UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Bachelor's Project

Rune Nielsen

Implementation of Graph Algorithms in Futhark

Advisor: Troels Henriksen

Date: June 12, 2023

Contents

1 Introduction

1.1 The Futhark Language

Futhark is a statically typed, data-parallel, purely functional array language, with a heavily optimizing compiler. Futharks main purpose is to be a more comfortable way of writing efficient parallel code than in languages like CUDA or OpenCL. The point of Futhark isn't to replace existing general-purpose languages, but instead to be used for small compute intensive parts of a program. Futhark has different compile targets, with the main ones being for generating CUDA or OpenCL GPU code. The code generated with Futhark can easily be integrated with other languages such as Python, C, or Haskell [9].

1.2 The Problem Based Benchmark Suite (PBBS)

The Problem Based Benchmark Suite, shortened PBBS, is a collection of more than 20 benchmarks, grouped into 5 categories. The benchmarks are designed to make it possible to compare different algorithms and implementations of algorithms in different programming languages. To facilitate this, the PBBS suite provides for each benchmark a specification for the input and output of the algorithm, as well as at least one parallel and, for most cases, sequential implementation. The PBBS suite also provides a set of tools for generating the input files and for checking the correctness of the output.

1.3 Irregular Algorithms

For a regular problem, such as sorting an array, it is at each step known which elements are needed, and we can thus fairly easily figure out how to most efficiently parallelize the required operations. With an Irregular problem, the needed elements are irregularly spread over the dataset, or depend on certain conditions. We can thus not easily optimize these problems, as it isn't known what needs to be done in the next step.

1.4 Graph Algorithms

Graph algorithms are algorithms operating on a graph, which is a collection of edges and vertexes, with each edge connecting two vertexes. The edges can either be directed or undirected, meaning it's only possible to go from one vertex to the other one way. The vertexes can be weighted or not depending on the problem statement. Graph algorithms are very irregular, and aare thus a good choice when testing irregular algorithms.

2 The Work/Depth Model of Parallel Computation

When analysing the the efficiency of a sequential algorithm, we often use big O notation to show the asymptotic time complexity of an algorithm. This can be seen as the amount of steps an algorithm needs to take to complete the task. We would eg. say that the time complexity of the merge sort algorithm is $O(n \times log(n))$, meaning it would take in the order of $n \times log(n)$ steps to sort an array of n elements.

However, when using a parallel algorithm, multiple operations can be done at the same time. Given an infinitely parallel computer, we would even be able to do an infinite amount of operations at once, meaning the only thing the time complexity would depend on is the longest

sequence of operations that have to be performed sequentially. There will for some programs always be some amount of sequential steps, where a parallel operation might for example have to wait on the result of a previous parallel operation. The length of this sequence of sequential operations, also called the critical path, denotes what's refered to as the **span** of the parallel algorithm, also written asymptotically. The larger the span, the less parallel the algorithm is.

Even if an infinite amount of operations can be done in parallel, they still need to be done. The total amount of steps required for the algorithm to complete, asymptotically, is refered to as the **work** of the algorithm. A parallel algorithm with the same asymptotic work as the most efficient sequential algorithm is said to be work-efficient.

3 Data-parallel Programming with Futhark

We will in the section look at how data-parallel programming is done in Futhark. We will look at the tools provided by the language to enable parallelization, how these can be used, as well as the limitations set by using the language.

3.1 Tools for Parallel Programming

With Futhark being a language designed for parallel programming, it contains built in functions that enable this style of programming for the user, without having to manage parallelization themselves. The main group of functions doing this are the second-order array combinators, which are operationally parallel in a way that can be exploited by the compiler [1]. Various of these functions are used throughout the implementations of the algorithms to make them run in parallel, but we will here go through 5 of the most important. All the built in second-order array combinators, alongside their cost and typing, can be found at [1]

3.1.1 Map

Map applies a given function to each element of an input array, returning an array containing the return values of the function. Map is the function used most in the implementations, as doing something to every element is a very common occurrence. Map takes a function and an array, and returns an array. The function has the typing `map 'a [n] 'x: (f: a -> x) -> (as: [n]a) -> *[n]x` and a cost of **Work:** $O(n \times W(f))$ and **Span:** $O(S(f))$.

3.1.2 Hist

Hist computes a generalized k-bin histogram by applying a given function pairwise to the elements of each bin. Hist is in the implementations used as a kind of filter, to only get one of each entry with a specific property, such as being the smallest of the entries in that bin. Hist takes a function, a default value, the number of bins, and two array, and returns an array. The function has the typing `hist 'a [n]: (op: a -> a -> a) -> (ne: a) -> (k: i64) -> (is: [n]i64) -> (as: [n]a) -> *[k]a` and a cost of **Work:** $O(k + n \times W(op))$ and **Span:** $O(n \times W(op))$ in the worst case.

3.1.3 Filter

Filter applies a given function returning either true or false to each element of an array, and returns an array containing only the entries for which the function returned true. Filter takes a function and an array, and returns an array of booleans. Filter has the typing `filter [n] 'a: (p: a -> bool) -> (as: [n]a) -> *[]a` and the cost is **Work:** $O(n \times W(p))$ and **Span:** $O(log(n) \times W(p))$.

3.1.4 Scatter

Scatter takes a target array, an array of values, and an array of destinations. Each element in the array of values is then written to the index of the target array, given by the corresponding value of the array containing the destinations. Scatter is mainly used save the results from an iteration of the algorithm running, to keep track of the progress between iterations. The function has the typing `scatter 't [k] [n]: (dest: **[k]t) -> (is: [n]i64) -> (vs: [n]t) -> *[k]t` and cost **Work:** $O(n)$ and **Span:** $O(1)$.

3.1.5 Scan

Scan applies a given associative function to a given input and the first item of an input array. The result is then fed the the function alongside the 2nd item of the list, saving each partial result. As an example, calling the function with the input parameters `scan (+) 0 [1, 2, 3, 4, 5]` would give the result [1, 3, 6, 10, 15]. Scan has the typing `scan [n] 'a: (op: a -> a -> a) -> (ne: a) -> (as: [n]a) -> *[n]a` and a cost of **Work:** $O(n \times W(op))$ and **Span:** $O(log(n) \times W(op))$.

3.2 Restrictions of Futhark

3.2.1 Immutable Arrays

Futhark arrays are considered immutable, meaning we can't make any edits to the contents of an array without making a copy of it. This makes some problem require a different solution to what would normally be done, as a lot of algorithms assume in place updating is possible. Futhark does however have something that makes it so not all data has to be copied. Functions can "consume" the array and make edits to it, with eg. scatter doing this to facilitate writing new values to the target array. The consumed array can however not be used afterwards, meaning that it does still require some working around.

3.2.2 Simple Data Structures

Futhark does have features for structuring data such as types and modules, but due a reliance on arrays to perform parallel work and now having enabling features such as pointers, complex data structures aren't used much. This means that a different way of thinking about problems is required compared to other languages that are more free with how data is structured.

3.2.3 Currently not much Tooling

While Futhark does have some some built in features such as the benchmarking tool, and some extisting tools for adding language support to different editors, it is still lacking tools found for more well established languages. While missing tools doesn't have a detremental effect on the usage of the language, it can make it less comfortable to program in.

The same can be said about the availability of external packages. This wasn't a problem for the algorithms implemented as it does have packages for standart features such as a sorting library, but it might have been a problem for more complex problems, where more well established languages might have existing packages that can help with some of the work.

4 Representation of Data for Parallel Computing

For Futhark to be able to efficiently work on the data in parallel, the data needs to be stored in some kind of array format. As such, we can't easily make use of more complex data structures, like the PBBS implementations do. This does in some instances, which will be discussed in more detail in later chapters, make some of the implementations look less clean than what the PBBS implementations can do. It also means that for the sake of efficiency, different parameters of, what would in other languages probably be represented as objects or structures, are spread over multiple different arrays, which can sometimes make it difficult to follow. It also results in some cases where some extra work is required to organize the different parameters stored in multiple arrays, such that they can be used together, and cases where extra storage is needed, such as an array containing the numbers from 0 to N, to be able to keep track of the data.

5 The Method Used for Benchmarking

We will in this section go through the process of setting up, running, and collecting the data from the benchmarks, for both the PBBS suite and the Futhark implementations. We will look at where the data used when benchmarking the algorithms come from, how the benchmarks were run, and any considerations needed when performing and comparing the results.

5.1 The Benchmarking Process

Both Futhark and the PBBS benchmark suite comes with built in tools for benchmarking programs. As these tools are very similar in the way they perform the benchmarks, both reading the data into memory before starting the benchmark and both doing multiple runs of the benchmark to ensure that a specific result wasn't an outlier, the tools were deemed comparable, and were thus used when performing the benchmarks. For running the PBBS benchmarks, a python script is provided with which you can benchmark a specific algorithm or all algorithms provided, choose if only parallel algorithms should be benchmarked, as well as which dataset the the benchmark should be run on. For running the benchmarks, the following command was used for each relevant algorithm

```
./runall -par -nonuma (-small) -only algorithm/implementation
```

With Futhark, the benchmarking tool is built into the Futhark compiler/interpreter itself. It is then only necessary to specify which futhark program should be benchmarked, as well as the backend the program should be compiled to. The dataset file as well as a file containing the expected output of the benchmark is specified inside the program itself. As such, for running the benchmarks on Futhark, the following two commands were used, when benchmarking for CUDA and multicore targets respectively.

```
futhark bench --backend=cuda algorithm.fut
futhark bench --backend=multicore algorithm.fut
```

5.2 Generating the Benchmark Data and Checking Correctness

The PBBS Benchmark Suite includes tools for generating test data for each of the different Benchmarks being run. The test data used by the Futhark implementations is the same as the data used by the PBBS benchmarks, except the data is converted to a format that can be easily read by Futhark programs. This conversion results in a smaller file size, but as the benchmark timing doesn't start until the input data has been read, this doesn't have an effect on the results.

