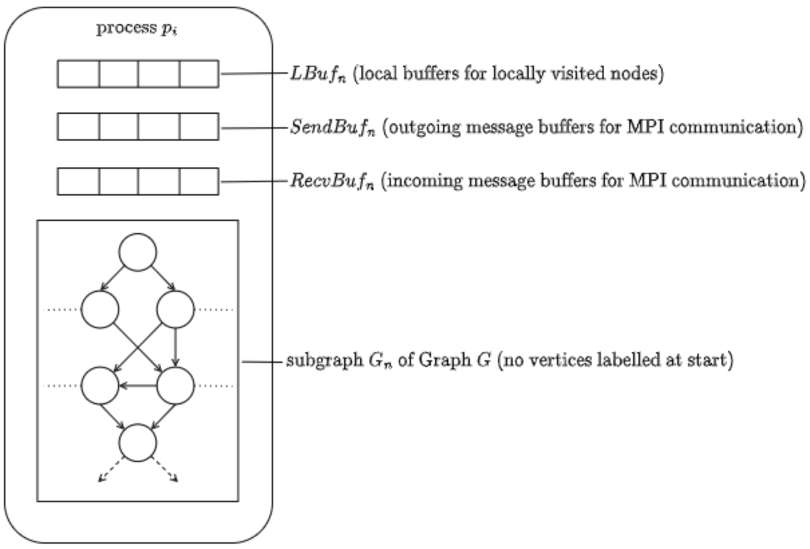

process p_i
$LBuf_n$ (local buffers for locally visited nodes)
$SendBuf_n$ (outgoing message buffers for MPI communication)
$RecvBuf_n$ (incoming message buffers for MPI communication)
subgraph G_n of Graph G (no vertices labelled at start)

Figure 1: Model of a typical process in parallel BFS

Let p_n denote the n-th process of a system for distributed computing with one or more local buffers $LBuf_{n_i}$ (depending on level of multithreading), as well as an outgoing message buffer $SendBuf_n$ and an incoming message buffer $RecvBuf_n$ for MPI communication. The algorithm takes a partitioning scheme of n subgraphs $G_n(V,\ E)$, $V = \{\text{OWNED VERTICES}\}$, $E = \{\text{OWNED EDGES}\}$ and tries to determine if there is a valid path from the source node s to the sink node t. The only output is the corresponding boolean value which is used too determine whether or not Dinic's Algorithm is finished (if the sink node is no longer reachable via BFS, then blocking flow has been detected and thus the maximum possible flow on this graph has been achieved).
At the start, no node is part of the frontier Stack FS. The distance (level) array d is distributed among the processes so that each process only manages those vertices it owns, so for example process i might manage $d[v]$ if it owns vertex v. The level is set to zero and process 1 is identified as the owner of s in this example. Process 1 sets $d[s]$ to zero and sends s to the frontier stack. The level is increased by one since there are no unchecked nodes on level zero remaining.
Then, all processes check each frontier element for neighbours they might own, which in this example only process 1 does, the neighbours being nodes u and v. p_1 adds those to its read buffer $LBuf_1$ as well as its send buffer $SendBuf_1$, even though it doesn't make any sense in this scenario since p_1 incidentally both reaches and owns all neighbouring nodes from s (this is already a potential point for optimisation). Going through the buffer contents, p_1 determines if $d[u]$ or $d[v]$ haven't been visited yet; if so, then it sets the levels accordingly ($d[u] = d[v] = 1$).
Now, p_1 pushes u and v to its stack of newly-visited vertices NS_1 and sends the contents to the newly forming next frontier stack. The level is increased once more since all level 1 neighbours were owned and processed by p_1.
With the new frontier constructed, neighbours for u and v are searched for next, but

