Universiteit Leiden
The Netherlands

Bachelor Computer Science

Fineman's Parallel Digraph Reachability in Practice

Tim Zuijderduijn (s3620166)

Supervisors:
A. W. Laarman & S. Hegeman

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

04/08/2025

Abstract

As demand for energy efficiency increases, it is important to look into parallel programming, as this can give rise to better energy efficiency. This thesis focuses on the implementation of Fineman's parallel algorithm for directed graph reachability that, with high probability, reduces the diameter of a graph by using shortcuts. The algorithm has only been proven to work fast and efficiently in theory, but not in practice. Therefore, we ask the question of how well Fineman's algorithm can be implemented in practice and how well the modified parallel BFS component of this algorithm performs. The thesis presents a parallel description and partial implementation of Fineman's algorithm written for the GPU. Afterwards, part of the implementation is evaluated on large graphs by measuring runtime. The results show that in practice the algorithm has the potential to bring about improvements in energy efficiency.

Contents

1 Introduction

Today, one cannot deny the need for energy efficiency with regard to computing. Several reasons can be named, such as keeping up with higher computing power or the importance of minimizing damage to our climate. Research has shown that parallel GPU implementations can be more energy efficient than the usual CPU implementations [7]. If processing power continues to increase, it is important to pay attention to improvements in energy efficiency. For this reason, it is interesting to delve deeper into parallel algorithms that are faster and can bring about more efficient energy usage.

This thesis focuses on a parallel algorithm for directed graph reachability. The parallel algorithm in question was given by Fineman in 2017 [5]. We focus on understanding his algorithm and the pseudocode that he has provided in his paper. We attempt to implement this algorithm on the GPU. Finally, we test a part of the algorithm and see if it performs well compared to, for instance, a normal breadth-first search. The research questions to support this process are the following:

How well can Fineman's parallel algorithm of digraph reachability be implemented on the GPU? Furthermore, how well does the algorithm's graph search perform in practice?

2 Background

This section provides an explanation for several concepts that are important for understanding the thesis. First, we look at directed graph reachability in more detail. Afterwards, we explore what parallelization means and give an explanation of CUDA, which has been used to implement the algorithm on the GPU. Lastly, a detailed explanation of Fineman's algorithm is given.

2.1 Directed graph reachability

To understand the problem at hand, it is important to understand graph theory. In graph theory, a graph is defined as $G = (V, E)$, where V is a set of vertices, nodes, or points, and E is a set of edges [15]. Edges can be seen as a pair of vertices, a connection between two vertices. Two vertices are adjacent if there exists an edge between them. Furthermore, there is a distinction between undirected and directed graphs. The set E of an undirected graph $G = (V, E)$ consists of unordered pairs, and the set E' of a directed graph $G' = (V', E')$ consists of ordered pairs. In other words, in undirected graphs, the edges do not have a specific direction, in directed graphs, the edges do have a specific direction and are one-way. See Figure 1 for an example of a directed graph.

The number of edges emerging from a vertex is called the degree of that vertex. In the case of a directed graph, we can make a distinction between the in-degree and out-degree of a vertex. Respectively, these are the edges that come into a vertex and the edges that come out of a vertex. Another important concept with graphs are paths, which is a sequence of edges [15]. For example, a path from Figure 1 could be (a, b, c, d, c). A graph is connected if and only if there exists at least one path between two vertices. Furthermore, a graph is strongly connected if there exists a path between each vertex of the graph. Moreover, the diameter of a graph is the longest shortest path of that graph.

$$\begin{tikzcd}
a \arrow[r] & b \arrow[r] & c \arrow[r, bend left] \arrow[d] & d \arrow[l, bend left] \arrow[d] \\
& e \arrow[u] & f \arrow[l] & g
\end{tikzcd}$$

Figure 1: An example of a directed graph.

Two important algorithms can be used to traverse a graph, namely, depth-first search (DFS) and breadth-first search (BFS) [15]. Both algorithms start from a so-called root vertex. DFS walks through unvisited vertices as long as it can continue, and BFS walks through all adjacent unvisited vertices layer-per-layer. As an example, starting from a in Figure 1, a DFS walk could be (a, b, c, d, g, f, e) and a BFS walk could be (a, b, c, d, f, g, e). In particular, BFS uses a queue called a frontier. At the start of the BFS, the root vertex is inserted into the frontier, after which the same is done with the adjacent vertices of the vertices in the frontier. This process is repeated until all the vertices have been visited.

In Fineman's paper, directed graph reachability is referred to as the single-source reachability problem for directed graphs. They state: "Given a directed graph $G = (V, E)$ and a source vertex $s \in V$, identify the set of vertices reachable by a directed path originating at s [5]". Naturally, we can infer that a graph traversal algorithm could be used to solve this problem. Sequentially, BFS or DFS can be used [5], but parallelly, other algorithms are used, which is discussed in later sections.

2.2 Parallelization

Conventionally, computations are done serially, meaning tasks are done one by one. Computing like this consumes a lot of time and energy [7], which is where parallelization comes in. Parallelization is done by performing as many of the desired computations as possible in parallel [6]. In practice, sizable problems are often divided into several smaller problems that can be solved at the same time. We are interested in the parallelization of algorithms, creating parallel algorithms. With respect to parallel algorithms, it is possible to achieve sublinear runtimes, which is again beneficial for energy efficiency. This is generally achieved by using a polynomial number of processors in some parallel computational model, such as PRAMs [6].

In theory, there exist some important terms to determine the efficiency of a parallel algorithm. In his paper, Fineman describes the work and span [5]. The work of a parallel algorithm is the total number of primitive operations that are performed; what these operations entail is discussed later. A parallel algorithm is work efficient if, for a given problem, it asymptotically has $O(T^*(N))$ work, where $T^*(N)$ is the best known sequential running time for that given problem [5]. $\tilde{O}$ is the notation for nearly work efficient time, where $\tilde{O}$ hides logarithmic factors. The span of a parallel algorithm is defined as the length of the longest sequence of operations that are executed sequentially.

On the hardware level, parallel tasks are usually done on GPUs as opposed to CPUs. The reason for this is a difference in the structure between the two units. A CPU typically has a smaller number of cores, whereas a GPU has a large number of cores. Intuitively, the CPU is designed for latency, and the GPU is designed for throughput [12]. Usually, the GPU performs the same task on a large amount of data, as opposed to the CPU, which handles individual tasks, although with more speed. On the GPU, this is called a single-program, multiple-data programming model [12]. GPU cores are usually independent of each other and cannot communicate directly. There is a shared global memory to which the cores can read and write, thus allowing communication.

On the software level, there are multiple programming interfaces that allow parallel computing to be performed. Initially, these were less general purpose, mainly graphics APIs that lacked abstraction [12]. However, over the years, more high-level programming languages have become available. Intuitively, this multipurpose programming is called General-Purpose Computing on Graphics Processing Units, or GPGPU for short [12]. We are interested in one specific parallel programming language, namely NVIDIA's CUDA C++, which is looked at later.

In parallel programming, there exist multiple primitives that are important for creating parallel applications, namely scatter, gather, map, reduce, and scan [12]. Scatter and gather are the actions of writing or reading from a specific location in global memory. Mapping is the application of an operation to all elements in a given collection of data. Reduction is the action of reducing a collection of values into a single value. This is done using a binary associative operation, for example, addition. Lastly, scanning is the action of taking an array and returning the same array except that every element is an addition of the current element and all the elements that came before. More specifically, it returns an array A in which each element $A[i]$ is the sum of $A[0]$ through $A[i]$.

2.2.1 CUDA

NVIDIA CUDA [11] is a toolkit that can be used to utilize the GPU when executing code. It not only provides a parallel programming language, but also GPU-accelerated libraries, debugging tools, and optimization tools. More specifically, we are using CUDA C++. As the name implies, this is a programming language that is very similar to C++ and it includes new syntax for GPU utilization.