The PBBS Benchmark Suite also includes tools to verify that the output of the program is correct. These tools have been used to check the correctness of the implementations, after conversion from the Futhark data format to the PBBS data format. It should, however, be noted that it isn't ensured that the algorithm performs correctly even if the checker accepts the output [4]. Thus, the output has also been manually verified on smaller data sets to ensure that what the algorithms are outputting isn't completely wrong.

As such, a complete check of the implementation of a Futhark algorithm is as follows

1. Generate input data using corresponding PBBS generation tool
2. Convert input data from PBBS format to Futhark format using *pbbs2fut*
3. Run Futhark on the input file
4. Convert the resulting output from Futhark format back to PBBS format using *fut2pbbs*
5. Verify the result using the input and output file in the PBBS format using the corresponding verification tool

5.3 Standartisation

For it to make sense to compare the results of the different benchmarks, it is required that the runs are performed in similar enviroments, such that the results aren't effected by external variables. As such, the benchmarks were run on compute nodes provided and maintained by University of Copenhagen, with the same SLURM config for each instance. The programs were run with 4 (virtual) CPUs, 64GB of RAM, and a NVIDIA Titan RTX graphics card. As it isn't possible to specify the exact parameters of the processor being used, and we can't see the load being put on the processor by other processes running on the physical machine, it isn't possible to get the exact same setup for each run. Different runs of the same benchmarks with the same setup have however been within few percentage points of each other, and as such, the setup is being deemed stable enough for use.

5.4 CPU vs. GPU

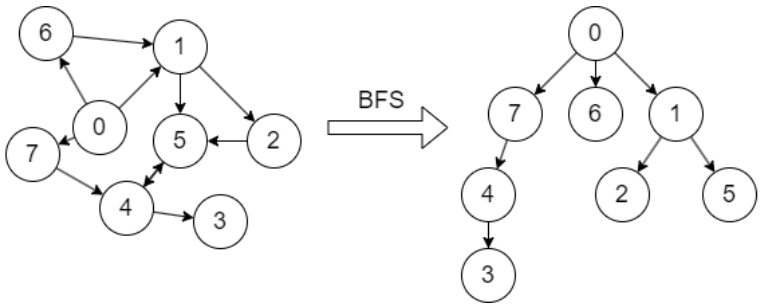
As the PBBS implementations run only on the CPU, while the Futhark compile target we will mainly be focusing on is CUDA based running on a GPU, it isn't possible to directly compare the runtimes of the benchmarks. The implementations are all heavily parallelized, and as such, adding more or faster virtual CPUs would speed up the CPU benchmarks while having little to no effect on the benchmarks run the GPU, while adding more or faster GPUs would speed up the benchmarks running on the GPU. The virtual components have, however, been chosen such that neither is greatly out of proportion to the other, but the actual performance numbers of the different results shouldn't be weighed too heavily.

6 Breadth First Search

6.1 Introduction

The explanation of the problem, as given by the PBBS benchmark suite is as follows

From	To
0	1
0	6
0	7
1	2
1	5
2	5
4	3
4	5
5	4
6	1
7	4



Node	Parent
0	0
1	0
2	1
3	4
4	7
5	1
6	0
7	0

Figure 1 – Example of BFS going from a graph defined by its edges, to a tree defined by each nodes parent

> *"Run a breadth first search on a directed graph starting at a specified source vertex and return a bfs-tree. Duplicate input edges are allowed. The input is in adjacency array format with an ordering among the outgoing edges. If the graph is not connected then only the tree over the vertices reachable from the source should be returned." [2]*

The breadth first algorithm is an algorithm used for searching through a tree or graph, generally for one or more entries that meet certain criteria, eg. finding the smallest entry of the graph or tree. The algorithm starts at a given starting node, and searches every node at the current depth level, before proceeding to the next depth level. The algorithm runs until every node of the graph or tree has been visited, or until a specified node has been found, depending on the usage of the algorithm.

For this implementation of the BFS algorithm, the edges of the graph are directed, meaning we can only traverse the edges in one direction. Also, we are to ignore nodes that aren't reachable from the starting node. The objective of this problem isn't to find nodes in the graph, but instead to build a BFS-tree, meaning we for each node in the graph need to find its parent node in a BFS-tree built from the graph. That starting node should have itself as its parent. An example can be seen in figure 1.

6.2 Implementation in Futhark and PBBS

The PBBS benchmark suite provides 3 parallel implementations of the algorithm, these being simpleBFS, backForwardBFS, and deterministicBFS. One of them is a determenistisk implementation, providing the same result each time it's run, while the two others are nondeterministic, meaning that can provide different, but still correct, outputs each time they're run, due to race conditions in the parallel parts. We will here focus on the implementation of simpleBFS, as it's the one the Futhark implementation is to some extend based on, and because the fastest implementation, this being backForwardBFS, makes use of a subset of the Ligra library, which was deemed too much to go through.

The simpleBFS implementation instead makes use of the Parlay library [6] for its parallelization. The implementation gets all edges from the "frontier", the frontier being the currently looked at nodes. As multiple nodes in the frontier might have edges pointing to the same node, one of these is chosen is chosen to be the parent of the node being pointed to. The nodes pointed to by the edges going out of the frontier is then set as the new frontier, continuing until there are no more reachable nodes in the graph. The main part of the implementation can be seen in Listing 1.

```
1 while (frontier.size() > 0) {
2
3   // get out edges of the frontier and flatten
4   auto nested_edges = parlay::map(frontier, [&] (vertexId u) {
```

```
return parlay::delayed_tabulate(G[u].degree, [&, u] (size_t i) {
    return std::pair(u, G[u].Neighbors[i]);});});
  auto edges = delayed::flatten(nested_edges);

  // keep the v from (u,v) edges that succeed in setting the parent array at v to
    u
  auto edge_f = [&] (auto u_v) {
    vertexId expected = -1;
    auto [u, v] = u_v;
    return (parent[v] == -1) && parent[v].compare_exchange_strong(expected, u);
  };
  frontier = delayed::filter_map(edges, edge_f, [] (auto x) {return x.second;});
}
```

Listing 1 – Main part of the simpleBFS implementation from [7]

The futhark implementation works in much the same way as the simpleBFS implementation, with the major difference being that it also handles the edges going to already seen vertexes, and filters those results out at the end of each loop. We first get all the vertexes reachable from the currently looked at vertexes, while returning a placeholder instead of the actual vertex, if it has already been visited. These placeholders are then filtered out of the queue, whereafter duplicate entries are filtered out. These duplicate entries are a result of multiple vertexes have edges pointing to the same vertex, but as each vertex can only have 1 parent, only one can remain. This again continues until there are no more reachable vertexes. To facilitate this implementation, a queuePair structure is used, that stores the current vertex, as well as the vertex from which the edge pointing to it originated. The main loop of the implementation can be seen in Listing 2, while the full implementation can be found in the appendix in Listing 9.

```
let (parents, _) = loop (parents, queue) while length queue > 0 do
  -- Setup a function that takes a queuePair, and returns how many vertexes goes
    out of it
  let get_edges_of_vert_fun = edges_of_vertex verts (i32.i64 nEdges)
  -- Setup a function that takes a queuePair and an int i, and returns the vertex
    pointed to by the i'th edge
  let get_ith_edge_from_vert_fun = get_ith_edge_from_vert verts edges parents

  -- Get the vertexes in the next layer
  let newQueue = expand (get_edges_of_vert_fun) (get_ith_edge_from_vert_fun) queue
  -- Remove empty placeholders ({-1, -1} queuePairs)
  let filteredQueue = filter (\q -> q.parent != -1) newQueue
  -- Remove duplicates from the queue
  let noDupesQueue = remove_duplicates nVerts filteredQueue

  in (update_parents parents noDupesQueue, noDupesQueue)
in parents
```

Listing 2 – Main part of the Futhark implementation of BFS

6.2.1 The Cost

The cost of the algorithm is dominated by how many times the main loop of the algorithm runs, as well as the cost of each run. The amount of sequential operations the algorithm goes thorough is equal to the longest path from the starting node to any node, this also being the depth of the resulting tree. In a worst case scenario, this can be equal to the amount of vertexes in the graph, if the graph is similar to a linked list, but should realistically be way lower in a normal graph. We call this length **d**. Thus, the cost of the loop is **Work:** $O(d)$ and **Span:** $O(d)$.