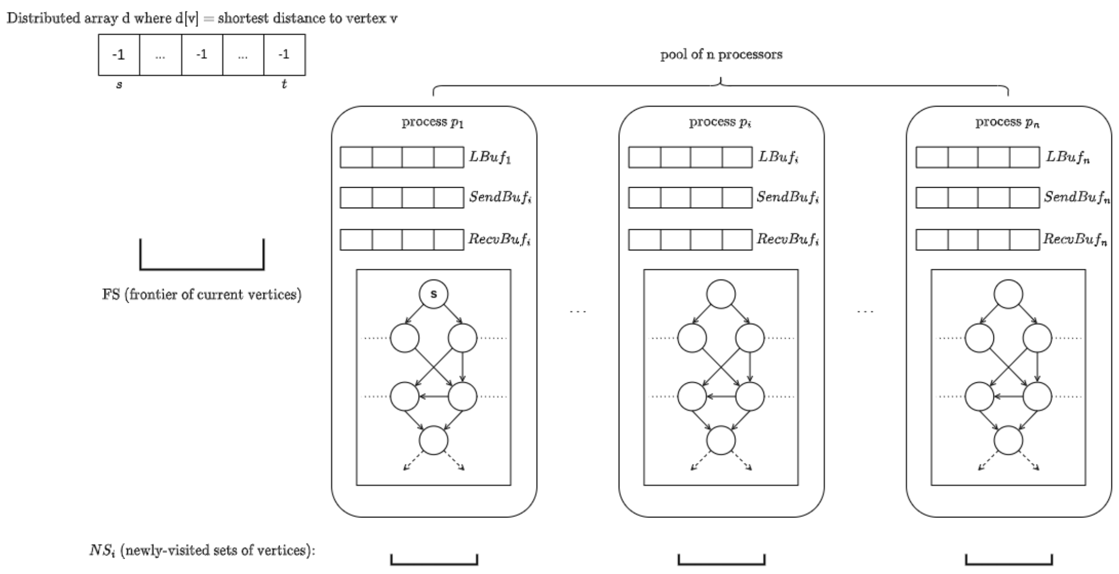


Figure 2: Determining the source node

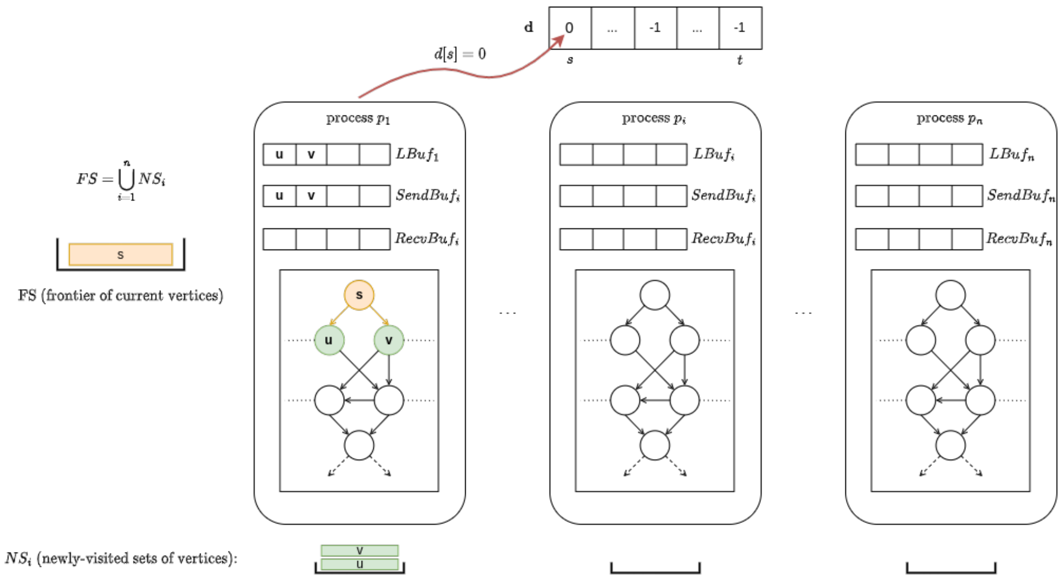


Figure 3: Setting the distance and incrementing the level

this time p_1 does not own all of them, so it calls `find_owner()` to get the rank of those processes owning the neighbours in question, w and x. p_1 places the two nodes into its send buffer and sends them to their owner's ranks; in this example, p_i holds w while p_n holds x. Again, newly-visited vertices have their distance array entries set in parallel and are collected into their newly-visited buffers for aggregation at the new frontier.