Using CUDA C++, one can define functions that can be executed in parallel on the GPU; these are called kernels. In the documentation, a sequence of tasks is called a thread [11]. In this way, we can say that a task can be divided into multiple threads, in our case CUDA threads. In general, given N different CUDA threads, a kernel is executed N times in parallel by each of these threads. Multiple CUDA threads can form a block, and each CUDA thread in such a block has a unique identifier, namely, a thread ID, which can be accessed through built-in variables in the kernel. The thread ID is a 3-tuple which can be accessed by calling for the attributes x, y, or z of the 3-tuple.

The structure of CUDA threads is implemented in such a way that it can be used naturally with containers such as vectors, matrices, or volumes. As mentioned above, multiple threads can form a one-, two-, or three-dimensional block, called a thread block [11]. Usually, there are multiple thread blocks, which can also form a one-, two-, or three-dimensional structure, namely a cluster of thread blocks. In practice, this is necessary, as on current GPUs, a thread block may contain only up to 1024 threads [11]. Like independent threads, different thread blocks are also independently executed. Therefore, each thread block also has a unique 3-tuple identifier, namely the block ID. Usually, there are also multiple thread block clusters that together form a grid.

There are different levels of memory that can be accessed by CUDA threads, a memory hierarchy [11]. At the lowest level, there is the local memory of a single CUDA thread, which can only be accessed by threads in the same warp, where a warp is a bundle of 32 threads. Thread blocks have their own shared memory, which can be accessed by all CUDA threads inside the thread block. A thread block can access the shared memory of other thread blocks when they are in the same cluster. At the highest level, there exists a global memory that can be accessed by all CUDA threads. Important to note is that the higher the communication, the slower the communication gets. Using shared memory is slow, so, in general, it is good practice to avoid this.

As is evident, CUDA threads are executed on the GPU, while normal code is executed on the CPU. In this case, the CPU is called the host and the GPU is called the device [11]. There exists a physical gap between the host memory and the device memory, which needs to be bridged manually to some extent; the GPU has its own RAM, separate from the CPU. For instance, if we want to pass an array to a CUDA kernel, the needed memory needs to be manually allocated, and the data from the array needs to be manually copied from the CPU to the GPU. The allocated memory also has to be manually released at the end, similar to how memory allocation is usually done in C++.

To simplify the programming process, we can use Thrust, which is a parallel template library for use with CUDA C++ [1]. This library, for one, simplifies the process of bridging the gap between host and device memory as described above. This is done by including a container called a device vector, which stores memory on the device side. Moreover, Thrust contains a lot of kernels for tasks such as sorting, scanning, set operations, sequence generation, and much more. In this way, we do not have to reinvent the wheel when we want to use one of these tasks in our code, as it is included in Thrust.

2.3 Fineman's algorithm

As mentioned, the problem to be tackled is directed graph single-source reachability. There exist simple sequential solutions, but different solutions are necessary in parallel. Fineman mentions two parallel algorithms that come to mind, namely parallel BFS and transitive closure [5]. Parallel transitive closure is highly parallel [8], but is not work efficient, it has work of about $\tilde{O}(n^{2.372869})$ [5], where n is the number of vertices of a graph. Parallel BFS is work efficient, but the span is linear. Given a root vertex v, the span of parallel BFS starting from v is asymptotically equal to the longest shortest path starting in v. In general, it is better to have a lower span because this allows for more efficient parallelization and less dependency on sequential operations. It is evident that an algorithm that is both work efficient and has a low span is needed for better energy use.

As an improvement over the algorithms mentioned above, Fineman presented a randomized parallel algorithm for the problem of directed digraph reachability [5]. He showed that, with high probability, the algorithm has nearly linear work and span, namely $\tilde{O}(m)$ work and $\tilde{O}(n^{2/3})$ span, where m is the number of edges and n the number of vertices. The general idea of the algorithm is that, by introducing shortcuts, the diameter of a given graph is reduced with high probability. A shortcut is an edge between two vertices, where one vertex can initially reach the other vertex through a path. Again, the diameter is the longest shortest path of a graph. The goal of reducing the diameter of a graph is to be able to perform a faster parallel BFS to solve the reachability problem. The lower the diameter, the lower the span of the parallel BFS. The paper also includes mathematical proofs regarding the complexity of the algorithm, but for the sake of this thesis, we largely skip over those matters.

In his paper, Fineman presents three different versions of his algorithm. The first two versions consist of one subroutine. The last one consists of two, where the first subroutine is called in the second subroutine. There are a few terms that are commonly used throughout the paper with respect to these algorithms. Usually, the subroutines are given a graph $G = (V, E)$, where V is the set of vertices of the graph and E is the set of edges of the graph. For a given graph G, n is the number of vertices, and m is the number of edges.

2.3.1 First versions of the algorithm

The first subroutine shows a basic idea of the algorithm [5, Algo. 1]; see Figure 2. It is a recursive algorithm that takes a graph $G = (V, E)$ as input. The subroutine immediately returns an empty set if given G has no vertices. Otherwise, a random pivot vertex x is selected from V. From this pivot, two sets are derived, namely R^+ and R^-. Respectively, these are the set of vertices that can be reached from x and the set of vertices that can reach x. Afterwards, the set of shortcuts S is generated. All pairs of x and vertices of R^+ and all pairs of vertices of R^- and x are included in S. Then, the sets of V_F, V_B, and V_U are generated. V_F is the set of all vertices that can be reached from x but cannot reach x. In contrast, V_B is the set of all vertices that can reach x but cannot be reached from x. V_U is the set of all vertices that are not in R^+ or R^-, basically all vertices that cannot be reached from x or reach x. The subroutine then recurses with $G[V_F]$, $G[V_B]$, and $G[V_U]$. Here, $G[V_x]$ means that G only includes the vertices in V_x. The subroutine returns S, along with the return values of the three recursion calls.

Algorithm 1 Sequential algorithm for shortcutting

```
SeqSC1($G = (V, E)$) :
if $V = \emptyset$ then return $\emptyset$
select a pivot $x \in V$ uniformly at random
let $R^+$ denote the set of vertices reachable from $x$
let $R^-$ denote the set of vertices that can reach $x$
$S := \{(x, v) | v \in R^+\} \cup \{(u, x) | u \in R^-\}$
$V_F := R^+ \setminus R^-$
$V_B := R^- \setminus R^+$
$V_U := V \setminus (R^+ \cup R^-)$
return $S$ ∪ SeqSC1($G[V_F]$) ∪ SeqSC1($G[V_B]$) ∪ SeqSC1($G[V_U]$)
```

Algorithm 2 Modified sequential algorithm for shortcutting

```
SeqSC2($G = (V, E)$) :
if the recursion depth is $\lg n$ then return $\emptyset$
$S := \emptyset$
while $V \neq \emptyset$ do
    select a vertex $x \in V$ uniformly at random
    $R^+ := R^+(G, x)$
    $R^- := R^-(G, x)$
    $S := \{(x, v) | v \in R^+\} \cup \{(u, x) | u \in R^-\}$
    $V_F := R^+ \setminus R^-$
    $V_B := R^- \setminus R^+$
    $V_U := V \setminus (R^+ \cup R^-)$
    $S := S$ ∪ SeqSC2($G[V_F]$) ∪ SeqSC2($G[V_B]$)
    $G := G[V_U]$
return $S$
```

Figure 2: Algorithm 1 and 2 from Fineman's paper [5].

In his paper, Fineman mentions that Algorithm 1 is very similar to the divide-and-conquer algorithm by Coppersmith et al. [3], the only difference being that shortcutting is used. From this he deduces that his algorithm also has $O(m \log n)$ runtime like the aforementioned algorithm. Moreover, the algorithm appears to reduce the diameter of a graph, with high probability given the randomization of pivot x. The problem with Algorithm 1 is that it is difficult to parallelize in its current form. Finding R^+ and R^- is the single-source reachability problem that the paper tries to solve. The solution for this is to limit the search distance, which is included in the last algorithm [5, Algo. 3].