	RLG_J_10	RMG_J_12	3D_J
Futhark-cuda	0.241	0.291	0.964
Futhark-multicore	5.372	5.731	34.441
PBBS-simple	1.932	2.049	16.179
PBBS-backForwardBFS	0.821	0.701	16.939
PBBS-deterministicBFS	2.649	2.578	27.358

Table 1 – Breadth First Search benchmark times (seconds)

The worst case cost of the inside of the loop is **Work:** $O(k+n)$ and **Span:** $O(n)$, with n being the number of edges and k being the number of vertexes. The cost is dominated by the worst case of the hist function, however, this span only occurs in specific circumstances, which shouldn't be the case for our datasets. Thus, a more realistic cost of the function would be **Work:** $O(k+n)$ and **Span:** $O(log(n))$, with the span being dominated by eg. the filter function.

Putting this together, we get that the worst case cost of our BFS algorithm is **Work:** $O(d \times (k+n))$ and **Span:** $O(d \times n)$, but **Work:** $O(d \times (k+n))$ and **Span:** $O(d \times log(n))$ in a more realistic case.

6.3 Benchmark Results

Running the 3 PBBS implementations and two two Futhark compile targets, these being CUDA based GPU and multicore CPU, on the datasets provided by the PBBS benchmark suite, we get the times presented in Table 1.

Setting the overall best PBBS implementation as the standard, this being backForwards-BFS, we can look at the speedup of each of our implementation. We can in Figure 2 see that our multicore compile target has a speedup of approximately 0.15, 0.12, and 0.49 respectively. This is to some extend to be expected, as the Futhark compilers main focus isn't to generate multicore code, and should thus not be able to compare to optimized PBBS implementation. The CUDA target has a speedup of approximately 3.41, 2.41, and 17.58 respectively, showing a decent speedup. Especially the 3Dgrid dataset performs extremely well, where it also managed to do better than the others on the multicore target. It is not clear why the implementation performs especially well on this dataset. We can see that it has a higher ratio of edges to vertexes, making the resulting tree less deep, making the work more parallel. As Futhark is focused on heavily parallel workloads, this might explain why it performs better here than on the other datasets, but it has not been thoroughly tested.

7 Maximal Independent Set

7.1 Introduction

The explanation of the problem, as given by the PBBS benchmark suite is as follows

"Given a undirected graph return a maximal independent set for the graph" [2]

An independent set is a set of vertices in a graph, where none of the vertices in the set are connected by and edge. A maximal independent set is an independent set where there are no more edges in the graph that can be added, without it no longer being an independent set. The maximum independent set would be the largest possible maximal independent set of a given graph, but finding the maximum is a more costly algorithm. An example of a Maximal Independent Set can be seen in Figure 3.

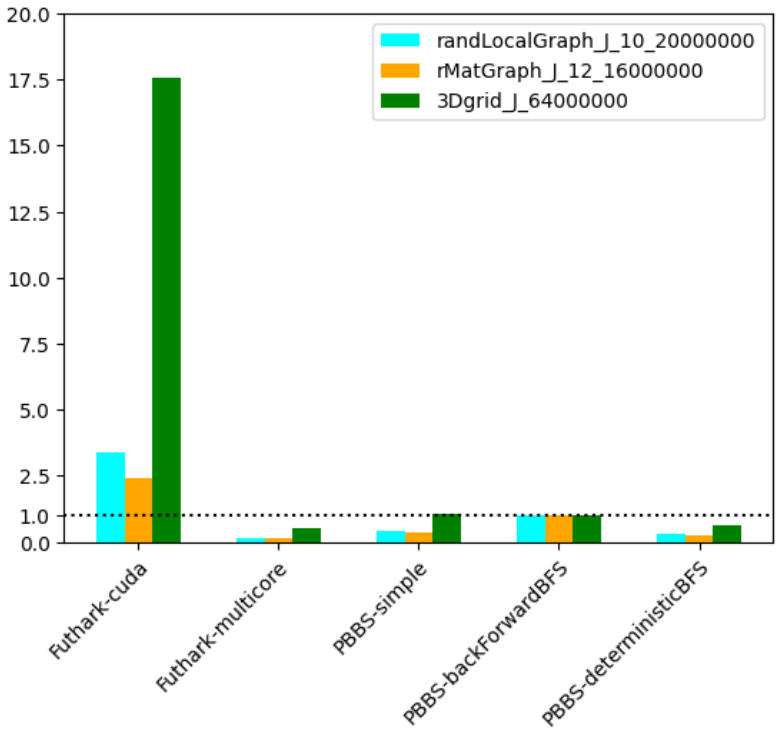


Figure 2 – The speedup of each BFS implementation compared to the best overall PBBS

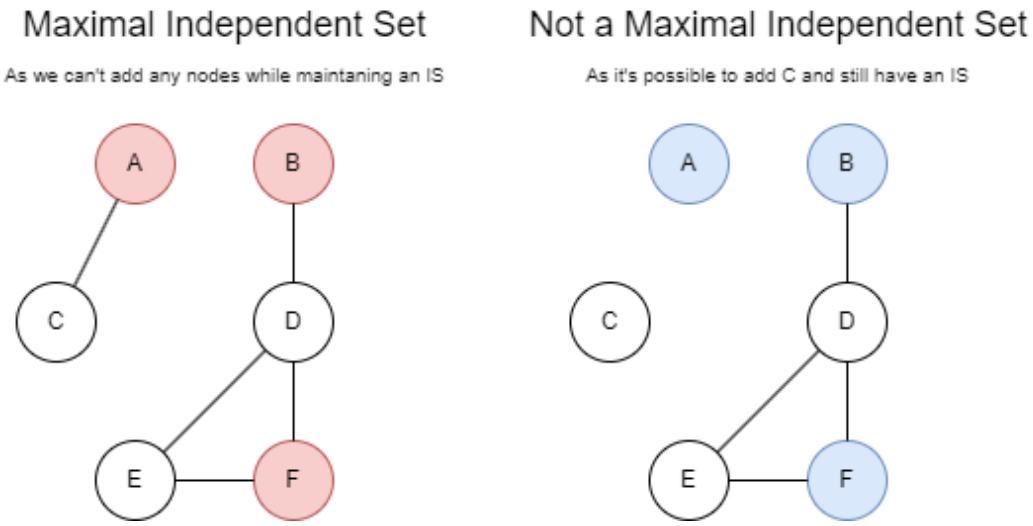


Figure 3 – Example of a Maximal Independent Set and a not Maximal Independent Set

7.2 Implementation in Futhark and PBBS

The PBBS suite provides 2 parallel implementations of the Maximal Independent Set algorithm, these being incrementalMIS and ndMIS. We will focus on the ndMIS implementation as it's one of the more clear PBBS implementation and, although not being the overall fastest implementation, was the fastest for two out of the three datasets.

The ndMIS runs for each vertex until the vertex is either in, or impossible to add the the MIS. The ndMIS implementation works by using an atomic lock system, and having each vertex as either undecided, in the MIS, or impossible to add due to a neighbour being in the MIS. Every vertex is also either locked or unlocked at all times, with locking being an atomic action. The algorithm only allowing vertexes that are unlocked, and manage to lock themselves, to try and be added to the MIS. If a vertexes manages to lock itself, it then tries to lock all of its neighbours that aren't already marked impossible to add. If it doesn't manage lock all its neighbours, we can't be certain that one of its neighbours isn't also trying to add itself, and thus it unlocks itself and all its neighbours. If it did manage to lock all neighbours, we know that we can add the vertex to the MIS, and mark its neighbours as impossible. At some point, every vertex will either be added or impossible to add, and the algorithm will have finished. Pseudocode for the main part can be found in Listing 3.

```
# Called for each vertex in parallel, with i being the vertex
parallel_for(i):
  while true:
    if (already_determined_in_or_out(i)) break;
    if (!not_self_locked(i)) continue;

    k = number_of_neighbours_not_in_MIS_or_locked(i);

    if (k = number_of_neighbours(i)):
      set_self_as_in_and_neighbours_as_out_of_MIS(i);
    else:
      unlock_all_neighbours(i);
```

Listing 3 – Pseudocode for the main part of the ndMIS implementation from [7]

The Futhark implementation is largely based on Luby's Algorithm [5]. Each vertex is assigned a unique number between 0 and the number of vertexes, being the id of the vertex. A Vertex is only added if its id is smaller than all of its neighbours, making it so no neighbouring vertexes are added in the same loop. As there is always at least one vertex with an id smaller than all others, the algorithm is bound to complete at some point. All the chosen vertexes are then added to the MIS, and their neighbours are marked as being impossible to add. This is done until there are no unmarked vertexes, after which the algorithm is complete. The main loop of the algorithm, can be found in the Listing 4 and the full implementation can be found in Listing 10.

```
let (_, I) = loop (C, I) while (i64.sum C) > 0 do
  -- Get an array of flags for which vertexes can be added to MIS
  let newI = map (can\_add vertexes edges random_state C) indexes
  -- Map the index of each 0-flag to -1, as to be ignored by scatter
  let targets = map2 (\i j -> j*i + (-1) * (1-i)) newI indexes
  -- Update our MIS with found values
  let I = scatter I targets newI

  -- For each newly added vertex, get its neighbours
  let marked = expand (edges_of_vertex_or_0 vertexes nEdges newI) (mark_neighbour
   vertexes edges) indexes
  -- Remove the vectors neighbours and self
  let C = remove_neighbour_and_self marked targets C
  in (C, I)
```

	RLG_JR_10	RMG_JR_12	3D_JR
Futhark-cuda	0.616	0.490	1.534
Futhark-multicore	7.937	6.458	18.212
PBBS-incrementalMIS	2.011	1.149	2.598
PBBS-ndMIS	1.528	2.251	2.147

Table 2 – Maximal Independent Set benchmark times (seconds)

```
14 in I |> (map i32.i64)
```

Listing 4 – Main part of the Futhark implementation of MIS

7.2.1 The Cost

The cost of the algorithm is determined by how many times the serial loop has to be run, as well as the cost of each run. It has not been possible to verify the expected amount of loops that implementation requires, but Luby's Algorithm has an expected loop count of $O(log(n))$ [5]. As our implementation is based off of Luby's, and as it is close to the amount of loops measured when testing the algorithm, we will be using this value for the calculations. Thus, the cost of the loop is **Work:** $O(log(n))$ and **Span:** $O(log(n))$.