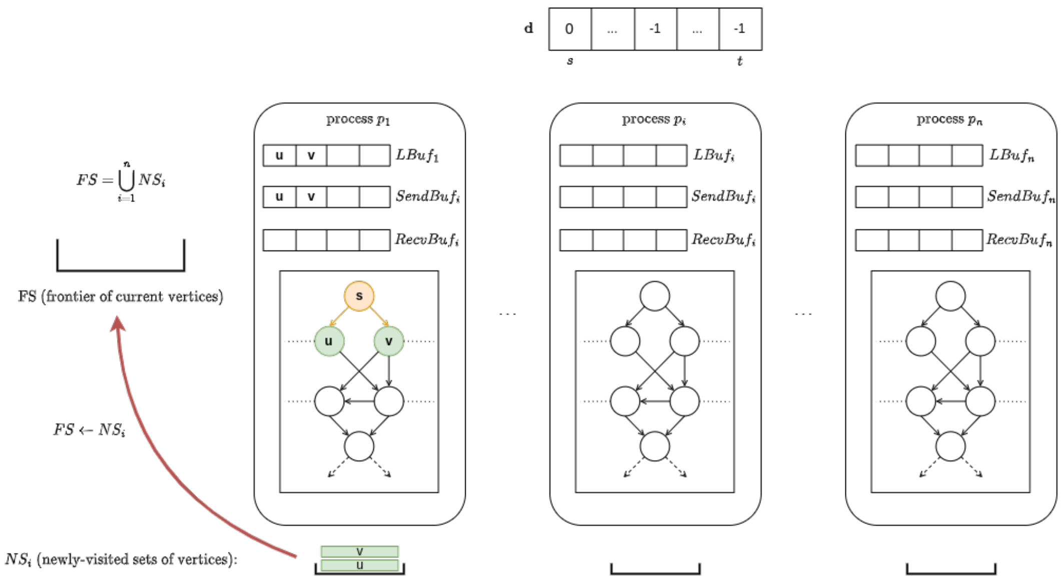


Figure 4: Sending buffer contents to the new frontier

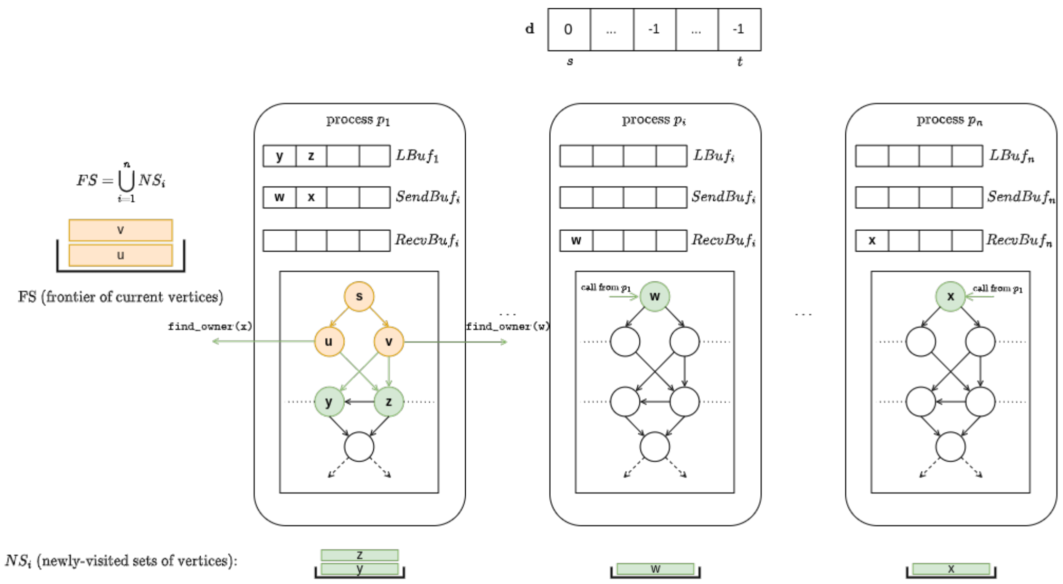


Figure 5: Searching for neighbours

Once more, the level is increased, and all processes send the contents of their buffers to the root process, where they are merged into the new frontier. Measures should be undertaken that all worker processes are finished with sending in their NS_i buffers by the time the new frontier gets broadcasted to avoid data loss (for example via `MPI_Barrier()`).
Finally, the frontier stack is broadcasted again and each element is inspected for possible neighbours by each process. The processes call `find_owner()` accordingly to get the ranks of the processes owning the neighbours to send them the respective vertices. The previous steps are repeated until the frontier stack is empty, as in that there are no new vertices arriving after a loop traversal. Then, the distance array is accumulated at the root process, which checks if $d[t] \neq -1$, meaning if t is still reachable. In the context of Dinic's algorithm, the partitioned adjacency matrix should contain the capacity values for each edge instead of zeros and ones, and flow values are kept separately to ignore edges on which the capacity is already saturated.