The second subroutine [5, Algo. 2] is a slight modification of the first subroutine; see Figure 2. The first difference is that, instead of immediately returning an empty set if $V = \emptyset$, it returns an empty set if the recursion depth is equal to $\lg n$. However, it eventually returns the empty set by returning S if $V = \emptyset$, since it does not pass through the while loop when $V = \emptyset$. The other difference is the way the recursion is performed, V_U being excluded as a recursive call. V_U is instead included by altering G to only include vertices from V_U, essentially replacing `SeqSC1`$(G[V_U])$ from Algorithm 1 with a loop. Algorithm 2 improves on Algorithm 1 by truncating the recursion depth and is a step in the right direction when it comes to parallelization, but is not yet ready to be parallelized [5].

There are two obstacles that Fineman mentions with respect to the first two algorithms [5]. The first is the graph search when finding R^+ and R^-. It is possible for this search to have a linear span and complexity instead of logarithmic. The natural solution is to limit the distance of searches, but this introduces some other problems, which is looked at in the next section. The second obstacle is one that Fineman describes using an example. If a given graph G has no edges, the loop in Algorithm 2 still requires at least $\Omega(n)$ iterations. The solution for this problem is to introduce multiple pivots at once, which is discussed further in Section 4.

2.3.2 The main algorithm

In this section, we look at the main algorithm, namely Algorithm 3 [5, Algo. 3], see Figure 3. As stated above, limiting the search distance is a solution that results in better runtime and complexity. The problem that arises is one that Fineman explains with another example [5]. Say we have a pivot x, a search distance D, and a path $(v_0, v_1, ..., v_\ell)$ where ℓ is the distance of the path. There is a possibility that all vertices v_{2k} can be reached from x, but that all vertices v_{2k+1} cannot be reached from x, given $0 \leq k \leq \ell/2$. For technical reasons, this would divide the even and uneven vertices into V_F and V_U, respectively. Fineman deduces that this would split the path into $\Theta(\ell)$ pieces, with no potential to shortcut between these pieces later on.

The solution Fineman gives for the problem described above is to slightly extend the search distance [5]. Given a distance dD, all vertices reached within dD are called core vertices. Then, given the new distance $(d+1)D$, all vertices reached within this new distance are called fringe vertices. When shortcuts are created, the set of fringe vertices and the set of core vertices are combined. For technical reasons, this solves the problem of the path splitting into even and uneven components. As can be seen in Figure 3, fringe sets have been included in Algorithm 3. Here, the fringe set only includes the vertices reached by the fringe search, as vertices from the core search are excluded from the fringe set.

Algorithm 3 Shortcutting algorithm with distance-limited searches

```
ParSC($G = (V, E), h$) :
1: if $h = 0$ then return $\emptyset$
2: $S := \emptyset$
3: randomly permute $V$, giving vertex sequence $X = x_1, x_2, ..., x_{|V|}$. Mark each $x_j$ live.
4: split $X$ into subsequences $X_1, X_2, ..., X_{2k}$, with $|X_i| = |X_{k-i+1}| = \lfloor (1 + \epsilon_\pi)^i \rfloor$ for $i < k$ and
   $|X_k| = |X_{k+1}| \leq \lfloor (1 + \epsilon_\pi)^k \rfloor$
5: for $i := 1$ to $2k$ do
6:     $d_{min} = 1 + hN_kN_L - iN_L$
7:     $d_{max} = d_{min} + N_L - 1$
8:     choose random $d \in \{d_{min}, d_{min} + 1, ..., d_{max} - 1\}$
9:     for each live $x_j \in X_i$ do
10:        $R_j^- := R_{dD}^-(G, x_j)$
11:        $R_j^+ := R_{dD}^+(G, x_j)$
12:        $F_j^- := R_{(d+1)D}^-(G, x_j) \backslash R_j^-$
13:        $F_j^+ := R_{(d+1)D}^+(G, x_j) \backslash R_j^+$
14:        $S := S \cup \{(x_j, v) | v \in R_j^+ \cup F_j^+\} \cup \{(u, x_j) | u \in R_j^- \cup F_j^-\}$
15:        append a tag of $j$ to all vertices in $R_j^+ \cup R_j^-$
16:    for each live $x_j \in X_i$ do
17:        remove vertices with tag$< j$ from $R_j^+$, $R_j^-$, $F_j^+$, $F_j^-$
18:        $V_{F,j} := R_j^+ \backslash R_j^-$
19:        $V_{B,j} := R_j^- \backslash R_j^+$
20:        $S := S \cup \texttt{ParSC}(G[V_{F,j} \cup F_j^+], h - 1) \cup \texttt{ParSC}(G[V_{B,j} \cup F_j^-], h - 1)$
21:    mark all vertices in $\bigcup_j (R_j^+ \cup R_j^-)$ as dead in $X$
22:    $V_U := V \backslash \bigcup_j (R_j^+ \cup R_j^-)$
23:    $G := G[V_U]$
24: return $S$
```

Figure 3: Algorithm 3 from Fineman's paper [5].

The inclusion of fringe vertices does create another problem, namely the possibility of fringe vertices being active in multiple sub-problems, which would have a negative impact on the progress bound [5]. The initial solution for this is to select a random search distance d from a range $d \in \{1, 2, ..., N_L - 1\}$, where N_L stands for the number of layers. To expand on this solution, Fineman notes that it is important that the search distance does not increase for each iteration of the algorithm [5]. To account for this, the range is modified such that $d \in \{d_{min}, d_{min} + 1, ..., d_{max} - 1\}$. Here, d_{min} decreases every time the algorithm recurs with the help of h, and d_{max} is calculated using d_{min} and N_L, see Figure 3. N_k is also included as a factor in d_{min}, which is discussed later.

As mentioned in the previous section, the solution of the second obstacle regarding the first two algorithms is to initiate searches from multiple pivots [5]. This solution is also beneficial for parallelization. As we can see in Figure 3, this has been included in the design of Algorithm 3. First, a number of random pivots have to be chosen. The first step is to change the order of the vertices in V to obtain a vertex sequence X. Afterwards, X is divided into several subsequences that increase in size the first $k - 1$ times and decrease in size until $2k$. What k entails is discussed later. When searching, the core and fringe vertices found using a pivot x_j are included in a corresponding set denoted with j. The found vertices are assigned a tag of j.

After searching, for every vertex v found using x_j, v is removed from the corresponding core and fringe sets if it has a tag lower than j. In other words, if v has been reached by a pivot earlier than x_j, it is not used, as only the first core search to have found v is used. Afterwards, $V_{F,j}$ and $F_{B,j}$ are calculated; these are the forward and backward sets. The algorithm recurs using a graph with vertices from the forward and fringe sets, and then with the backward and fringe sets. Moreover, all vertices are first marked as live. Vertices that are reached by searching are marked as dead, meaning that these can be ignored by searches after the recursion. This way of handling vertices can also be applied to removing vertices from the graph. Instead of modifying the graph explicitly to exclude dead vertices, it is more efficient to simply ignore dead vertices [5].

Algorithm 4 Diameter reduction with distance-limited searches

```
   ParDiam($\hat{G} = (\hat{V}, \hat{E}), \gamma$) :
1: $G' = (V', E') := \hat{G}$
2: for $i := 1$ to $\Theta(\log n)$ do
3:     for each $j \in \{1, 2, ..., \Theta(\gamma \log n)\}$ do
4:         $S_j :=$ ParSC($G', \lg n$), aborting if number of shortcuts or work exceeds bounds
5:     $E' := E' \cup (\bigcup_j S_j)$
6: return $G'$
```

Figure 4: Algorithm 4 from Fineman's paper [5].