The cost of the body of the loop is dominated by the first map, seen on line three of Listing 4, and the expand function imported from the external package "segmented". The work is dominated by the map, which calls the can_add function on each vertex, giving us $n \times W(can\_add)$. can_add checks all edges going out of the vertex, giving us a worst case of work $n \times e$. But as we are at most checking each edge once, the actual work is only the $O(e)$, with e being the amount of edges. The span comes from the expand function, which makes use of the Scan function, which in this case has **Span:** $O(log(n))$.

Putting the loop and loop body together, we get that the cost of the MIS algorithm is **Work:** $O(log(n) \times e)$ and **Span:** $O(log(n) \times log(n))$.

7.3 Benchmark Results

Running the 4 implementations on the datasets provided by PBBS, we get the times presented in Table 2. The incrementalMIS implementation is the fastest on two of the three datasets, but ndMIS is used as the standard as it is slightly faster overall.

The speedup of each implementation compared to ndMIS can be seen in Figure 4. The Futhark multicore target has a speedup of 0.253, 0.178, and 0.143 respectively, while the Futhark CUDA target has a speedup of 3.263, 2.345. and 1.694 respectively. We can again see that the multicore compile target of the Futhark implementation is quite a bit slower than the optimized PBBS implementation, and that the CUDA target is faster than the PBBS implementation.

Looking at the data, we can see that the ratio between the speedup of the different data structures is pretty similar between the CUDA and multicore target, implying that the implementation scales well with more parallel hardware.

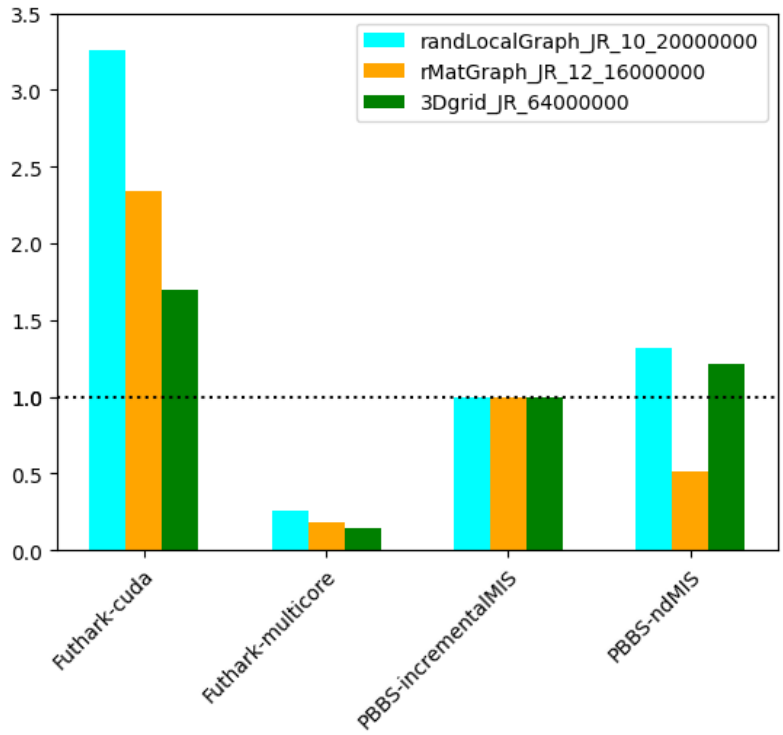


Figure 4 – The speedup of each MIS implementation compared to the best overall PBBS

8 Maximal Matching

8.1 Introduction

The explanation of the problem, as given by the PBBS benchmark suite is a follows

> *"Given a undirected graph return a maximal matching for the graph. The input graph can be in any format (as long as it does not encode the MIS somehow). Also the code cannot reorder the graph for locality. The output needs to be a sequence of integers corresponding to the positions of the edges in the input (zero based) in the MM." [2]*

A Matching, also called an Independent Edge Set, is a set of edges in a graph, where none of the edges share a vertex. A Maximal Matching is a Matching for which no more edges in the graph can be added while it remaining a matching. A Maximum Matching is the largest possible maximal matching of a graph, but as this is not required to be found by the problem description, we will not focus on it. An example of a Maximal Matching can be seen in Figure 5.

8.2 Implementation in Futhark and PBBS

The PBBS benchmark suite provide only 1 implementation for the Incremental matching algorithm. Each edge has a unique ID, being a number between 0 and the total amount of edges. The main part of the implementation consists of a sequential loop that is run until the algorithm finishes. The body of the loop is split into two parallel steps, the reserve and the commit steps.

First, in parallel, each edge tries to reserve itself with the two edges it's connected to. If neither of the vertexes connected to the edge are already matched, meaning it already has an edge that is part of the MM connected to it, it tries to write its id to the two connected vertexes. This is done using the reserve function, which is an atomic function that writes only if the new

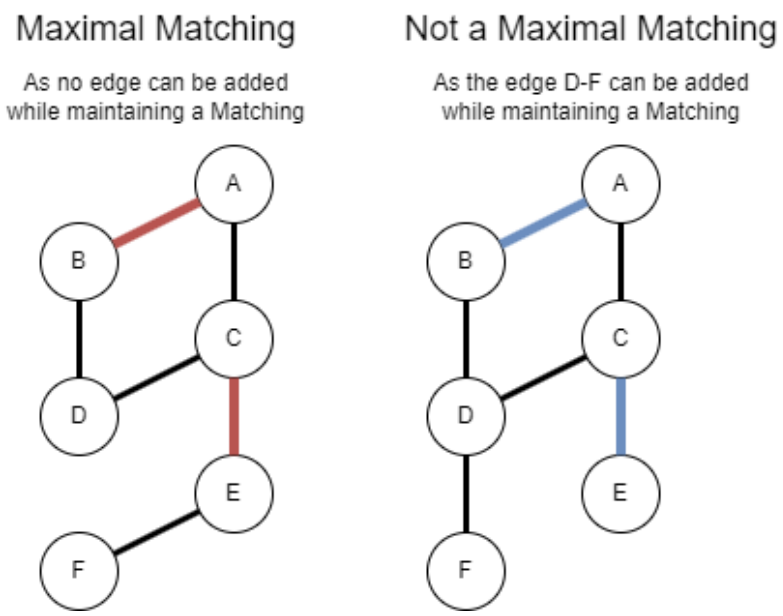


Figure 5 – Example of a Maximal Matching and a not Maximal Matching

value is smaller than the existing. After every edge has done this, each vertex is reserved by the connected edge with the smallest id.

In the second step, also in parallel, each edge tries to commit itself to the MM. This is done by each vertex checking if both of the connected vertexes has the same reserved id as itself. As each id is unique, this means that both vertexes was reserved by the edge, and it is thus safe to add it, as no other edges connected to these vertexes can add themselves as well. After each loop, the reserved vertexes are reset, and vertexes with included edges are marked. Pseudocode of the algorithm can be found in Listing 5.

```
while (!finished):
  # Called for each edge in parallel, with i being the edge
  parallel_for(i):
    u = i.vertex_1;
    v = i.vertex_2;

    if (u.matched || v.matched || (u == v)):
    else:
      u.reserve(i);
      v.reserve(i);

  parallel_for(i):
    u = i.vertex_1;
    v = i.vertex_2;

    if (v.reserved_equals(i)):
      v.reset();
      if (u.reserved_equals(i)):
        u.match = v.match = true;
    else if (u.reserved_equals(i)):
      u.reset();
```

Listing 5 – Pseudocode for the main part of the incrementalMatching implementation from [7]

The Futhark implementation is based on the incrementalMatching implementation, and thus works in much the same way. We keep an array of edges that have not yet been added or become impossible to add, and run the main loop until this is empty. Instead of using an atomic reserve function to get the smallest id for each vertex, two vertex-id pairs are created for each

edge, and the hist function is used to only keep the smallest id for each vertex. Just like in the PBBS implementation, each edge checks if both connected vertexes has its id, and if they do, adds itself to the MM and marks the vertexes. The edges that are now impossible to add to the MM are removed from the edges array, meaning less edges have to be gone through next loop, and the reserved ids are reset. As there each loop is at least one edge added to the MM, the algorithm is sure to terminate. The code for the main loop of the algorithm can be seen in Listing 6 and the full implementation can be found in Listing 11.

```
let (_, _, _, _, includedEdges) = loop (edges, edge2Ids, markedVerts,
    smallestEdgeId, includedEdges) while (length edges > 0) do
  let (smallestTargets, smallestValues) = getSmallestPairs edges edge2Ids nVerts
    nEdges

  let smallestEdgeId = scatter smallestEdgeId (map (i64.i32) smallestTargets)
    smallestValues

  let (markedVerts, includedEdges) = update edges edge2Ids smallestEdgeId
    markedVerts includedEdges

  let (edges, edge2Ids) = removeMarked markedVerts edges edge2Ids

  let smallestEdgeId = resetsmallestEdgeId smallestEdgeId
  -- I don't get why this copy is needed. I feel like it shouldn't be
  in (edges, edge2Ids, copy markedVerts, smallestEdgeId, includedEdges)
in filter (.1) (zip edgeIds includedEdges) |> map (.0)
```

Listing 6 – Main part of the Futhark implementation of MM

8.2.1 The Cost

The cost is once again determined by a loop and the body of said loop. The worst case for the loop is that we have to go through it once for each edge, which would happen given a graph shaped like a linked list, with the ids of the edges going in ascending order. This is however not a realistic structure for a graph, and as such, a better bound should be possible. It has not been possible to calculate a limit ourself, but an algorithm using the same overall method, just recursively, shows that the amount of remaining edges halves with each recursive call [8]. With this being the case, the amount of loops should be in the magnitude of $O(log(e))$, with e being the number of edges. As this is in the same order of magnitude as the measured number of loops, and as the overall method is the same, we will assume that this is the same for our algorithm.

