UNIVERSITY OF TWENTE.

Faculty of Science and Technology

Verifying GPGPU Implementation of Breadth-First Search Algorithm

Bachelor Thesis

Jan Smits (s1860704)

Bachelor Assignment Committee
prof. dr. M. Huisman (Chairperson)
M. Safari (Daily Supervisor)
dr. N. Sikkel (External Member)

Faculty of Science and Technology
University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Harish' Algorithm is the first GPGPU implementation of the Breadth-First Search algorithm for graphs. This thesis focuses on verifying data race freedom and functional correctness of Harish' algorithm using VerCors, a tool to verify parallel and distributed software developed at University of Twente. It was found beforehand that there are two data race conditions present in Harish's algorithm, which were resolved first. Method contracts using pre- and postconditions and permissions for variables were used to implement the fixed algorithm in PVL, the language used for VerCors, to verify data race freedom successfully. For this a different data structure than used originally in Harish's algorithm was developed that focussed on graph edges instead of vertices. This was done to help VerCors with verifying and also balances the total workload across all threads for increased efficiency. Then this implementation was used as input for an attempt to verify functional correctness using first a recursive path finding algorithm, and secondly operations and ghost variables. Both attempts failed and due to time constraints no further progress was made. Finishing this by introducing lemmas or a different data structure is recommended as future work.

Contents

Chapter 1

Introduction

1.1 Motivation

Graph theory is the study of graphs, not to be confused with graphs of mathematical functions. Graphs are depicted as networks of points connected by lines. In the context of graph theory a point is called a vertex (plural vertices) and the lines edges. A graph is a simple yet powerful method to describe a wide range of systems. They can represent complex molecules, at which point isomorphism can be used to prove that two molecules are the same. When planning the shortest route to your destination your navigation system uses graph theory.

Many algorithms have been made for analysing graphs. The breadth-first search (BFS) algorithm defines a source vertex and determines the minimum number of edges to reach the source from any other vertex in the graph. It does this starting from the source, and spreads out searching the graph in an even manner, hence the breadth part. Depth-first search uses the same premise, but searches along a whole chain of vertices first, before considering other neighbours of the source. These algorithms can be used to, for example, determine a group of vertices that are all connected or to find the minimum spanning tree in an unweighted graph.

Graph size can range into millions of vertices. When performing algorithms on such graphs using conventional hardware on a CPU, the computation time can be exceptionally long. This is because a sequential implementation of the algorithm is simply not fast enough, and a conventional CPU is not optimised for parallel computing. Therefore implementations of algorithms were made on a GPU. GPUs are specialised in parallel computing since it is required for processing graphics. This can be used to accelerate the algorithms for large graphs. The use of GPUs for purposes other than graphical computation is called General Purpose GPU, or GPGPU. This way every vertex in a graph can have its own dedicated thread, running a vertex-specific computation.

However, because of this parallel nature, special care must be taken when creating such implementations. The order in which threads execute is not predetermined and thus it should be verified that the outcome is independent of the order. In other words it must be verified that there are no data race conditions. Performance of the algorithm

must also be considered. Workload imbalance between threads will negatively impact the performance time [7]. Functional correctness, ensuring the algorithm provides the correct result, must also be proven afterwards.
This thesis focuses on verifying data race freedom and functional correctness in a breadth-first search algorithm implemented on Nvidia's CUDA [1], called Harish's algorithm. The **research question** for this thesis is therefore; Is Harish's algorithm free of data race conditions, and is it functionally correct?
The paper in which Harish's algorithm was published showed one of the first implementations of GPGPU algorithms. It gave a first look at the potential that GPGPU has and also contains algorithms for single source shortest path and all pairs shortest path in graphs.

1.2 Contributions

This project showed that the original BFS algorithm published by Harish et al ([1]) contains two data race conditions. In order to rectify this, a modified algorithm that uses an atomic operation and an comparison is proposed. It is then verified that this algorithm does indeed not contain data race conditions. For this purpose an alternative data structure is created. The work to verify functional correctness was started by formally defining constructs from graphs like paths and neighbours. A recursive algorithm to find all paths between two vertices is proposed to help verify functional correctness, but attempts to verify the functionality of this failed. Next ghost variables used with operations are introduced to the BFS algorithm, but verification failed thus far and is left unfinished.