We now look at the terms N_k and k. Both terms depend on ϵ_π, which has a value such that $0 < \epsilon_\pi \leq 1$ and is chosen beforehand. The paper describes N_k as the maximum number of iterations needed in the for loop of Algorithm 3 [5]. Moreover, k is chosen to be large enough to include all vertices, and mathematically, this is described as a geometric series [5]. The first k values are summed as $\sum_{i=1}^{k} \lfloor (1+\epsilon_\pi)^i \rfloor$. The last k values are similar to the first k values but going downward. In practice, the sum of the first and last k values can be calculated using the same summation. Together, they form $2\sum_{i=1}^{k} \lfloor (1+\epsilon_\pi)^i \rfloor$. The value k is the smallest value such that $2\sum_{i=1}^{k} \lfloor (1+\epsilon_\pi)^i \rfloor \geq |V|$. This can be further calculated using $\sum_{i=1}^{k} r^i = \frac{r(1-r^k)}{1-r}$, where $r = \lfloor (1+\epsilon_\pi)^i \rfloor$. We can deduce that $2(\frac{r(1-r^k)}{1-r}) \geq |V|$, from which we get $k \leq \log_r(\frac{2r-|V|+|V|r}{2r})$.

To actually achieve diameter reduction using Algorithm 3, Fineman mentions that multiple passes are needed [5]. To do this, a second subroutine is needed to invoke Algorithm 3, see Figure 4. Algorithm 4 takes a graph $\hat{G} = (\hat{V}, \hat{E})$, which is the original input graph, and returns a graph G' to which diameter reduction has been applied. Furthermore, γ is given to the algorithm, which is a failure probability and has a value such that $\gamma \geq 1$. For $\Theta(\log n)$ iterations, Algorithm 3 is executed independently $\Theta(\gamma \log n)$ times. After each iteration, all the shortcuts found in that iteration are added to G'.

3 Related Work

Since 2017, there have been several improvements to Fineman's work. In 2019, Jambulapati et al. showed a parallel algorithm that, with high probability, solves the problem of directed graph reachability with better work and span than Fineman's algorithm [9]. More specifically, it has $\tilde{O}(m)$ work and $n^{1/2+o(1)}$ span. Similarly, their algorithm attempts to reduce the diameter of a given graph using shortcuts. Their algorithm is also comparable to that of Fineman's; the difference lies in the way they handle fringe vertices. Instead of recursing with the core and fringe vertices lumped together, they handle them separately. Moreover, they separately track the recursion depth of the fringe vertices to account for recursion bounds.

In 2020, a group that includes Fineman expanded on the algorithm mentioned above. That is, Cao et al. expanded the algorithm by Jambulapati et al. to solve single-source shortest-path problems [2]. For some choice of parameter $\rho \in [1, \sqrt{n}]$, their algorithm has $\tilde{O}(m\rho^2 + n\rho^4)$ work and $\frac{n^{1/2+o(1)}}{\rho}$ span. Their algorithm is tuned to find a hopset, which is a set of weighted edges that approximates the shortest-path distances by paths of at most a certain length. Furthermore, when it comes to undirected graphs, efficient parallel algorithms have long been known. In 1982, Shiloach and Vishkin presented a logarithmic parallel connectivity algorithm [13]. Their algorithm has a $O(\log n)$ span. Connectivity is a similar problem to reachability, where the focus lies on whether all other vertices can be reached by a root vertex. In addition, their algorithm also uses shortcuts.

4 Algorithm Parallelization and Implementation

In this section, we look at how to implement Fineman's algorithm in practice. To do this, we first look at a description of how to parallelize the algorithm. Afterwards, we look at the implementation process. Specifically, we look at two BFS implementations that are necessary for an implementation of Fineman's algorithm, namely, the normal parallel BFS and the modified parallel BFS.

4.1 Algorithm parallelization

We look at the parallelization of Algorithms 3 and 4 as described in Fineman's paper [5, Section 5]. A large portion of this description is devoted to how to implement graph searches to find, for example, R_j^+. They give the work and the span of the parallel algorithm, as well as the size of the diameter the algorithm produces. More specifically, the parallel algorithm has $O(m \log_6 n + n \log_{10} n)$ work and $O(n^{2/3} \log_{19} n)$ span, and the diameter of the graph it produces is $O(n^{2/3} \log n)$. They prove this by performing the main algorithm first and then running a standard parallel BFS limited to a certain number of hops. If the diameter reduction is successful, the BFS completes before this limit and the whole process is completed.

4.1.1 Parallelization of graph searches

First, we discuss the parallelization of graph searches. This has been done by using a modified parallel BFS. Given X_i, the modified BFS is the set of all searches from X_i, for example, to obtain R_j^+. A key property that Fineman mentions is that there is a high probability that a vertex is not visited by more than $O(\log n)$ individual searches. Thus, we can abort the modified BFS if a vertex is visited too many times [5]. This version of BFS also uses a frontier as mentioned in Section 2.1. There is also a distinction between the current frontier and the next frontier; the next frontier contains the vertices of the next BFS round. Furthermore, each vertex v is paired with a pivot ID, which is the corresponding pivot in X_i from which the search that found v originated. In this way, a distinction can be made between different searches on the current and next frontier. Before starting the modified BFS, some initializations are performed. First, $\Theta(\log n)$ space is allocated per vertex in order to store its ID tag list. As the modified BFS performs multiple searches at once, such a list is needed to track which pivots have reached which vertices. Afterwards, all live pivots $x_j \in X_i$ are copied to the frontier and, for each live pivot, their corresponding spot in the ID tag list includes itself.

The paper describes a single round of the modified BFS as follows [5]. First, for each vertex in the frontier, the out-degree is calculated. Parallel prefix sums are performed on each of these out-degrees such that each edge has a distinct index in the next frontier. Afterwards, for each edge (u, v) of vertices u in the current frontier, let x_j be the corresponding pivot ID. Check whether v's ID tag list contains x_j. If not present, record v and x_j in the edge slot on the next frontier; otherwise, record null on the next frontier. Then, sort the frontier list by vertex first and pivot ID second, after which duplicates are removed using a compaction pass. Duplication removal is necessary, as it is possible that multiple vertices with the same pivot ID reach the same vertex. Lastly, for each slot j in the next frontier list, v is the corresponding vertex. Check whether this is the first occurrence of v in the next frontier list by checking if $j - 1$ stores a different vertex. If so, scan through the next $O(\log n)$ slots of the next frontier list, for each entry of v, append the corresponding pivot ID to the ID tag list. Given a distance dD, this process is repeated dD times.

When the process described above has been completed, some finalizations are performed [5]. First, the arrays of all the vertices reached by the searches are sorted. For each vertex v reached in these searches, the lowest pivot ID reaching v is identified using parallel prefix sums. These pivot IDs are copied to a new array that can be used for recursive searches. This new array is also sorted by pivot ID. Building $G[V_U]$ explicitly would require too much processing, so the vertices in V should be marked as dead when reached and should be ignored by the modified BFS.

4.1.2 Aborting Algorithm 3

The last issue mentioned is that Algorithm 4 should be able to abort Algorithm 3 when the maximum search distance has been exceeded, which is equal to $O(n^{2/3} \log_{12} n)$ [5]. Aborting simply based on run-time or recursion depth is inadequate. A suggestion from Fineman is to implement Algorithm 3 iteratively instead of recursively, in the same vein as a BFS. Instead of the frontier containing vertices, it contains the different subproblems that would spawn from recursive calls. For each level of recursion, prefix sums are used to add up the total number of vertices across all the corresponding subproblems. Fineman suggests that subproblems from a certain level of recursion can be launched in parallel. Aborting can be performed by checking whether the total number of edges traversed exceeds the work bound.

4.2 Implementation

We now look at the practical implementation of the parallelization described above[1]. Note that the implementation is not entirely finished, however, the modified BFS is complete. We look at the

[1]For the code, see: https://github.com/zimtuijd/thesis-fineman

implementation process and discuss some ambiguities regarding the description of the algorithm and its parallelization. As mentioned, CUDA C++ and Thrust are used for the implementation. Specifically, `nvcc` version 12.9.86 has been used to compile the CUDA C++ code and Thrust version 2.8.2 has been used for certain parallel tasks. In general, Thrust device vectors (`thrust::device_vector`) have been used to store arrays for use on the GPU.