The cost of the loop body is once again dominated by the hist function with its worst case span of $O(n)$, but just like last time, the way we use the function should not result in the worst case span for the function. As such, the more realistic cost of the span is given by eg. the filter function, with a span of $O(log(n))$.

This gives, with e being the number of edges and v being the number of vertices in the input graph, a worst case cost of **Work:** $O(log(e) \times (v + e))$ and **Span:** $O(log(e) \times e)$ and a more realistic cost of **Work:** $O(log(e) \times (k + n))$ and **Span:** $O(log(e) \times log(e))$.

8.3 Benchmark Results

Running the 3 implementations on the datasets given by PBBS, we get the times presented in Table 3.

Looking at the speedup in Graph 6, we can see that the multicore target has a speedup of 0.320, 0.359, and 0.484 respectively, while the speedup of the CUDA target is 3.331, 3.188,

	RLG_E_10	RMG_E_10	2D_E
Futhark-cuda	0.970	1.244	0.994
Futhark-multicore	10.088	11.056	7.887
PBBS-incrementalMatching	3.231	3.964	3.819

Table 3 – Maximal Matching benchmark times (seconds)

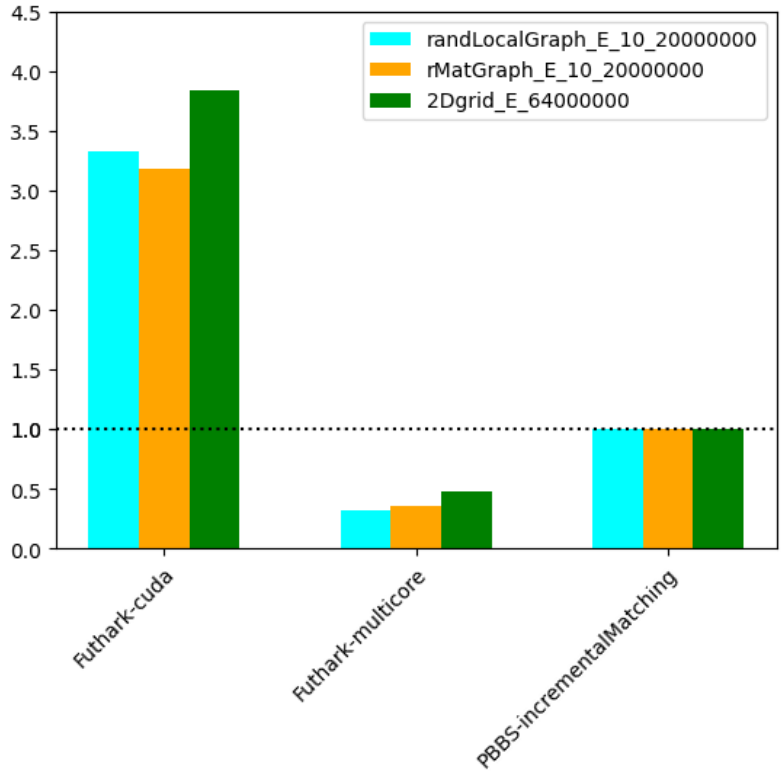


Figure 6 – The speedup of each MM implementation compared to the PBBS implementation

and 3.844. The same pattern emerges here as with the other implementations, where we can see that the multicore target performs slightly worse, the CUDA target performs slightly better, and the ratio is more or less the same between all the CUDA and multicore benchmarks. The speedup for each dataset was for this algorithm pretty close to each other, compared to the two others where one dataset did quite a bit better than the others. It is not clear why this is the case for specifically this algorithm.

9 Minimum Spanning Forest

9.1 Introduction

The explanation of the problem, as given by the PBBS benchmark suite, is as follows

> *"Given a weighted undirected graph return the minimum spanning tree (MST), or minimum spanning forest (MSF) if the graph is not connected. . . . In the case that there are multiple possible MSFs (due to equal weights), any MSF is valid." [3]*

A spanning tree is a collection of edges of a graph that connect all the vertices of the graph. As the graph is weighted, each edge has a cost associated with it. A minimum spanning tree is the set of edges that connect all vertexes, while having the smallest sum of edge costs. If

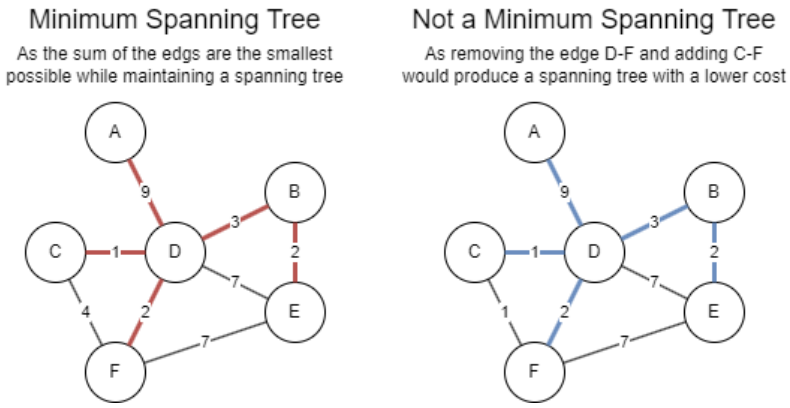


Figure 7 – Example of a Minimum Spanning Tree and a not Minimum Spanning Tree

the graph is not connected, it is instead possible to create a spanning tree for each group of vertexes, resulting in a spanning forest instead. The weights can be negative, and as such, cycles are not allowed in a minimum spanning tree, but as the wights given by the datasets are all positive, this will not be an issue. As there might be multiple different spanning forests with the same cost, eg. if all edges have the same cost, any of the possible MSFs are valid. An example can be seen in Figure 7.

9.2 The Disjoint-Set Data Structure

The disjoint-set data structure, also called the union-find data structure, is a data structure used in Kruskal's algorithm. The usage of the data-structure is to efficiently find out if two elements are part of the same set. The data structure then has two main functions, these being find and link (also called Union or Merge). Link takes an element from two different sets, and merges the two sets into one, while find takes a single element and finds which set it's part of.

9.3 Implementation in Futhark and PBBS

The PBBS benchmark suite provides 2 implementation for the MSF algorithm, with parallelFilterKruskal being the overall best performing. We will, however, focus on the parallelKruskal algorithm as it is what the Futhark implementation is based on.

The implementation starts by sorting every edge based on it's cost. The main part of the algorithm takes place in a sequential for loop that runs until the algorithm is finished. The loop, just like the Maximal matching algorithm, has two main parallel sections. The algorithm also uses the UnionFind data structure to keep track of the different subtrees that form when the algorithm is running.

In the first section, in parallel, each edge gets the sets that the two connected vertices are part of, while at the same time updating the vertex id's to be that of the set they are in, making the next loop faster. If the two vertexes are not part of the same set, it means that it could be valid to add the edge, and they are both reserved. Reserving is the same as from the previous algorithm, where the value is atomically written only if the new value is smaller.

In the second section, again in parallel, each edge again gets the set of each vertex. It is checked if first the 1st of the vertexes reserved id matched that of the edge. If it does, the sets of the two vertexes are merged, and the edge is added to the MSF. If it isn't the case for the 1st vertex, the 2nd is then checked, and teh same is done if the value is reserved there. Pseudocode of the main part of the implementation can be found in Listing 7.

```
1 edges = sort_by_cost(edges)
```

```
2
3  while (!finished):
4    # Called for each vertex in parallel, with i being the edge
5    parallel_for(i):
6      u = i.vertex1 = unionFind.find(i.vertex1);
7      v = i.vertex2 = unionFind.find(i.vertex2);
8
9      if (u != v):
10       v.reserve(i);
11       u.reserve(i);
12
13   parallel_for(i):
14     u = i.vertex1;
15     v = i.vertex2 ;
16
17     if (v.reserved_equals(i)):
18       u.reset_if_equals(i);
19       unionFind.link(v, u);
20       add_to_MSF(i);
21     else if (u.reserved_equals(i)):
22       unionFind.link(u, v);
23       add_to_MSF(i);
```

Listing 7 – Pseudocode for the main part of the parallelKruskal implementation from [7]

The Futhark implementation is based on the parallelKruskal implementation, and thus works in much the same way. Before the main loop, all the edges are sorted based on their weight, and then given an id from 0 to the number of edges. The id is given after sorting, such that the smallest weight edges has the lowest id.