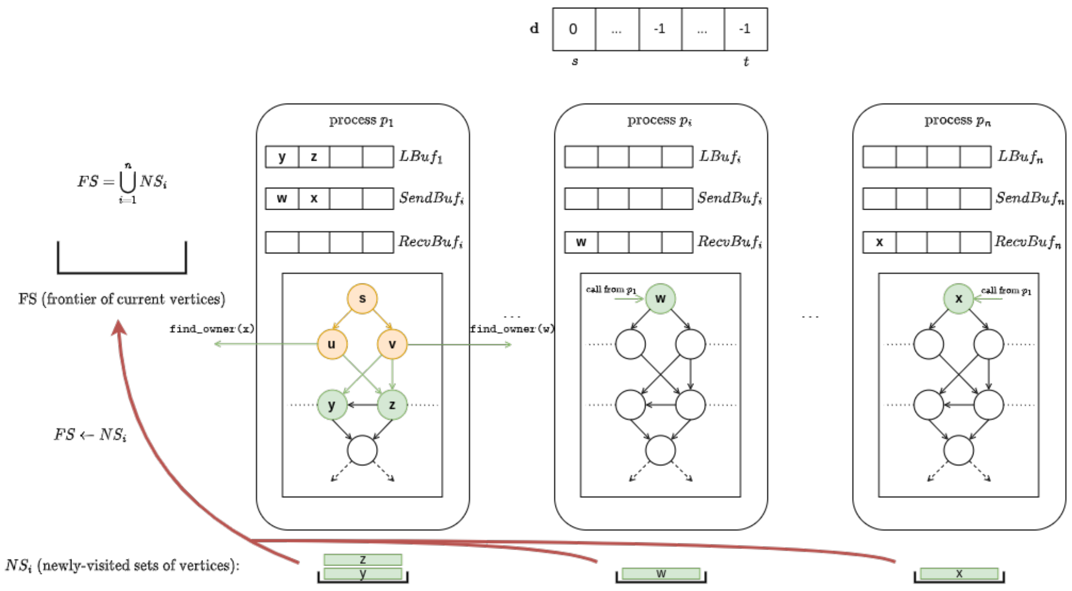


Figure 6: Collecting all newly-visited nodes again

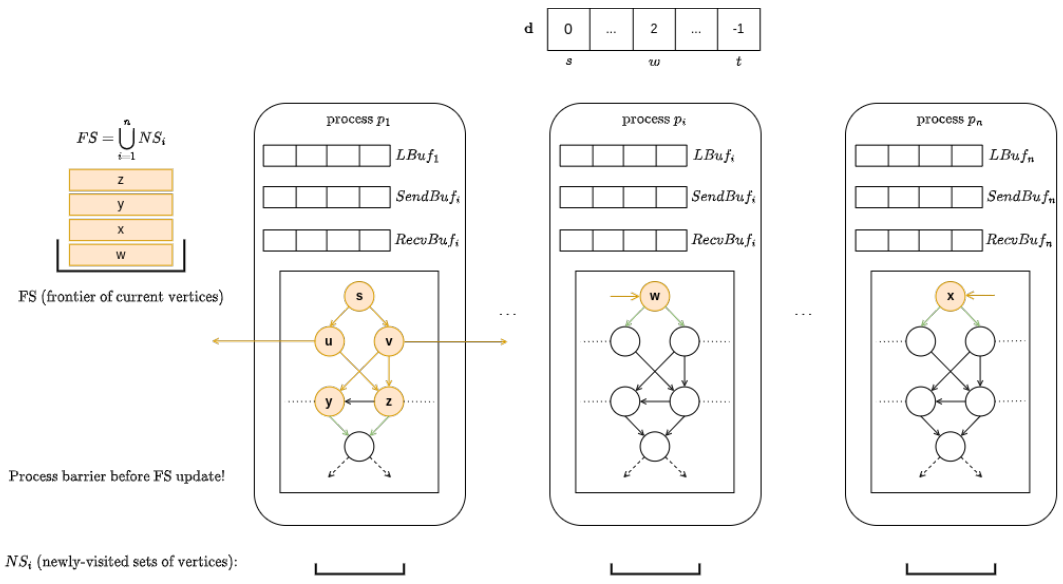

d
0 ... 2 ... -1
s
w
t
$FS = \bigcup_{i=1}^{n} NS_i$
FS (frontier of current vertices)
Process barrier before FS update!
NS_i (newly-visited sets of vertices):
process p_1
process p_i
process p_n

Figure 7: Searching for neighbours once more

3.3 Other Implementation Goals

On our learning journey, we worked on a few topics related to our project, and since a significant amount of time and effort went into those endeavours, they should be briefly highlighted.

3.3.1 Graph Frameworks

A not insignificant amount of time was spent on deciding whether or not to use a graph processing framework for the underlying systems of our algorithm and which framework would be most appropriate, especially considering parallel workloads. We used findings from Heidari et al. [Hei+18] as well as Guo et al. [Guo+14] to gain insights on that topic, but after discussing the idea with our supervisor, it was decided that we would not be using graph frameworks to achieve our goal.

3.3.2 Optional Features

Much thought went into ideas about how to improve the basic implementation, and even though most of them didn't make it into our final implementation, they shall still be presented briefly. Firstly, for our parallel BFS, we also considered the so-called 2D decomposition of the adjacency matrix given by algorithm 3 from the paper by Buluç and Madduri [BM11], where graph traversal breaks down into simple linear algebra operations which are efficiently realised with libraries like BLAS. The graph partitioning scheme is not trivial, however, and load balancing plays an even greater role in this approach than in the (naive) 1D implementation, and since time was already running short, it was decided against using the 2D decomposition. Our parallel BFS implementation could nevertheless benefit from some additional performance improving, chiefly following a so-called *Direction-Optimising* approach in which the BFS algorithm dynamically changes between the classical top-down processing scheme and a novel bottom-up approach which was discussed by Beamer et al. [Bea+13], thereby reducing the number of times already visited vertices are checked, which especially becomes an issue with dense graphs at their broadest point. All in all, there is no single best variant, however, because while bottom-up BFS excels in situations like the aforementioned, it incurs performance penalties when used on sparse or narrow graph portions, which makes the combined, dynamic usage of both approaches highly effective in reducing the overall node visits, as was shown by Beamer et al. [BAP12]. On the other hand, this approach also requires substantial load balancing efforts as well as a clever partitioning scheme for classifying graph portions as suitable for bottom-up and top-down processing, respectively. As such, the idea was deemed too time-intensive and was put on hold. What we did do, however, was replacing our csv parser with a JSON parser due to the performance penalties incurred when opening and caching csv files with millions of entries (which is an entirely valid scenario, since a graph with 10000 nodes may already have tens of millions of edges each occupying a line in our input file

4 Performance Analysis

We executed at least 100 different experiments to find the patterns and to analyze the effect of the paralleling the BFS algorithm. The experiments consists of several scenarios. At first the sequential algorithm was executed while the size of graphs were continually changing in term of vertices and edges. In the second step, the execution time of sequential algorithm was compared to the parallel one, for many different graphs, where the graphs remained unchanged during the two parallel and regular executions. In the third scenario, parallel program was tested for different number of cores and on different graphs.

4.1 Analysis of the Results

Graph 8 demonstrates how run time of BFS sequential algorithm changes when the number of vertices of input graphs growth. In this experiment the density of the graphs remained unchanged to make the analysis easier. The significant growth in the execution time, when the graph scale up,We executed at least 100 different experiments to find the patterns and to analyze the effect of the paralleling the BFS algorithm. The experiments consists of several scenarios. At first the sequential algorithm was executed while the size of graphs were continually changing in term of vertices and edges. In the second step, the execution time of sequential algorithm was compared to the parallel one, for many different graphs, where the graphs remained unchanged during the two parallel and regular executions. In the third scenario, parallel program was tested for different number of cores and on different graphs.