An important thing to consider before coding is the graph representation. In practice, there are several ways to store a graph; the most well-known types are adjacency lists and adjacency matrices [14]. These can be used for both directed and undirected graphs. Both also have trade-offs when it comes to memory usage or efficiency when augmenting the graph. For efficiency sake, we use the compressed sparse row (CSR) representation, which has been used for parallel BFS in a paper by Merrill et al. [10]. This type of representation can store a sparse graph with fewer elements than an adjacency matrix, and an adjacency matrix can be converted into a CSR format. In our case, the representation would consist of two arrays, one for the column indices and one for the row offsets. The column indices contain, as the name suggests, the column in which a nonzero value is located. The row offsets contain the number of nonzero values above a given row j. See the arrays on the left in Figure 5 for an example; the CSR representation of the graph in Figure 1.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\texttt{columnIndices} = [1, 2, 3, 5, 2, 6, 1, 4]$$
$$\texttt{rowOffsets} = [0, 1, 2, 4, 6, 7, 8, 8, 8]$$

$$\texttt{columnIndices} = [1, 2, 3, 5, 2, 6, 1, 4]$$
$$\texttt{edgesOffset} = [0, 1, 2, 4, 6, 7, 8, 8]$$
$$\texttt{edgesSize} = [1, 1, 2, 2, 1, 1, 0, 0]$$

Figure 5: Adjacency matrix and CSR representation of Figure 1, and the modified CSR format.

4.2.1 Parallel BFS

Before the modified BFS or the rest of Fineman's algorithm, work had to be done on another type of BFS. As mentioned, the parallel algorithm uses a bounded parallel BFS to see if the algorithm has reduced the diameter to an adequate level. This implementation is simple; take a standard parallel BFS and keep track of the number of completed rounds. If more than $O(n^{2/3} \log n)$ rounds are needed before completing, return and continue with the algorithm. Otherwise, stop the algorithm. Important to note is that this parallel BFS uses only a single root vertex.

To avoid reinventing the wheel, existing code has been used for the parallel BFS, and the license for this has been included in the repository. The code has been simplified using Thrust. A major advantage of this existing code is that it refers to the paper by Merrill et al. [10], and thus it already uses the CSR graph representation. It differs slightly from the format described above, as the row offsets have been split into two arrays, namely `edgesOffset` and `edgesSize`. The first is for the actual row offsets and the second, given an element $\texttt{edgesSize}[i]$ where $0 \leq i < |\texttt{edgeOffset}|$, is the difference $\texttt{edgesOffset}[i+1] - \texttt{edgesOffset}[i]$. Note that these arrays contain one less element than the regular row offsets array; see the arrays on the right in Figure 5 for an example.

The existing parallel BFS implementation uses four different CUDA kernels, namely `nextLayer(...)`, `countDegrees(...)`, `scanDegrees(...)`, and `assignVerticesNextQueue(...)`. All of these kernels have two things in common. The first is the way the thread ID is calculated, they only use one dimension, and it is calculated using $\texttt{thid} = \texttt{blockIdx.x} * \texttt{blockDim.x} + \texttt{threadIdx.x}$. The second is that the thread ID is compared to a given size, in this case always the current frontier size. More specifically, they check if $\texttt{thid} < \texttt{queueSize}$, described later. This prevents overflow, as only threads with thread IDs numbered 0 to $\texttt{queueSize} - 1$ are needed. In the end, three of the four kernels have been used, as `scanDegrees(...)` has been replaced with Thrust API, which we discuss later.

There are several arrays that are used by the four kernels, all of them use integer values. These are the following:

· `d_adjacencyList`: column indices of the CSR representation.

· `d_edgesOffset`: first part of the row offsets of the CSR representation.

· `d_edgesSize`: second part of the row offsets of the CSR representation.

· `d_distance`: given a certain vertex, the distance from the root vertex. All values are first initialized with `INT_MAX`, except the root vertex, which has 0.

· `d_parent`: given a certain vertex, the adjacent vertex that first reached it. All values are first initialized with `INT_MAX`, except the root vertex, which has 0.

· `d_currentQueue`: the frontier of the current level.

· `d_nextQueue`: the frontier of the next level.

· `d_degrees`: the out-degree of a given vertex in the current frontier.

`nextLayer(...)` is used to update `d_distance` and `d_parent`. It takes `d_adjacencyList`, `d_edgesOffset`, `d_edgesSize`, `d_distance`, `d_parent`, and `d_currentQueue` as input. It also takes `level` and `queueSize` as input, where `queueSize` is the size of the frontier. Given a vertex u in the current frontier, all vertices that are directly reached by u are iterated over. Given such a vertex v, $\texttt{level} + 1 < \texttt{d\_distance}[v]$ is checked. In other words, it checks whether v has been reached by a vertex, and if so, whether v currently has a longer path to the root than through u. If true, it changes the values in $\texttt{d\_distance}[v]$ and $\texttt{d\_parent}[v]$ to correspond with u. The value i is assigned to $\texttt{d\_parent}[v]$, where i is used as a for loop iterator with respect to the row offsets.

`countDegrees(...)` is used to find the out-degree of a certain vertex v. Here, the out-degree of v is restricted to vertices that have v set as their parent in `d_parent`. It takes `d_adjacencyList`, `d_edgesOffset`, `d_edgesSize`, `d_parent`, `d_currentQueue`, and `d_degrees` as input. It also takes `queueSize` as input. Consider vertex u and a vertex v reached by u as described in `nextLayer(...)`. Given v, it checks whether `d_parent`$[v]$ is equal to i corresponding to u. In other words, it checks if v actually has u as its parent in `d_parent`. It also checks if $u \neq v$ in order to prevent an infinite cycle. If both are true, a value `degree` is incremented by one. In the end, `d_degrees`[`thid`] is assigned `degree`, corresponding to the current frontier.

As mentioned, `scanDegrees(...)` has not been used, even though it is part of the original code. The reason for this is the use of an array called `incrDegrees`. As `scanDegrees(...)` is a CUDA kernel, it has a maximum block size of 1024. To account for multiple blocks, `incrDegrees` had been used to keep track of the sum of degrees for a single block. The problem lies in the implementation of this array, as it is modified on both the host and the device, which did not convert well when changed to a `device_vector`. Thus, `incrDegrees` was removed and `scanDegrees(...)` was exchanged for a Thrust algorithm, namely `thrust::inclusive_scan`. The latter is feasible, as `scanDegrees(...)` basically performs an inclusive scan on `d_degrees`.

Before calling `assignVerticesNextQueue(...)`, `nextQueueSize` is changed to the last value in `d_degrees`, which is equal to the sum of all out-degrees. Afterwards, the kernel does what its name implies, it assigns vertices to `d_nextQueue`. It takes `d_adjacencyList`, `d_edgesOffset`, `d_edgesSize`, `d_parent`, `d_currentQueue`, `d_nextQueue`, and `d_degrees` as input. It also takes `queueSize` and `nextQueueSize` as input. Again, consider vertex u and v as described above. Given v, it checks whether `d_parent`$[v] == i$ and $v \neq u$, similar to the check from `countDegrees(...)`. If both are true, v is inserted into the corresponding position of `d_nextQueue`. This position is calculated using the sum of `d_degrees`$[\texttt{thid} - 1]$ and a counter variable. Here, `d_degrees`$[\texttt{thid} - 1]$ is the total number of out-degrees up to u. The counter goes up to the value in `d_degrees`[`thid`], so in the end, all the values between `d_degrees`[`thid-1`] and `d_degrees`[`thid`] are included in `d_nextQueue` in their corresponding positions.