In the remainder of this thesis the background of this project is first explained in chapter 2. Next Harish's algorithm is described in detail and the two data race conditions are addressed in chapter 3. Chapter 4 contains all work towards formally proving data race freedom of the modified algorithm. In chapter 5 an attempt is made to verify functional correctness. Finally chapter 6 shows related work and chapter 7 contains the conclusions and describes future work.

Chapter 2

Background

2.1 VerCors and PVL

In order to formally verify data race freedom and functional correctness the VerCors tool is used, developed by the Formal Methods and Tools research group at University of Twente [2]. VerCors supports verification of software in multiple languages such as Java, C and PVL. Prototypal Verification Language or PVL is a toy language created for prototyping new features for VerCors. It is used in this thesis because internally all code fed to VerCors is translated to PVL before verifying, so it contains all functionality necessary and is more efficient. OpenCL, a commonly used framework for parallel computing, would fit the requirements for this project, but the functionality necessary for this project is not supported yet by VerCors.
When verifying programs, every method gets a contract. This contract will state what is expected of the variables passed to the method and what the method ensures for the return value. A contract consists of pre- and postconditions which come in the form of a boolean expression. Preconditions will specify a property of the passed arguments while postconditions specify properties of the return value.
Another key aspect of verification are resources. With this is meant the permission to read from and write to variables. In PVL a thread will need permissions in order to manipulate values. In order to use the value of a variable a thread will first need read permission. When (re)assigning variables write permission is required. Read permission for a variable can be given to multiple threads simultaneously, whilst write permission for a variable can only be granted to one thread at a time. Write permission always automatically implies read permission. The specifying of permissions is also done in the method contract using pre- and postconditions, and they can be passed along to other methods called.
Apart from the primitive data types like `int` and `boolean`, PVL supports arrays and sequences amongst others. Arrays are declared like `T[] var`, where `T` is the type the array uses. Sequences are written `seq<T> var`. Nested sequences are possible, to create the concept of a matrix. While both arrays and sequences are ordered and allow duplicate values, sequences are immutable where the content of arrays can be reassigned.

In PVL, for a thread to read from or write to an array, permission is needed, whilst for sequences this is not necessary. When VerCors verifies code, it checks the contract of a method. Based on the preconditions and the operations performed in the method VerCors will determine whether the postconditions and assertions present are not violated. If they are, verification fails. Furthermore a thread may not access variables beyond their permission. Any runtime error such as a negative or out of bounds index will also cause the verification to fail.
Here follows a list of functions, specifications, constructs and their syntax in PVL used in this thesis.

- `requires expr` - Used to state that `expr` is a precondition for the associated method. `expr` must result in a boolean.
- `ensures expr` - Identical to `requires` but states a postcondition.
- `context expr` - Combines `requires` and `ensures` into one statement for the same expression.
- `invariant expr` - Used before a method. Creates a `context` for this expression and also ensures that it holds everywhere inside the associated method, such as loops by adding the expression as `loop_invariant`.
- `loop_invariant expr` - Used before a loop to state that the expression is preserved by the loop. This means that before and after each iteration of the loop `expr` holds. During an iteration the expression does not need to hold.
- `assert expr` - Can be used in the body of a method to assert that `expr` is true.
- `given T var` - Used before a method declaration to state that a ghost parameter of type `T` named `var` is given to the method.
- `Perm(var, p)` - Used to define the permission for the calling thread for `var`. `p` is the level of permission. When it is 1, write (and read) permission is declared. Any value between 0 and 1 gives read only permission. Alternatively `write` and `read` can be used instead of numerical values for `p`.
- `(\forall T var; range; expr)` - Quantified expression that defines `var` of type `T`, then states the range of `var`, and uses it in `expr`. Often used with arrays or sequences to state a property. `\forall*` is used in the same way but for the resource type, like permissions.
- `(\exists T var; range; expr)` - Quantified expression similar to `\forall`, except this evaluates to true only if there exists a value for `var` in the range specified such that `expr` is true.