4.2 Analysis of the Results

Graph 8 demonstrates how run time of BFS sequential algorithm changes when the number of vertices of input graphs growth. In this experiment the density of the graphs remained unchanged to make the analysis easier. The significant growth in the execution time, when the graph scale up, shows the necessity of a parallel algorithm which can handle the latency in huge graphs. A part of results in these experimental results are provided

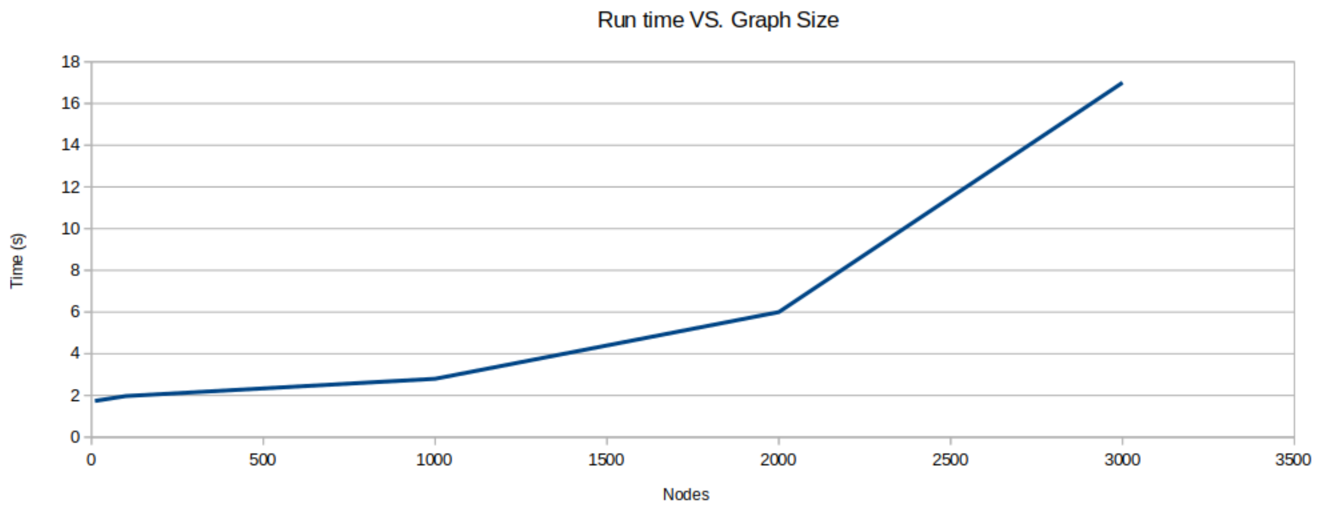


Figure 8: RunTime VS Graph size

in table 1. In the following, based on the results we analyze how paralleling the algorithm can affect performance. The sequential algorithm was tested for a quite big graph with 10 million vertices, and the execution was perfectly done in 78 seconds. In contrast, the parallel algorithm could be executed when the graph is at most 15×10^4 vertices. The execution for a sparse graph of size 2×10^5, was broken with memory failure error for 60 allocated cores. As we could not allocate more than 60 CPU cores, the execution of the algorithm for bigger graphs was not doable, and we were limited to the size 100 K.

Vertices	Edges	Sequential Run time(s)	Parallel Run time(s)
1000	10^4	2.25	4.21
2000	$4 * 10^5$	8	8.45
2000	$5 * 10^5$	10.86	10.69
2000	$6 * 10^5$	15.8	13
3000	10^6	33	39

Table 1: Execution of the sequential and parallel algorithms on 20 cores

The results show that the parallel algorithm does not essentially improve the execution time. To prove this claim, refer to The table 2. This table has demonstrated the results of execution the sequential and parallel algorithms for a fixed graph with one thousand vertices and 10 thousand edges. The parallel algorithm could not be run on less than 11 cores, and for more than this many cores, the execution time increased significantly in comparison with the sequential execution.

Sequential	Parallel-10 cores	Parallel-11 cores	Parallel-13 cores	Parallel-15 cores
2.25 s	Not Executed	3.44 s	3.57s	4.29s

Table 2: Sequential and parallel execution on the same graph

Although the parallel algorithm is not essentially faster than the sequential version, it is seen that it some times has a shorter execution time. To see this, refer table 3 and notice how the paralleled algorithm on 20 cores is working faster while the number of edges in a graph with 2 thousand vertices increase.

Number of Edges	Sequential Execution	Parallel Execution- 20 cores
$4 * 10^5$	8.45 s	8 s
$5 * 10^5$	10.8 s	10.6 s
$6 * 10^5$	15 s	13 s
$7 * 10^5$	16.1 s	14.5 s

Table 3: Sequential and parallel execution on changing number of edges

On another point of view, we examined the execution on the same graph, while we increased the number of cores. This experiment was run for many different graphs, and the results showed that the best scenario in terms of execution time is experienced when the minimum possible number of cores are used. To make this discussion clearer, graph 9 shows the results of a graph with 9 hundred vertices and 3 hundred thousand edges, which is a quite dense graph. The parallel algorithm could not be executed on less than 19 cores, and from this number, the growth on the number of cores leads to slower execution. Requesting more cores than necessary, could lead to resource contention, which might extend the job's run time, and this can justify the result that we have experienced.