As a whole, the bounded parallel BFS runs in the following order. First, it checks `level`, if it has reached a value of `maxLevel`, the BFS is aborted. Afterwards, `nextLayer(...)` and `countDegrees(...)` are called, in that order. Then, `thrust::inclusive_scan` is used as described above, after which `nextQueueSize` is set. After this, `assignVerticesNextQueue(...)` is run. Lastly, the `d_currentQueue` and `d_nextQueue` are swapped, including their size variables, and `level` is incremented by one.

4.2.2 Modified parallel BFS

As discussed, Fineman's algorithm uses a modified parallel BFS to find, for example, R_j^+. To implement this, we build upon the parallel BFS code described above and modify it as needed. Two main features have been added, namely the possibility of using multiple starting pivot vertices and the ID tag list. There are four CUDA kernels that are called from the host, these are `modCountDegrees(...)`, `modAssignVNQ(...)`, `numberPivotID(...)`, and `assignPivotID(...)`. Some of the kernels also use kernels themselves; these are device kernels and cannot be called from the host, and we discuss these later on. Similarly to bounded parallel BFS, all kernels called from the host calculate `thid` in the same one-dimensional way. The only difference is that `numberPivotID(...)` and `assignPivotID(...)` compare `thid` with `nextQueueSize` instead of `queueSize`.

There is a difference in the arrays used compared to bounded parallel BFS. Note that `d_distance` and `d_parent` are no longer used. The following arrays are used:

- `d_adjacencyList`: column indices of the CSR representation.
- `d_edgesOffset`: first part of the row offsets of the CSR representation.
- `d_edgesSize`: second part of the row offsets of the CSR representation.
- `d_currentQueue`: the frontier of the current level.
- `d_nextQueue`: the frontier of the next level.
- `d_degrees`: the out-degree of a given vertex in the current frontier.
- `d_IDTagList`: the ID tag list as described in Section 4.1.
- `d_queueID`: given a vertex v in the current frontier, its corresponding pivot vertex ID tag.
- `d_nextQueueID`: given a vertex v in the next frontier, its corresponding pivot vertex ID tag.
- `d_modDistance`: given a certain vertex, the distance from a corresponding pivot in the ID tag list.
- `d_IDTagCount`: given a certain vertex, the number of pivot IDs that have reached it.
- `d_nextQueueIDListNum`: used in accordance with `d_nextQueue` for ID tag list assignment.

`modCountDegrees(...)` is, as the name implies, a modified version of `countDegrees(...)`. This kernel closer matches the parallelization as described in Section 4.1. The difference lies in the absence of `d_distance` and `d_parent`. The reason for this is the inclusion of multiple pivot vertices. Consider u and v as described before. To still check if a vertex has been visited before by some pivot ID, `d_parent`$[v] == i$ is replaced with a device function called `isVisitedByPivotID(...)`. This kernel searches the ID tag list of v for the pivot ID of u. If it is not found, the kernel returns false, which means that the vertex has not been visited before by the corresponding pivot ID. Moreover, interesting to note is the absence of `nextLayer(...)`. The reason for this is that duplicates are removed later on, as described in the parallelization. `d_modDistance` is modified later on, as opposed to `d_distance` in the normal parallel BFS implementation.

Similarly to bounded parallel BFS, `thrust::inclusive_scan` is used to perform a prefix sum on the out-degrees of the current frontier, and `nextQueueSize` is also calculated the same as above. After this, `modAssignVNQ(...)` is called, which is a modified version of `assignVerticesNextQueue(...)`. The difference between the two is the inclusion of the ID tag list and the exclusion of `d_parent`. As such, the kernel also takes `d_IDTagList`, `d_queueID`, and `d_nextQueueID` as input on top of the arrays used in `assignVerticesNextQueue(...)`, except `d_parent` of course. Moreover, `IDTagSize` is given, which is the length of the ID tag list for a single vertex and is equal to $\log n$. Similarly to `modCountDegrees(...)`, only $v \neq u$ is checked, given u and v as described earlier. If true, `isVisitedByPivotID(...)` is used to check whether v has already been reached by the corresponding pivot ID of u. If it has been reached, v is not added to the next frontier. Otherwise, v is added to the next frontier and the corresponding pivot ID of u is added to `d_nextQueueID`.

`numberPivotID(...)` is used to count the number of pivot IDs per vertex and to assign an index in `d_IDTagList` to each vertex in the next frontier. Moreover, by counting, it checks whether a vertex has been visited by more than $\log n$ pivots. It takes `d_nextQueue`, `d_IDTagList`, `d_IDTagCount`, and `d_nextQueueIDListNum` as input. It also takes `nextQueueSize`, `IDTagSize`, and an array of length 1 called `IDTagListOverflow` as input. This last array contains a single boolean. First of all, the kernel checks whether vertex $v =$ `d_nextQueue[thid]` is the first occurrence in `d_nextQueue`. It does this by checking if either `thid` $== 0$ or `d_nextQueue`[`thid`] $\neq$ `d_nextQueue`[`thid-1`]. If either is the case, the next $\log n$ places in `d_nextQueue` are checked for occurrences of v. For each occurrence, `d_IDTagCount`[v] is incremented by one, as each v has a distinct ID tag because of the duplication removal. Moreover, `d_nextQueueIDListNum` is modified for each v to contain the corresponding position in `d_IDTagList` for v. If `d_IDTagCount`[v] exceeds `IDTagSize`, it means that a vertex has been visited by too many pivots, the kernel is then aborted and `IDTagListOverflow` is modified accordingly.

`assignPivotID(...)` is used to update `d_IDTagList` and `d_modDistance`. For each vertex u in the next frontier, the corresponding pivot ID is taken. With help of `d_nextQueueIDListNum`, this pivot ID is inserted in the corresponding position in the ID tag list. The position in the ID tag list is calculated using `IDTagLoc` $= u *$ `IDTagSize` $*$ `d_nextQueueIDListNum`[`thid`]. Moreover, the same position is used to modify `d_modDistance`. This array is parallel to `d_IDTagList`, for each vertex v in the ID tag list, `d_modDistance`[v] contains the distance of v from each pivot ID that has visited v.

There are two additional matters that are handled in between the calls of `modAssignVNQ(...)` and `assignPivotID(...)`. First, `d_nextQueue` and `d_nextQueueID` are sorted by vertex first and pivot ID second. To aid in this, `thrust::zip_iterator` objects and tuples were used. Furthermore, `thrust::stable_sort` was used for sorting itself, it uses the custom function `sort_vertex_ID()` as a comparison operator. `sort_vertex_ID()` takes tuples a and b as input, where a is assumed to be `d_nextQueue` and b to be `d_nextQueueID`. It sorts by a first and b second. Secondly, duplication removal is performed on `d_nextQueue` and `d_nextQueueID`. This is done using `thrust::unique`, which takes the zip iterators used in the sorting and performs a compaction pass. Given $i > 0 > |$`d_nextQueue`$|$, this removes the tuple in position i if it is equal to the tuple in position $i - 1$. An iterator to the last element is returned by `thrust::unique`, which is used to check if any duplicates have been removed. If so, all values after the new bound in `d_nextQueue` and `d_nextQueueID` are reset, as the old values are not explicitly removed by `thrust::unique`. Moreover, the value of `nextQueueSize` is changed accordingly if duplicates were removed.

The following is the order of execution for one round of BFS. First, it checks whether the BFS is bounded and, if it is, whether `level` has surpassed `distance`. Here, `level` represents the maximum search distance dD. If this check returns true, the BFS is aborted. Otherwise, modCountDegrees(...) is run, after which the inclusive scan is performed on d_degrees and nextQueueSize is set. nextQueueSize is compared to the maximal queue size. If it is larger, that means that a vertex has been visited more than IDTagSize times; IDTagListOverflow is set and the BFS is aborted. If the check is satisfied, modAssignVNQ(...) is run. d_nextQueue and d_nextQueueID are sorted and duplication removal is performed on both, as described above. Then, numberPivotID is run, if it detects ID tag list overflow, the BFS is aborted. Otherwise, assignPivotID(...) is run, after which level is incremented by one. Lastly, d_currentQueue and d_nextQueue are swapped, as well as d_queueID and d_nextQueueID.