- `invariant name(resources) {body}` - Invariant block used in combination with `atomic` blocks inside the body of the invariant. `resources` are the permissions and statements used during the associated atomic operation, referenced using the identifier `name`.
- `atomic(inv) {body}` - Defines an atomic operation in the `body`, an operation that is not interruptible by other threads, i.e. other threads only see the pre- and post state of the operation. The resources (permissions and expressions) for the operation are acquired from the invariant block `inv`.
- `par name (iterator) {body}` - Parallel block used to execute `body` in parallel. `iterator` holds an iterator with specified type, identifier and range that defines the number of threads. Every thread will hold one of the values of `interator` in the specified range.
- `\old(expr)` - Keyword that can be used to access the previous value of `expr`. When used in a postcondition it refers to the value before calling the method. When used in a loop it refers to the value of the previous loop iteration.
- `\result` - Keyword that can be used in postconditions to refer to the output of a method.
- `expr1 ==> expr2` An implication, only false if `expr1` is true while `expr2` is false.
- `expr ? expr1 : expr2` - Conditional expression. When `epxr` is true, `expr1` is returned, otherwise `expr2` is.
- `head(seq)` - Returns the first element in the given sequence.
- `tail(seq)` - Returns a sequence containing all elements in the given sequence except the head, in the same order.
- `pure` - Used before a method declaration to indicate it is a pure method. This means that the result of a method only depends on the passed arguments, and these are not modified in any way, and the method has no side effects.

For loops are relatively new in PVL, and to ensure that no problems arise from using new features, while loops are used exclusively in this thesis. More detailed information on the syntax of PVL can be found on GitHub [3].

2.2 GPGPU

GPGPU, or General Purpose Graphics Processing Unit, is the use of GPU's for general computing. This is done because of the parallel nature of the hardware of GPU's, which consists of hundreds processing cores as compared to the tens of a CPU where general computing conventionally takes place. In order to create and display graphics, a GPU

needs to have high parallel processing capabilities. A GPU can thus support thousands concurrent threads [4]. Therefore algorithms that require large amounts of computing can be implemented for parallel computation on a GPU. This way computation times can be reached that otherwise require supercomputers [1].
GPGPU programs can be created in OpenCL or using Nvidia's CUDA, which uses the language C to implement algorithms for execution on GeForce GPU's [4].
When executing GPGPU programs, the CPU or host will copy all required data to the memory of the GPU or device. The copying of data and instructions from the host to the device is a relatively slow process as is indicated in [12]. This is caused by the hardware connection between the two, the PCI bus. The data includes the kernel, which consists of the code that will be executed on the device. A kernel usually contains the code normally found in the body of a loop. Instead of a loop iterator, each thread executing the kernel has its own ID that is used in the kernel to perform ID specific operations. In the context of graphs, the kernel can be a vertex specific operation, and each thread represents one vertex, where the ID refers to the vertex number. When the kernel terminates the result is copied from the device back to the host, where it can be stored permanently or used further. A graph algorithm will perform many kernel calls compared to other algorithms [12].

2.3 Graphs

On a basic level, graphs consist of vertices and edges, depicted as circles and lines between those respectively. When concerning a directed graph, the edges appear as arrows, showing a one-directional connection. Weighted edges have a certain cost or weight associated with them, usually a positive integer. The degree of a vertex is equal to the number of neighbouring vertices that vertex has.
Breadth-first search is an algorithm to traverse a graph from a starting point, the source vertex. It is called breadth-first since all neighbours of the starting point are traversed before the neighbours of the neighbours are, see figure 2.1. This is opposed to depth-first search, which will traverse an entire branch starting at a neighbour and ending in dead ends before the next neighbour is considered.
Breadth-first search can have many applications. In unweighted graphs it can find the shortest path between the starting vertex and any other vertex [5]. It can also be used to see if two vertices are connected or not, and to find all vertices that are connected to the source vertex. More practical applications are in finding neighbouring nodes in peer to peer networks, traversing social networks to find the 'distance' between two people and in navigation systems to find all neighbouring locations. In most of these applications graphs are unweighted, directed and do not contain double edges, meaning there are no two edges that have the same start and end vertex, or self loops; edges that start and end at the same vertex. Therefore these properties are used in all graphs in this thesis.
Chapter 3 gives a more in-depth description of the specific breadth-first search algorithm being verified in this thesis (Harish's algorithm).