In the loop, at first each edge where both vertexes are part of the same set are filtered out. Then, for each vertex, the lowest id connected edge is selected and saved. The edges are then added to the MSF, as well as the Union-Find data structure being updated by linking the vertexes from all the newly added edges. At the same time, one round of optimization is done on the Union-Find structure. This is continued until the algorithm is finished. The main loop of the algorithm can be found in Listing 8 while the complete implementation can be found in Listing 12.

```
1  let (_, _, _, _, includedEdges) = loop (UFparents, edges, edgeIds, smallestEdgeId,
       includedEdges) while (length edges > 0) do
2    let (edgeIds, edges) = map (find2 UFparents) edges |> zip edgeIds |> filter (\e
       -> e.1.0 != e.1.1) |> unzip
3
4    let (smallestTargets, smallestValues) = getLowestIndexes edges edgeIds nVerts
       nEdges
5    let smallestEdgeId = scatter smallestEdgeId smallestTargets smallestValues
6
7    let (includedEdges, UFparents) = update UFparents edges edgeIds smallestEdgeId
       includedEdges
8    in (UFparents, edges, edgeIds, smallestEdgeId, includedEdges)
9  in filter (.1) (zip edgeIndexes includedEdges) |> map (.0)
```

Listing 8 – Main part of the Futhark implementation of MSF

9.4 The Cost

The cost of the algorithm is dominated by initially having to sort the algorithm, and the use of the hist function inside the loop. The loop could in the worst case again have to run in the order of $O(e)$ times, but is expected to run in the order of $O(log(e))$. The hist function should, given

	RLG_WE_10_small	RMG_WE_12_small	3D_WE_small
Futhark-cuda	1.173	1.257	1.310
Futhark-multicore	20.435	21.097	19.685
PBBS-parallelFilterKruskal	1.254	1.856	2.639
PBBS-parallelKruskal	1.884	2.999	3.244

Table 4 – Min Spanning Forest benchmark times (seconds)

our graph inputs, not have the worst case span, and as such, the span of the loop body would be $O(log(e))$ from eg. the filter function. This would give a total span of $O(log(e) \times log(e))$, which is equal to that of the sorting function.

As such, the worst case cost of the implementation is **Work:** $O(log(e) \times (e + v))$ and **Span:** $O(e \times log(e))$, with a realistic cost of **Work:** $O(log(e) \times (e + v))$ and **Span:** $O(log(e) \times log(e))$.

9.5 Benchmark Results

Running the 4 implementations on the datasets given by PBBS, we get the times seen in Table 4. The datasets used for these benchmarks were the smaller versions provided by PBBS, rather than the large version used for the other benchmarks. This is because the memory usage of the CUDA target exceeded the 24GB of available memory of the RTX Tian graphics card used for the benchmarks.

The parallelFilterKruskal implementation is the overall fastest of the two PBBS implementations, and will thus be used as the standart for comparison. Looking at the speedup in Graph 8, we can see that the multicore target has a speedup of 0.069, 0.088, and 0.134 respectively, while the CUDA target had a speedup of 1.069, 1.476, and 2.015. The multicore target performed significantly worse than for the other benchmarks, with the RLG_WE_10 dataset being more than 16x slower than running on the PBBS implementation. The CUDA target also performed worse than the others, with the worst being only just faster, and the best being only a 2x speedup.

Part of this might come from the, compared to the other benchmarks, significantly smaller datasets not being as parallel, and thus not performing well on the implementation optimized for heavy parallelization. However, a more significant problem is probably the implementation of the unionFind data structure, which will be explained in more detail in Section 10.2.

10 Discussion

10.1 Programming in Futhark

The overall experience of implementing the different graph algorithms in Futhark has been pretty good. No major hacks have been required to get the algorithms to work, even with the limitations of the language. Having to to keep the data spread over multiple arrays have sometimes been an issue for the readability and overall structure of the program, but not to an extend where major changes to the structure of the implementation were necessary. The use of the histogram function to filter out duplicates was not initially obvious, but after it was pointed out as a potential usage of the function, much of the other problems were fairly easy to solve.

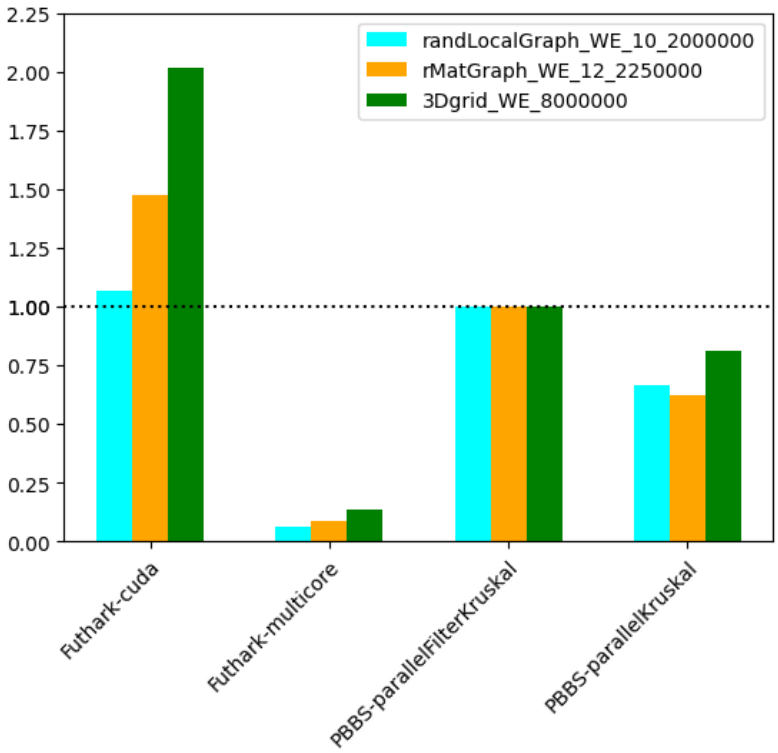


Figure 8 – The speedup of each MSF implementation compared to the best overall PBBS

10.2 The Fit of the Algorithms

Even though the problems are all highly irregular, they all had obvious points where the parallel work could be done, and where sequential work was needed. The only place where this isn't the case was the UnionFind data structure of the Minimum Spanning Forest algorithm. For the Union Find data structure to be efficiently implemented, the find will retroactively update the points to the head of the set when it's searching, making the following find operations a lot faster. Due to the immutability of the language, doing this was not possible as would require writing to the array in a sequential loop. Instead of this, a solution was found that does a single step of optimization for all elements of the set where applicable. While not enough tests have been done to ensure that this is what results in the relativly bad performance of the algorithm compared to the other futhark implementations, it most likely does have an impact.

10.3 The Overall Performance and Improvements

While all Futhark CUDA targets did see an improvement compared to the PBBS implementations, the speedup isn't what could be expected of a program running on a powerful GPU. The Futhark implementations previously showcased can definetly be improved to run faster while keeping the same overall design choices, but to make them a lot faster, a different approach is probably needed. This could eg. be algorithmically tuning the algorithm to make it better suit the dataset provided, or swapping to a different algorithm depending on the density of the graph.

11 Conclusion

We have managed to implement 4 different graph algorithms in Futhark, with no major hacks being needed to make the algorithms fit Futharks limitations. Although there has been some minor issues when implementing certain parts, the overall experience of using Futhark has been pretty good. While having managed to see a consistent speedup when comparing the Futhark CUDA programs to the multicore PBBS programs, the speedup is not what is to be expected when comparing these types of programs run on a CPU and GPU. Thus, while it is possible to implement relatively fast Graph algorithms in Futhark using the techniques used in the 4 implementations, a different method is probably needed to get really fast algorithms.

References

[1] */prelude/soacs*. URL: https://futhark-lang.org/docs/prelude/doc/prelude/soacs.html (visited on 05/31/2023).

[2] cmuparlay. *Breadth First Search (BFS)*. URL: https://cmuparlay.github.io/pbbsbench/benchmarks/breadthFirstSearch.html (visited on 01/07/2023).

[3] cmuparlay. *Minimum Spanning Forest (MSF)*. URL: https://cmuparlay.github.io/pbbsbench/benchmarks/minSpanningForest.html (visited on 01/07/2023).

[4] cmuparlay. *The PBBS Benchmark Suite (V2)*. URL: https://cmuparlay.github.io/pbbsbench/ (visited on 05/28/2023).

[5] Michael Luby. "A Simple Parallel Algorithm for the Maximal Independent Set Problem". In: *SIAM Journal on Computing* 15 (Nov. 1986), pp. 1036–1053. DOI: 10.1145/22145.22146.

[6] *ParlayLib - A Toolkit for Programming Parallel Algorithms on Shared-Memory Multicore Machines*. URL: https://github.com/cmuparlay/parlaylib (visited on 06/07/2023).

[7] *pbbsbench*. URL: https://github.com/cmuparlay/pbbsbench (visited on 05/02/2023).

[8] Julian Shun. *Notes on Simple Analysis for Parallel Maximal Independent Set and Maximal Matching Algorithms*. URL: https://people.csail.mit.edu/jshun/simple-analysis.pdf (visited on 05/10/2023).