Two matters are addressed when the modified BFS is finalized. The first matter is related to IDTagListOverflow. The implementation returns the boolean in this array. When this value is false, the second matter does not need to be handled. The second is related to what is described as "sorting the arrays of all vertices reached by the searches" in Section 4.1. To do this, an array called d_IDTagVertices of length $\log n$ is used. For each vertex v in the ID tag list, the lowest ID tag is inserted into d_IDTagVertices[v]

During the implementation process, some ambiguities had to be handled with respect to the description of the modified BFS. One of these is saving the frontier for each BFS round. This implies that a non-constant amount of arrays would have to be accounted for. Instead of doing this, one can look at the ID tag list. By scanning the list of a vertex v, all the pivots that reached this vertex can be found. We can deduce that, given a pivot x_j, we can find all the vertices it has reached by scanning the entire ID tag list. Again, doing this also addresses the "sorting of the arrays of all vertices reached by the searches".

Another ambiguity is how one would store the frontier along with the pivot IDs. It is not explicitly said how this needs to be done. In this implementation, two individual arrays have been used, one for the vertices and one for the corresponding pivot ID. Another way to interpret this is to have an array of pairs or 2-tuples. Then, the array would consist of pairs (v, x_j), where v is a vertex, and x_j its corresponding pivot ID. Implementing pairs can be done in two ways, the first of which is explicitly storing pair objects. The other way is to have an array that is twice as large and stores v in the even positions and x_j in the odd positions.

There is also an ambiguity with regard to the ID tag list. The paper describes that the pivot ID is simply appended to the tag list of the corresponding vertex. In principle, one cannot simply append an element in this implementation. There are two ways in which this can be implemented. The first possibility is that for each corresponding vertex v in d_nextQueue, its ID tag list is scanned to find an empty position for the corresponding pivot ID. The second way is that these positions are calculated beforehand, which has been done in the implementation.

Moreover, there is another oversight regarding duplication removal. As described, the values after the new end of d_nextQueue and d_nextQueueID need to be reset. However, the paper does not seem to take this into account. The reason for this could be that the paper assumes that duplication removal explicitly removes duplicate elements. In practice, this is not always the best way to handle this task, as it can cause memory issues when duplication removal is performed on part of the array.

Lastly, there is an important issue regarding memory usage. With normal parallel BFS, one can assume that the maximum number of vertices in the frontier is equal to the number of vertices in the graph. With the modified parallel BFS, this is not the case. The number of possible vertices in the queue could grow to $n(n-1)$, which is in $O(n^2)$. The reason for this is that duplication removal of the next frontier is only done after counting the degrees. In turn, a vertex v that has not yet been visited by a vertex with pivot ID x_j can be placed in the next frontier multiple times by vertices with the same pivot ID x_j, if these vertices are adjacent to v. In the implementation, the maximum size of the frontier has been set at $n \cdot$ `IDTagSize` to prevent memory issues. However, this is only a temporary solution.

4.2.3 BFS validation

We also want to see whether the BFS implementations do what they are supposed to do. To test this, a way to validate them has also been integrated into the program. A sequential BFS has been included in the program, which can be run on a given graph to obtain two arrays, namely `distance` and `parent`. These are similar to the `d_distance` and `d_parent` arrays from the normal parallel BFS implementation and are compared to validate the results of the parallel BFS. First, sequential BFS and parallel BFS are run on a graph with a root vertex v. If afterward `d_distance` and `distance` contain the same values, the parallel BFS has correctly traversed the graph.

Testing the modified BFS is trickier, as it performs searches from multiple root vertices. Multiple `parent` arrays are needed for this. In order to still be able to validate the modified BFS, such arrays can be obtained by performing a different normal parallel BFS search for each root vertex. The result of each search is saved. After the modified BFS finishes, the array `d_modDistance` described above is used for comparison. For each root vertex v, the ID tag list is traversed to find all instances of v, and the corresponding value in `d_modDistance` are used to build a new `distance` array. When the array has been built, it is compared to the corresponding `d_distance` array from the normal parallel BFS. If all arrays match, the modified BFS has also correctly traversed the graph.

4.2.4 Partial Fineman's algorithm

As mentioned above, the parallelization of Fineman's algorithm has not been completed. This refers to the parallelization of Algorithms 3 and 4 described in Figure 3 and Figure 4. Even still, the code that was finished is included in the repository. Like in Section 4.2.2, there were some ambiguities with regard to the parallelization of this part of the algorithm, which we briefly discuss in this section.

One of the ambiguities was the vague definitions of N_L and N_k. In Fineman's paper, N_L is described as the number of layers and N_k as the maximum number of iterations or as an upper bound on the number of necessary iterations [5]. However, it is unclear what the exact value of these terms would be. This is a problem, as they are essential for calculating d_{min} and d_{max}, from which a search distance dD is chosen at random for the modified BFS to be bounded by. In the implementation, $N_L = \log_{6.5} n$ and $N_k = 2k$ were chosen based on our interpretation of Fineman's paper, but in turn the difference between d_{min} and d_{max} became very small. Multiple definitions of both terms are given [5], another possibility for N_k is $\Theta(\log_4 n)$ and one of the other possibilities for N_L is $\Omega(\log n)$. Therefore, it is left open whether these are the correct values for N_L and N_k.

Another problem was the difference between forward graph searches and backward graph searches, for example, R_j^+ and R_j^-. In our case, performing the forward search using the modified BFS speaks for itself, as no changes need to be made to the graph. However, for backward searches, the graph needs to be transposed. The program uses CSR representation, and to transpose the graph, it needs to be built again. As mentioned above, rebuilding the graph is something that needs to be avoided, so transposing the graph should be done sparingly. A solution is to transpose the graph once during initialization, and to give it as input alongside the normal graph to the parallel implementation of Algorithm 4.

5 Experiments

In this section, an experiment is performed on the implementations given above. First, a description of what the experiment entails is given. Details are given on the setup of the experiment. Afterwards, we discuss the results and try to interpret them.

5.1 Description of the experiment

We now discuss an evaluation of the normal parallel BFS and the modified parallel BFS. As mentioned, Fineman's algorithm is not finished, so this part of the implementation is not used. For the experiment, we want to measure the run-time of the two BFS versions on different graphs and setups. One main difference is that, instead of starting with a single root vertex, both algorithms need to search from multiple root vertices. This is similar to the use case of the modified BFS, which is designed to perform multiple searches at once. Moreover, for reliable results, searches are performed for 100 epochs with a randomized set of root vertices.

Name	**Description**	n (10^6)	m (10^6)	**ID Tag Size**
`wikipedia-2007026`	Wikipedia page links	3.5	45.0	16
`coPapersCiteseer`	Citation networks	0.4	32.0	13
`asia_osm`	Asian road network	11.9	25.4	17
`germany_osm`	German road network	11.5	24.7	17
`eu-2005`	Crawl of the .eu domain	0.8	19.2	14
`wiki-Talk`	Wikipedia talk pages	2.3	5.0	15
`web-BerkStan`	Berkeley-Stanford web graph	0.6	7.6	14
`333SP`	2D Tri-element mesh of car	3.7	22.2	16

Table 1: Matrices used for the experiment, retrieved from the SuiteSparse Matrix Collection [4].

In the implementation, the experiment is performed as follows. First, a set of start vertices is generated randomly. The number of starting vertices is equal to `IDTagSize` defined in Section 4.2.2, the reason for this number is to keep implementation changes to a minimum. Then, the normal parallel BFS is executed once for each starting vertex, taking as input the said starting vertex. The total time of this BFS is calculated by adding the time taken for each execution. Afterwards, the modified parallel BFS takes the entire set of starting vertices as input and is executed once. This process is repeated a certain number of times, as dictated by the number of epochs. In our case, this is 100 times.