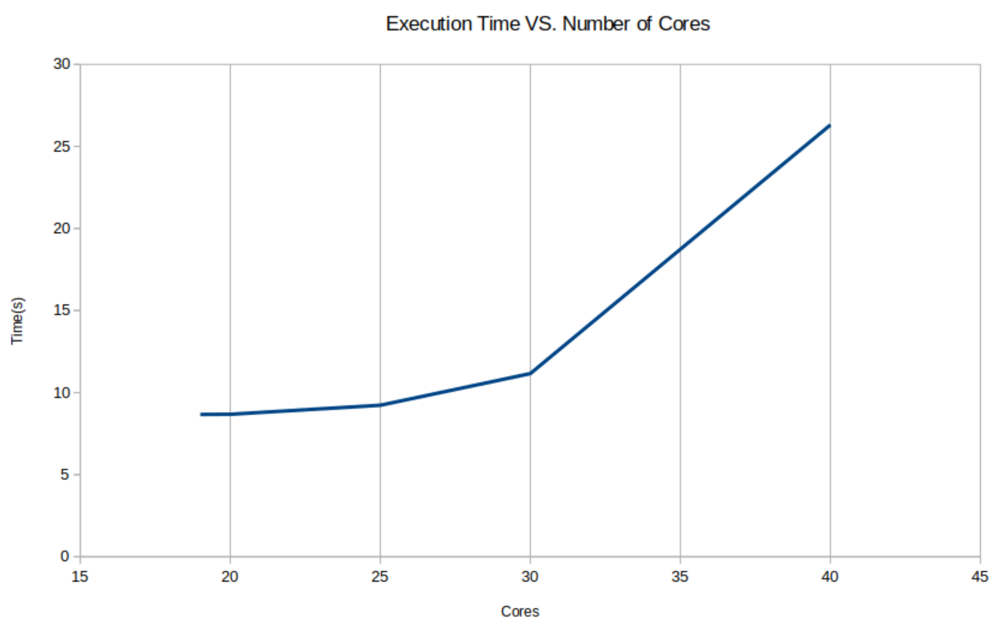


Figure 9: Execution time VS. number of cores

5 Challenges and Discussion

5.1 Challenges

The project was not without its challenges. While the development of a stable, sequential implementation of Dinic's algorithm proceeded smoothly and encouraged further work, the process of paralleling the algorithm proved surprisingly challenging due to the highly variable nature of unpolished MPI programs. Especially memory debugging turned out difficult, even with the help of debugging tools like Valgrind since the problems with C's reliance on manual memory management were only exacerbated by the addition of MPI functionality.

Most definitely, more delicate planning will be needed in future projects of this kind to account for possible memory issues comprehensively. On the other hand, using a memory-safe yet scalable language like Rust would possibly save even more time; for this project, Rust was deemed unsuitable because no native MPI-bindings existed and also language proficiency was greater with C, which might just as well change in the future. Since DFS on non-binary trees was already deemed difficult to paralleling, we eventually decided to focus on the BFS part of the algorithm since there, many possibilities for parallelisation exist. The original plan was to adapt the sequential DFS solution and combine it with the parallel BFS solution to create a somewhat parallel version of Dinic's algorithm.

What ultimately led to the change of course was the fact that BFS also needed to be modified to work with the flow calculations on edges done by DFS, and finding errors in this modification turned out to be like trying to find needles in a haystack, and this was what made us deem our initial goal of paralleling Dinic's algorithm unrealistic in the remaining time for our project, thus the focus on implementing just BFS to have a paralleling proof of concept emerged.

As was mentioned in the performance analysis, it became apparent that memory efficiency of our algorithm is rather subpar, which, while undesirable, doesn't break the logic of our program, which is why we gladly acknowledge that more could have been done in that regard, if there was more time. Overall, time proved to be the biggest challenge of this project. It is safe to say that we underestimated the time every milestone in our project would take and that our ambitions might have been a little high to begin with. For the next project of this kind, or any project at all, more emphasis should be put on regular progress reports and scheduled meetings where issues can be discussed and dealt with in a timely manner. A more consistent usage of web services like OpenProject could also prove beneficial to gain a better overview over the project and the tasks at hand.

5.2 Discussion

Although some limitations did not allow us to prove or test this claim, there still is a high possibility that the parallel algorithm will work faster in the range of very large scale graphs. Furthermore, one can work on optimizing the parallel algorithm, in order to make it faster than sequential for small and average graphs. For this purpose, better memory allocation can be a good idea.
As previously mentioned, the Dinic's algorithm is not thoroughly executed in this project, and we remained limited to BFS algorithm in labeling the vertices of a graph, which is the first part of the Dinic's algorithm. There can also be further attempts to paralleling DFS, or executing the parallel BFS and sequential DFS in tandem.

6 Conclusion

In conclusion, the project was primarily defined to parallelize Dinic's algorithm that find the maximum flow of a network. Maximum flow problem is a very famous and broadly used in computer science. This report contains all the efforts and attempts that the authors made to parallelize Dinic's algorithm, that is a well known algorithm in maximum flow problem.The project later during the project work has changed. Although the conceptual ideas and schems for entire algorithm are provided, the report mainly focused on the development and analysis of a parallel implementation of the Breadth-First Search (BFS) algorithm within the context of the larger Dinic's algorithm. The parallelization of BFS was explored in detail, and its performance was compared to the sequential BFS algorithm. While the parallel algorithm showed potential for improving execution times, it was found that resource contention, memory inefficiencies, and the overhead of managing multiple cores can limit its benefits, especially for smaller and medium-sized graphs. However, for extremely large-scale graphs, the parallel algorithm may still hold promise.