[9] *Why Futhark?* URL: https://futhark-lang.org/ (visited on 06/11/2023).

A Complete Futhark Implementation

```
import "lib/github.com/diku-dk/segmented/segmented"

type queuePair = {vertex: i32, parent: i32}

def remove_duplicates [nQueue]  (nVerts) (queue: [nQueue]queuePair): []queuePair =
    let verts = map (\q -> i64.i32 q.vertex) queue
    let indexes = iota nQueue
    let H = hist i64.min nQueue nVerts verts indexes
    in map2 (\i j -> H[i] == j) verts indexes
        |> zip queue
        |> filter (.1)
        |> map (.0)

-- Set the parent of each vertex in the queue. The queue must not contain duplicates
def update_parents [nVerts] (parents: *[nVerts]i32) (queue: []queuePair): *[nVerts]i32 =
    let qVerts = map (\q -> i64.i32 q.vertex) queue
    let qParents = map (\q -> q.parent) queue
    in scatter parents qVerts qParents

def edges_of_vertex (verts: []i32) (nEdges: i32) (edge: queuePair): i64 =
    let extended = verts ++ [nEdges]
    in i64.i32 (extended[edge.vertex + 1] - extended[edge.vertex])

def get_ith_edge_from_vert (verts: []i32) (edges: []i32) (parents: []i32) (q: queuePair) (i: i64) : queuePair =
    -- Get the i'th vertex of the edge
    let currentVert = edges[i64.i32 verts[q.vertex] + i]
    -- If it's an unseen vertex, add it to the queue with the current vertex being the parent
    -- Else return a placeholder that we filter later
    in if (parents[currentVert] == -1)
    then {vertex = currentVert, parent = q.vertex}
    else {vertex = -1, parent = -1}

def BFS [nVerts] [nEdges] (verts: [nVerts]i32) (edges: [nEdges]i32) (parents: *[nVerts]i32) (queue: *[]queuePair): [nVerts]i32 =
    -- Loop until we get an empty queue
    let (parents, _) = loop (parents, queue) while length queue > 0 do
        -- Setup a function that takes a queuePair, and returns how many vertexes goes out of it
        let get_edges_of_vert_fun = edges_of_vertex verts (i32.i64 nEdges)
        -- Setup a function that takes a queuePair and an int i, and returns the vertex pointed to by the i'th edge
        let get_ith_edge_from_vert_fun = get_ith_edge_from_vert verts edges parents

        -- Get the vertexes in the next layer
        let newQueue = expand (get_edges_of_vert_fun) (get_ith_edge_from_vert_fun) queue
        -- Remove empty placeholders ({-1, -1} queuePairs)
        let filteredQueue = filter (\q -> q.parent != -1) newQueue
        -- Remove duplicates from the queue
        let noDupesQueue = remove_duplicates nVerts filteredQueue

        in (update_parents parents noDupesQueue, noDupesQueue)
    in parents
```

```

def main [nVerts] [nEdges] (vertexes_enc: [nVerts]i32) (edges_enc: [nEdges]i32) =
    let start = 0

    let parents = replicate nVerts (-1)
    let queue = [{vertex = start, parent = start}]
    let parents = update_parents parents queue

    in BFS vertexes_enc edges_enc parents queue

-- ==
-- input @ data/randLocalGraph_J_10_20000000.in
-- output @ data/randLocalGraph_J_10_20000000.out
-- mem_16gb input @ data/rMatGraph_J_12_16000000.in
-- output @ data/rMatGraph_J_12_16000000.out
-- input @ data/3Dgrid_J_64000000.in
-- output @ data/3Dgrid_J_64000000.out
```

Listing 9 – Futhark Implementation of Breadth First Search

```
import "lib/github.com/diku-dk/cpprandom/random"
import "lib/github.com/diku-dk/cpprandom/shuffle"
import "lib/github.com/diku-dk/segmented/segmented"

module shuffle = mk_shuffle minstd_rand

def valid_neighbour (random_state: []i64) (C: []i64) (state: i64) (neighbour: i32)
    : i32 =
    if (C[neighbour] == 1) && (random_state[neighbour] < state) then
        1
    else
        0

def edges_of_vertex (verts: []i32) (nEdges: i64) (vert: i64): i64 =
    let extended = verts ++ [(i32.i64 nEdges)]
    in i64.i32 (extended[vert + 1] - extended[vert])

def edges_of_vertex_or_0 (verts: []i32) (nEdges: i64) (newI: []i64) (vert: i64):
    i64 =
    if (newI[vert] == 0) then
        0
    else
        edges_of_vertex verts nEdges vert

def can_add [nVerts] [nEdges] (vertexes: [nVerts]i32) (edges: [nEdges]i32) (
    random_state: [nVerts]i64) (C: [nVerts]i64) (index: i64): i64 =
    if (C[index] == 0) then
        0
    else
        let vEntry = (i64.i32 vertexes[index])
        let currentEdges = edges[vEntry:vEntry + (edges_of_vertex vertexes nEdges
    index)]

        let arr = map (valid_neighbour random_state C random_state[index])
    currentEdges

        let valid = i32.sum arr
        in if (valid == 0) then
            1
        else
            0
```

```
37
38 def mark_neighbour (vertexes: []i32) (edges: []i32) (index: i64) (i: i64): i64 =
39     let edgeStartIndex = vertexes[index]
40     in (i64.i32 edges[(i64.i32 edgeStartIndex) + i])
41
42 def remove_neighbour_and_self [nVerts] (marked: []i64) (targets: []i64) (C: *[
    nVerts]i64): *[nVerts]i64 =
43     -- Needed to write 0 into array. Please add scatterC, taking a constant
    instead of an array
44     let zeros1 = map (\_ -> 0) marked
45     let zeros2 = map (\_ -> 0) targets
46     let C = scatter C targets zeros2
47     in scatter C marked zeros1
48
49 -- Can probably be done without mapping over every vertex each loop, by keeping
    track of a queue-like array
50 let MIS [nVerts] (vertexes: [nVerts]i32) (edges: []i32) (random_state: [nVerts]i64
    ) (C: *[nVerts]i64) (I: *[nVerts]i64) (indexes: []i64) (nEdges: i64) =
51     -- Loop until every vertex is added to or excluded from the MIS
52     let (_, I) = loop (C, I) while (i64.sum C) > 0 do
53         -- Get an array of flags for which vertexes can be added to MIS
54         let newI = map (can_add vertexes edges random_state C) indexes
55         -- Map the index of each 0-flag to -1, as to be ignored by scatter
56         let targets = map2 (\i j -> j*i + (-1) * (1-i)) newI indexes
57         -- Update our MIS with found values
58         let I = scatter I targets newI
59
60         -- For each newly added vertex, get its neighbours
61         let marked = expand (edges_of_vertex_or_0 vertexes nEdges newI) (
    mark_neighbour vertexes edges) indexes
62         -- Remove the vectors neighbours and self
63         let C = remove_neighbour_and_self marked targets C
64         in (C, I)
65     in I |> (map i32.i64)
66
67 def main [nVerts] [nEdges] (vertexes_enc: [nVerts]i32) (edges_enc: [nEdges]i32) =
68     let indexes = iota nVerts
69
70     -- Random seed, could be anything
71     let rng = minstd_rand.rng_from_seed [5, 3, 1, 8, 0, 0, 8]
72     -- Shuffle the indexes, giving each vertex a unique random number
73     let (_, random_state) = shuffle.shuffle rng indexes
74
75     -- Vertexes no longer needed to be checked
76     let C = replicate nVerts 1
77     -- Vertexes part of the MIS
78     let I = replicate nVerts 0
79
80     in MIS vertexes_enc edges_enc random_state C I indexes nEdges
81
82 -- ==
83 -- input @ data/randLocalGraph_JR_10_20000000.in
84 -- output @ data/randLocalGraph_JR_10_20000000.out
85 -- input @ data/rMatGraph_JR_12_16000000.in
86 -- output @ data/rMatGraph_JR_12_16000000.out
87 -- input @ data/3Dgrid_JR_64000000.in
88 -- output @ data/3Dgrid_JR_64000000.out
```