For the graphs used, see Table 1. These were taken from the SuiteSparse Matrix Collection of the University of Florida [4]. This collection provides `.mtx` files, which were used as input for the experiment. These are adjacency matrices that can be converted into a CSR representation when the program is initialized. Some of these graphs have also been used by Merrill et al. [10] to test parallel BFS traversal, namely `wikipedia-2007026` and `coPapersCiteseer`.

The machine that has been used for this experiment is from the LIACS facility REL Compute. They provide several machines for specific use cases, which also includes machines for GPU-heavy usage. The machine we used for the experiment has the following specifications:

- CPU: 24 Intel Xeon Silver 4214 cores, clock speed 2.20GHz, 48 threads
- GPU: 2 NVIDIA GeForce RTX3090, 24GB memory (only one used in the experiment)
- RAM: 256GB
- OS: Rocky Linux 9.6 (Blue Onyx)
- CUDA Version 12.9

5.2 Results

As mentioned above, the graphs of Table 1 have been used for the experiment. Both normal parallel BFS and modified parallel BFS were run 100 times on the graphs. The average runtime was deduced from these 100 executions and the speedup was calculated. See Table 2 for the results.

Name	**ID Tag Size**	**Average runtime**		**Speedup**
		Par. BFS	**Aug. Par. BFS**	
`wikipedia-2007026`	16	1412ms	354ms	3.98
`coPapersCiteseer`	13	49ms	34ms	1.44
`asia_osm`	17	204ms	52ms	3.92
`germany_osm`	17	42ms	9ms	4.66
`eu-2005`	14	166ms	161ms	1.03
`wiki-Talk`	15	102ms	361ms	0.28
`web-BerkStan`	14	1028ms	193ms	5.32
`333SP`	16	14900ms	3295ms	4.52

Table 2: Results of the experiment.

As can be seen, the modified parallel BFS wins seven out of eight times. It appears that multiple-source BFS traversal for the modified BFS is faster compared to the regular version. In five out of seven cases, the modified version is at least 3.9 times faster. However, there is one case where the normal parallel BFS beats the modified version by quite some margin, namely on the `wiki-Talk` graph. This is interesting, as this is the graph with the least number of edges that was tested. It is possible that the structure of the graph plays a role in this, but we do not have enough information to be able to draw a conclusion from that.

Although we know too little about the properties of the graphs, there are some interesting differences to be noted. For example, the runtime of `asia_osm` and `germany_osm` differ by some margin, but are of comparable size with respect to the number of vertices and edges. Furthermore, despite having the largest number of vertices, the runtimes of these graphs are quite short compared to other graphs of smaller size. For example, `web-BerkStan` needs significantly more time despite having far fewer vertices and edges. There is also a graph with a significant difference in runtime compared to the rest, namely `333SP`. This is also surprising, as this is not per se the largest graph that has been used for evaluation. Compared to `wikipedia-2007026`, which has a similar number of vertices, the traversal takes a lot longer. All in all, there might be too little detail to see why the runtimes differ like they do, but it is clear that the modified BFS wins when it comes to searches from multiple vertices.

6 Conclusions and Further Research

At the beginning of the thesis, we asked how well Fineman's parallel algorithm of digraph reachability can be implemented on the GPU and how well the algorithm's graph search performs in practice. The second question refers to the modified BFS described in Section 4.2.2. Regarding the first question, it was not always clear how to implement parts of the algorithm. There were some ambiguities with respect to the parallelization of the algorithm. We have made several choices to attempt to achieve the best result regarding the implementation.

To answer the second question, an experiment was performed on the two implementations of parallel BFS. All in all, the results were clear. In almost all cases, the modified parallel BFS performs better than multiple runs of the regular parallel BFS. However, there does seem to be one case where the modified version is outperformed by the regular version. The reasons for this could lie in the properties of the graph, but it remains to be speculated.

There is potential future work with regard to this thesis. First of all, perhaps the rest of the implementation of Fineman's algorithm could be finished. Some improvements could also be made with regard to the implementation in its current form, for instance, with memory usage. It would be interesting to see how the full algorithm would perform in practice. In addition, practical implementations of the algorithms mentioned in Section 3 could be made. Specifically, these are the algorithms in the papers by Jambulapati et al. [9] and Cao et al. [2].

References

[1] Nathan Bell and Jared Hoberock. "Thrust". In: *GPU Computing Gems Jade Edition*. Elsevier, 2012, pp. 359–371. DOI: 10.1016/B978-0-12-385963-1.00026-5.

[2] Nairen Cao, Jeremy T. Fineman, and Katina Russell. "Efficient construction of directed hopsets and parallel approximate shortest paths". In: *Proceedings of the Annual ACM Symposium on Theory of Computing*. Association for Computing Machinery, June 2020, pp. 336–349. ISBN: 9781450369794. DOI: 10.1145/3357713.3384270.

[3] Don Coppersmith et al. *A Divide-and-conquer Algorithm for Identifying Strongly Connected Components*. Tech. rep. 2003. URL: https://escholarship.org/uc/item/1hx5n2df.

[4] Timothy A. Davis and Yifan Hu. "The university of Florida sparse matrix collection". In: *ACM Transactions on Mathematical Software* 38 (1 Nov. 2011), pp. 1–25. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

[5] Jeremy T Fineman. "Nearly work-efficient parallel algorithm for digraph reachability". In: *SIAM Journal on Computing* 49 (5 2020), STOC18500–SOTC18539. ISSN: 10957111. DOI: 10.1137/18M1197850.

[6] Steven Fortune and James Wyllie. "Parallelism in random access machines". In: *Proceedings of the tenth annual ACM symposium on Theory of computing - STOC '78*. ACM Press, 1978, pp. 114–118. DOI: 10.1145/800133.804339. URL: https://dl.acm.org/doi/abs/10.1145/800133.804339.

[7] S. Huang, S. Xiao, and W. Feng. "On the energy efficiency of graphics processing units for scientific computing". In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, May 2009, pp. 1–8. ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5160980.

[8] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., United States, Oct. 1992, pp. 1–41.

[9] Arun Jambulapati, Yang P Liu, and Aaron Sidford. "Parallel Reachability in Almost Linear Work and Square Root Depth". In: *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*. 2019, pp. 1664–1686. DOI: 10.1109/FOCS.2019.00098. URL: https://arxiv.org/abs/1905.08841.

[10] Duane Merrill, Michael Garland, and Andrew Grimshaw. "High-Performance and Scalable GPU Graph Traversal". In: *ACM Transactions on Parallel Computing* 1 (2 Feb. 2015), pp. 1–30. ISSN: 2329-4949. DOI: 10.1145/2717511. URL: https://dl.acm.org/doi/abs/10.1145/2717511.

[11] NVIDIA. *CUDA C++ Programming Guide*. Version 12.9.0. 2025. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[12] J.D. Owens et al. "GPU Computing". In: *Proceedings of the IEEE* 96 (5 May 2008), pp. 879–899. ISSN: 0018-9219. DOI: 10.1109/JPROC.2008.917757.

[13] Yossi Shiloach and Uzi Vishkin. "An O(logn) parallel connectivity algorithm". In: *Journal of Algorithms* 3 (1 1982), pp. 57–67. ISSN: 0196-6774. DOI: https://doi.org/10.1016/0196-6774(82)90008-6. URL: https://www.sciencedirect.com/science/article/pii/0196677482900086.

[14] Harmanjit Singh and Richa Sharma. “Role of Adjacency Matrix & Adjacency List in Graph Theory”. In: *INTERNATIONAL JOURNAL OF COMPUTERS & TECHNOLOGY* 3 (1 Aug. 2012), pp. 179–183. ISSN: 2277-3061. DOI: 10.24297/ijct.v3i1c.2775.

[15] Mikhail Tuzhilin and Dong Zhang. *Introduction to graph theory and basic algorithms*. 2024. arXiv: 2312.11543 [cs.SI]. URL: https://arxiv.org/abs/2312.11543.