Several challenges were encountered during the project, including memory management issues, debugging complexities, and time constraints. In future work, optimizing memory usage, parallelizing the DFS portion of Dinic's algorithm, and combining the parallel BFS with a parallel DFS could lead to more efficient solutions.

Overall, this project highlights the complexity of parallelizing graph algorithms and the importance of careful planning, thorough testing, and optimization to achieve optimal performance in parallel computing environments. Further research and development are needed to fully harness the potential of parallel algorithms in graph theory.

References

[BAP12] S. Beamer, K. Asanović, and D. Patterson. "Direction-Optimizing Breadth-First Search". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012. ISBN: 9781467308045.

[Bar08] Cynthia Barnhart. "Airline Scheduling: Accomplishments, Opportunities and Challenges". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Ed. by Laurent Perron and Michael A. Trick. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–1. ISBN: 978-3-540-68155-7.

[Bea+13] S. Beamer et al. *Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search*. Tech. rep. UCB/EECS-2013-2. EECS Department, University of California, Berkeley, Jan. 2013. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-2.html`.

[BM11] Aydin Buluç and Kamesh Madduri. "Parallel Breadth-First Search on Distributed Memory Systems". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: Association for Computing Machinery, 2011. ISBN: 9781450307710. DOI: `10.1145/2063384.2063471`. URL: `https://doi.org/10.1145/2063384.2063471`.

[Din70] Yefim A Dinitz. "An algorithm for the solution of the problem of maximal flow in a network with power estimation". In: *Doklady Akademii nauk*. Vol. 194. 4. Russian Academy of Sciences. 1970, pp. 754–757.

[Guo+14] Yong Guo et al. "How Well Do Graph-Processing Platforms Perform? An Empirical Performance Evaluation and Analysis". In: May 2014, pp. 395–404. ISBN: 978-1-4799-3800-1. DOI: `10.1109/IPDPS.2014.49`.

[Hei+18] Safiollah Heidari et al. "Scalable Graph Processing Frameworks: A Taxonomy and Open Challenges". In: *ACM Computing Surveys* 51 (June 2018), pp. 1–53. DOI: `10.1145/3199523`.

[Mou] Dave Mount. "Circulation with Demand". In: (). URL: `https://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect17-flow-circ.pdf`.

[PD11] Steven J. Plimpton and Karen D. Devine. "MapReduce in MPI for Large-Scale Graph Algorithms". In: *Parallel Comput.* 37.9 (Sept. 2011), pp. 610–632. ISSN: 0167-8191. DOI: `10.1016/j.parco.2011.02.004`. URL: `https://doi.org/10.1016/j.parco.2011.02.004`.

[Wik] Wikipedia. "MaxFlow". In: *Wikipedia* ().

A Work sharing

After jointly deciding on an algorithm to tackle the problem with, we tried to go for an approach where Jonas did most of the sequential and parallel coding as well as the modelling work leading up to those steps, while Zoya worked with the algorithms to perform runtime analyses and scaling benchmarks as well as doing some work on pseudocode for a parallelised DFS implementation.