Listing 10 – Futhark Implementation of Maximal Independent Set

```
1 -- Return the edge-id pairs with the smallest edge id
```

```
def getSmallestPairs [arraySize] (edges: [arraySize][2]i32) (edge2Ids: [arraySize][2]i32) (nVerts: i64) (nEdges: i64): ([]i32, []i32) =
    -- The length of the flattened arrays
    let arraySizeFlat = arraySize * 2

    let flatE = flatten edges :> [arraySizeFlat]i32
    let flatE2i = flatten edge2Ids :> [arraySizeFlat]i32

    let zippedArray = zip flatE flatE2i

    let verts = map i64.i32 flatE

    let H = hist i32.min (i32.i64 nEdges) nVerts verts flatE2i
    in filter (\i -> H[i.0] == i.1) zippedArray
        |> unzip

-- Return the edge if it's ID is the smallest, else return placeholder
def getMMEdges (smallestEdgeId: []i32) (e: [2]i32) (i: [2]i32): ([2]i32, [2]i32) =
    if smallestEdgeId[e[0]] == i[0] && smallestEdgeId[e[1]] == i[0] then (e, i) else ([-1, -1], [-1, -1])

-- Update the marked vertexes and included edges
def update [arraySize] (edges: [arraySize][2]i32) (edge2Ids: [arraySize][2]i32) (smallestEdgeId: []i32)
                       (markedVerts: *[]bool) (includedEdges: *[]bool): (*[]bool, *[]bool) =
    -- The length of the flattened arrays
    let arraySizeFlat = arraySize*2

    let (e, e2i) = unzip (map2 (getMMEdges smallestEdgeId) edges edge2Ids)
    let flatE = flatten e :> [arraySizeFlat]i32
    let flatEi2 = flatten e2i :> [arraySizeFlat]i32

    let trues = replicate arraySizeFlat true

    let markedVerts = scatter markedVerts (map i64.i32 (flatE)) trues
    let includedEdges = scatter includedEdges (map i64.i32 (flatEi2)) trues
    in (markedVerts, includedEdges)

-- Remove the marked edges
def removeMarked [arraySize] (markedVerts: []bool) (edges: [arraySize][2]i32) (edge2Ids: [arraySize][2]i32): ([][2]i32, [][2]i32) =
    zip edges edge2Ids
        |> filter (\(v, _) -> !(markedVerts[v[0]] || markedVerts[v[1]]))
        |> unzip

-- Reset the smallest id of each vertex
def resetsmallestEdgeId (smallestEdgeId: []i32): *[]i32 =
    map (\_ -> i32.highest) smallestEdgeId

def MM [nVerts] [nEdges] (edges: [][2]i32) (edgeIds: [nEdges]i64) (edge2Ids: [][2]i32) (markedVerts: *[nVerts]bool)
                         (smallestEdgeId: *[nVerts]i32) (includedEdges: *[nEdges]bool) =
    let (_, _, _, _, includedEdges) = loop (edges, edge2Ids, markedVerts, smallestEdgeId, includedEdges) while (length edges > 0) do
        let (smallestTargets, smallestValues) = getSmallestPairs edges edge2Ids nVerts nEdges

        let smallestEdgeId = scatter smallestEdgeId (map (i64.i32) smallestTargets
```

```
    ) smallestValues

        let (markedVerts, includedEdges) = update edges edge2Ids smallestEdgeId markedVerts includedEdges

        let (edges, edge2Ids) = removeMarked markedVerts edges edge2Ids

        let smallestEdgeId = resetsmallestEdgeId smallestEdgeId
        -- I don't get why this copy is needed. I feel like it shouldn't be
        in (edges, edge2Ids, copy markedVerts, smallestEdgeId, includedEdges)
    in filter (.1) (zip edgeIds includedEdges) |> map (.0)

def main [nEdges] (edges_enc: *[nEdges][2]i32) =
    let nVerts = flatten edges_enc |> i32.maximum |> (+1) |> i64.i32

    let edgeIds = iota nEdges
    -- Create a doubled iota to simplify scatter/map on flattened edges
    let edge2Ids = map (\i -> [i32.i64 i, i32.i64 i]) edgeIds :> [nEdges][2]i32

    let markedVerts = replicate nVerts false
    let smallestEdgeId = replicate nVerts i32.highest

    let includedEdges = replicate nEdges false

    in MM edges_enc edgeIds edge2Ids markedVerts smallestEdgeId includedEdges

-- ==
-- mem_16gb input @ data/randLocalGraph_E_10_20000000.in
-- output @ data/randLocalGraph_E_10_20000000.out
-- mem_16gb input @ data/rMatGraph_E_10_20000000.in
-- output @ data/rMatGraph_E_10_20000000.out
-- input @ data/2Dgrid_E_64000000.in
-- output @ data/2Dgrid_E_64000000.out
```

Listing 11 – Futhark Implementation of Maximal Matching

```
import "lib/github.com/diku-dk/sorts/merge_sort"

def is_root (UFparents: []i32) (vert: i64): bool =
    UFparents[vert] < 0

def is_root32 (UFparents: []i32) (vert: i32): bool =
    UFparents[vert] < 0

def find (UFparents: []i32) (vert: i32): i32 =
    if (is_root32 UFparents vert)
        then vert
    else
        -- Due to agressive flattening, this should never actually have to loop
        loop p = UFparents[vert] while !(is_root32 UFparents p) do
            UFparents[p]

def find2 (UFparents: []i32) (edge: (i32, i32)) : (i32, i32) =
    ((find UFparents edge.0), (find UFparents edge.1))

-- Do 1 step of flattening
def update_once [pLength] (UFparents: [pLength]i32) : (*[pLength]i32) =
    let indexes = iota pLength
    -- If neither the index or its parent are roots, set the parent as the parents parent.
    in map (\v -> if (is_root UFparents v) || (is_root32 UFparents UFparents[v])
```

```
   then UFparents[v] else UFparents[UFparents[v]]) indexes

-- This is safe as each element of us is unique
def link (UFparents: *[]i32) (us: []i32) (vs: []i32) =
    scatter UFparents (map i64.i32 us) vs

-- Order by weight, and if equal, by index
def ltm (a: (f64, (i32, i32), i64)) (b: (f64, (i32, i32), i64)): bool =
    (a.0 < b.0) || (a.0 == b.0 && (a.2 < b.2))

-- For each vertex, get the edge with the lowest index
def getLowestIndexes [arraySize] (edges: [arraySize](i32, i32)) (edgeIds: [
   arraySize]i64) (nVerts: i64) (nEdges: i64) =
    -- The length of the flattened arrays
    let arraySizeFlat = arraySize * 2

    let flatE = map (\e -> [e.0, e.1]) edges |> flatten |> map i64.i32 :> [
   arraySizeFlat]i64
    let flatE2i = map (\i -> [i, i]) edgeIds |> flatten :> [arraySizeFlat]i64

    let zippedArray = zip flatE flatE2i

    let H = hist i64.min nEdges nVerts flatE flatE2i
    in map (\i -> if H[i.0] == i.1 then i else (-1,0)) zippedArray
        |> unzip

-- If the edgis the one with the smallest index, return them for linking. Else
   return placeholder
def getMSFEdges (smallestEdgeId: []i64) (e: (i32, i32)) (i: i64) : ((i32, i32),
   i64) =
    if (smallestEdgeId[e.1] == i) then
        ((e.1, e.0), i)
    else if (smallestEdgeId[e.0] == i) then
        ((e.0, e.1), i)
    else
        ((-1, -1), -1)

-- Update the UF array and the array of included edges
def update[arraySize] (UFparents: *[]i32) (edges: [arraySize](i32, i32)) (edgeIds:
    [arraySize]i64)
                      (smallestEdgeId: []i64) (includedEdges: *[]bool) =
    let (UVs, IDS) = unzip (map2 (getMSFEdges smallestEdgeId) edges edgeIds)
    let (us, vs) = unzip UVs

    let UFparents = link UFparents us vs |> update_once
    let includedEdges = scatter includedEdges IDS (replicate arraySize true)
    in (includedEdges, UFparents)

def MSF [nVerts] [nEdges] (UFparents: *[]i32) (edges: [](i32, i32)) (edgeIds: *[]
   i64) (edgeIndexes: []i64)
                          (smallestEdgeId: *[nVerts]i64) (includedEdges: *[nEdges]
   bool) =
    let (_, _, _, _, includedEdges) = loop (UFparents, edges, edgeIds,
   smallestEdgeId, includedEdges) while (length edges > 0) do
        let (edgeIds, edges) = map (find2 UFparents) edges |> zip edgeIds |>
   filter (\e -> e.1.0 != e.1.1) |> unzip

        let (smallestTargets, smallestValues) = getLowestIndexes edges edgeIds
   nVerts nEdges
        let smallestEdgeId = scatter smallestEdgeId smallestTargets smallestValues
```

```
74
75          let (includedEdges, UFparents) = update UFparents edges edgeIds
        smallestEdgeId includedEdges
76          in (UFparents, edges, edgeIds, smallestEdgeId, includedEdges)
77      in filter (.1) (zip edgeIndexes includedEdges) |> map (.0)
78
79  def main [nEdges] (edges_enc: [nEdges][2]i32) (weights: [nEdges]f64) =
80      let nVerts = flatten edges_enc |> i32.maximum |> (+1) |> i64.i32
81
82      let edges = map (\e -> (e[0], e[1])) edges_enc
83
84      -- The innitial index of each vertex
85      let edgeIndexes = iota nEdges
86      -- The id of each vertex (Its index after sorting)
87      let edgeIds = iota nEdges
88
89      let (_, edges, edgeIndexes) = zip3 weights edges edgeIndexes |> merge_sort ltm
         |> unzip3
90
91      let smallestEdgeId = replicate nVerts i64.highest
92      let included = replicate nEdges false
93
94      let UFparents = replicate nEdges (-1)
95      in MSF UFparents edges edgeIds edgeIndexes smallestEdgeId included
96
97  -- These are the files from the small PBBS benchmark
98  -- ==
99  -- input @ data/randLocalGraph_WE_10_2000000.in
100 -- output @ data/randLocalGraph_WE_10_2000000.out
101 -- input @ data/rMatGraph_WE_12_2250000.in
102 -- output @ data/rMatGraph_WE_12_2250000.out
103 -- input @ data/3Dgrid_WE_8000000.in
104 -- output @ data/3Dgrid_WE_8000000.out
```

Listing 12 – Futhark Implementation of Min Spanning Forest