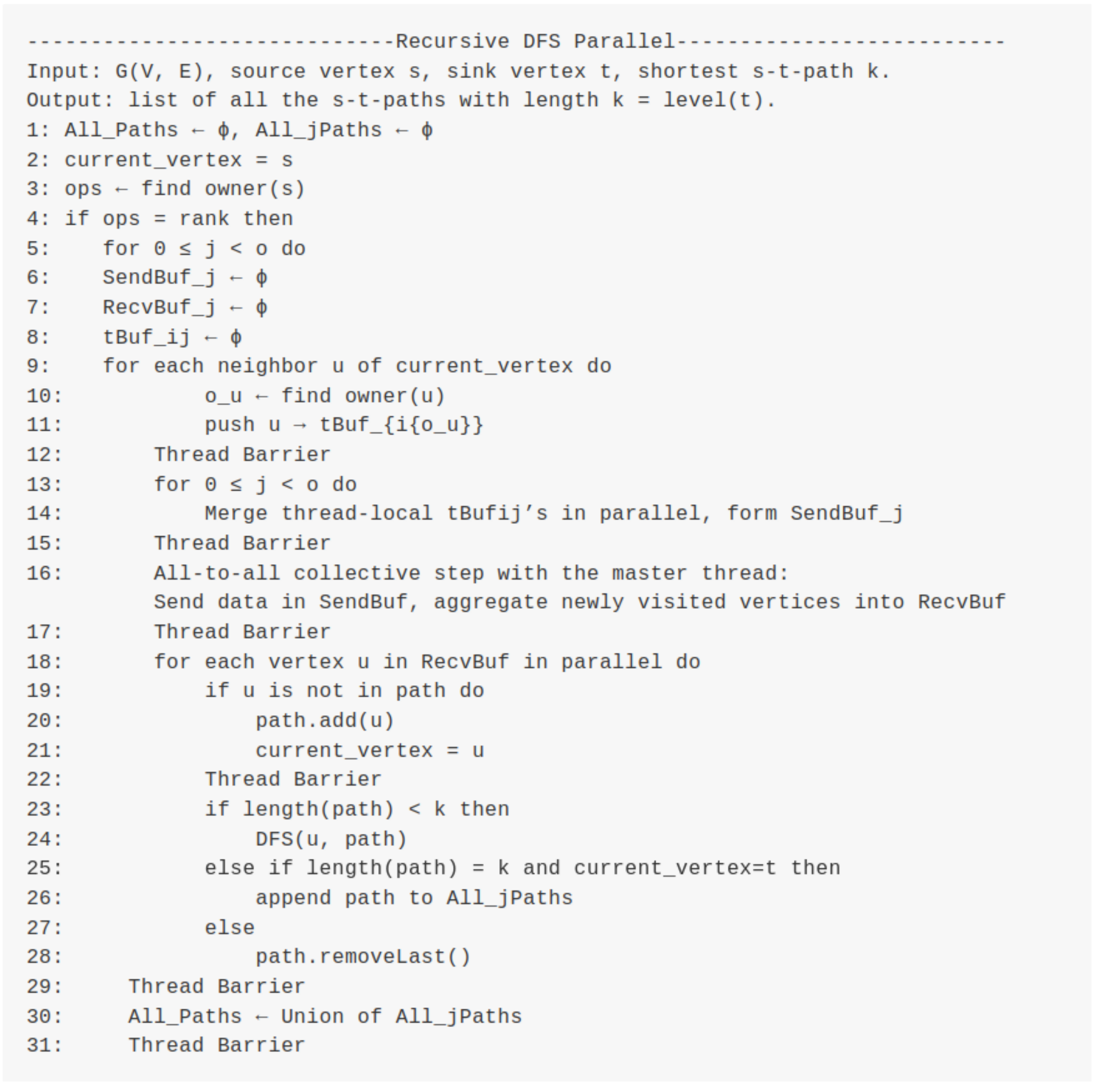
A.1 Jonas

Jona's responsibilities in this project were centered around the development and the implementation of both the sequential and the parallel version of Dinic's algorithm in C, though he also conducted preliminary research into which algorithm we should actually use to achieve our goal (parallelisation of a solution to the Maximum Flow problem) as well as skimming through data on different graph processing frameworks with an emphasis on implementations following the MapReduce programming scheme [PD11]. For his implementation of the algorithm, he used a sequential implementation of Dinic's algorithm in C++ and adapted it to accomodate C's lack of built-in data structures and automatic memory management. He then split the work on the parallelisation attempt into BFS and DFS as well as brainstorming optional features which ultimately didn't make it into my implementations. To model the parallelisation approach to BFS, he used diagrams.net to create figures illustrating the inner workings of his parallel algorithm idea. Lastly, He employed openMPI to parallelise at least the BFS portion of the algorithm due to time constraints. He also wrote sections 3 and 5.1, and a part of the abstract.

A.2 Zoya

First, she initiated the project by proposing the problem in graph theory and played a pivotal role in refining the project title and objectives. She dedicated significant time to comprehensively understanding the mathematical concepts underpinning the problem. Additionally, she participated in the process of selecting the most suitable algorithm for the project's goal. This involved conducting research to identify the optimal algorithm for parallelizing a solution to the Maximum Flow problem. Once the algorithm was decided upon, she articulated the problem's precise nature and clarified the specific requirements and tasks for the project. In terms of implementation, she developed the initial version of the algorithm in Python, which served as the foundational code. This Python code subsequently underwent translation and modification by Jonas to C, enabling the team to leverage the desired programming language. Furthermore, she contributed by creating a pseudocode for parallelizing the algorithm, laying out a framework for the parallelization process. She also created all the graphs used as input for the algorithms, providing the necessary data to test and refine our solutions. Lastly, she took charge of the project's evaluation phase, where the performance and efficiency of the implemented algorithms were tested. She also wrote sections 1, 2, 4, 5.2, 6, and a part of abstraction.

B Code Samples



```
----------------------------Recursive DFS Parallel-------------------------
Input: G(V, E), source vertex s, sink vertex t, shortest s-t-path k.
Output: list of all the s-t-paths with length k = level(t).
1: All_Paths ← ϕ, All_jPaths ← ϕ
2: current_vertex = s
3: ops ← find owner(s)
4: if ops = rank then
5:    for 0 ≤ j < o do
6:    SendBuf_j ← ϕ
7:    RecvBuf_j ← ϕ
8:    tBuf_ij ← ϕ
9:    for each neighbor u of current_vertex do
10:           o_u ← find owner(u)
11:           push u → tBuf_{i{o_u}}
12:       Thread Barrier
13:       for 0 ≤ j < o do
14:           Merge thread-local tBufij’s in parallel, form SendBuf_j
15:       Thread Barrier
16:       All-to-all collective step with the master thread:
          Send data in SendBuf, aggregate newly visited vertices into RecvBuf
17:       Thread Barrier
18:       for each vertex u in RecvBuf in parallel do
19:           if u is not in path do
20:               path.add(u)
21:               current_vertex = u
22:           Thread Barrier
23:           if length(path) < k then
24:               DFS(u, path)
25:           else if length(path) = k and current_vertex=t then
26:               append path to All_jPaths
27:           else
28:               path.removeLast()
29:     Thread Barrier
30:     All_Paths ← Union of All_jPaths
31:     Thread Barrier
```

Figure 10: Pseudocode idea for a parallelised DFS algorithm