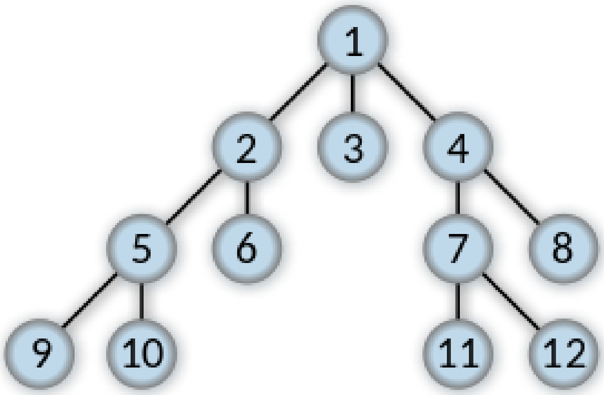

Figure 2.1: Order of traversal for BFS, where the top vertex is the source

2.4 Approach

In order to verify data race freedom and functional correctness, Harish's algorithm must be implemented in PVL. For this the workings of the algorithm must be understood in detail first, which is described in chapter 3.
A data structure is needed in order to represent a graph in code. When verifying data race freedom, this data structure is created with arrays, as opposed to sequences. This is done because data race freedom is proven using permissions for the data structure. Every thread will only receive the permissions they need according to the algorithm. When verifying, the VerCors tool will check that none of the permissions are violated at any time and that all the specifications hold. If that passes and the algorithm was correctly implemented, it is proven to be data race free [6].
The approach for verifying functional correctness is very different. Beforehand data race freedom must be proven. The desired result of the algorithm must first be determined. Using pure methods, definitions can be created. These will define how for example a graph or path is contained using the data structure. The implementation used for verifying data race freedom is augmented with ghost variables. These are used to mimic the workings of the algorithm using operations. The operations are present in code as pure methods. Finally the resulting values of the ghost variables are compared to the results of the algorithm. If they match and VerCors verifies the code, the algorithm works as intended.

2.5 Glossary

Many technical terms or abbreviations are used in this thesis. They are defined here in the glossary or included to prevent confusion.

CPU - Abbreviation for Central Processing Unit.

GPU - Abbreviation for Graphics Processing Unit.

GPGPU - Abbreviation for General Purpose GPU, the use of GPUs for other purposes than graphics.

BFS - Abbreviation for Breadth-First Search.

VerCors - The tool developed at University of Twente by the Formal Methods and Tools research group to formally verify programs.

PVL - Prototypal Verification Language, a toy language developed for VerCors.

Kernel - The part of code that runs in parallel on the GPU during GPGPU.

Neighbour - By neighbour of a vertex is meant a vertex that is directly reachable from the current vertex using a single edge.

Reverse neighbour - A vertex that can be reached from the current vertex by using a directed edge in the opposite direction. The current vertex is a neighbour of the reverse neighbour.

Source vertex - The starting point used for BFS.

Flag - A boolean that is used to indicate a binary property for something.

Frontier - In the context of BFS, the vertices that are currently actively traversed.

Cost - In the context of BFS, it is the positive integer associated with a vertex that indicates the number of edges required to reach that vertex from the source vertex.

Tail - The vertex that an edge is coming from.

Head - The vertex that an edge is going to. Not to be confused with the head of a sequence, the first element of that sequence.

Definition - In the context of verifying functional correctness, a pure method that returns a boolean stating whether the arguments passed satisfied the theoretical definition.

Operation - A pure method that can perform an operation using variables.

Reverse algorithm - The operations used in chapter 5 to perform a backwards version of BFS. It considers reverse neighbours instead of normal neighbours.

Chapter 3

The Algorithm

3.1 Workings

The algorithm being verified is a breadth-first search algorithm designed for GPGPU and is also known as Harish's Algorithm [1]. As input it requires a directed graph G and a source vertex S. As output all vertices in the graph will have an integer cost, equal to the minimum number of edges needed to reach that vertex from the source vertex. The algorithm consists of two parts. The first part runs on the host CPU, while the second part is called by the first part and runs on the GPU. The second part is the kernel, and is run for each vertex in parallel in the graph being analysed.

Part 1 of the algorithm; runs on the CPU

```
CUDA_BFS(Graph G, Source Vertex S)
Create vertex array $V_a$ from all vertices and edge Array $E_a$ from all edges in G.
Create frontier array $F_a$, visited array $X_a$ and cost array $C_a$ of size V.
Initialize $F_a$, $X_a$ to false and $C_a$ to $\infty$.
$F_a$[S] = true, $C_a$[S] = 0
while Fa not Empty do // outer while loop
    for each vertex V in parallel do
        Invoke CUDA_BFS_KERNEL(Va,Ea,Fa,Xa,Ca) on the grid. \\ execute the kernel on
    GPU
    end for
end while
```

Part 2 of the algorithm; the kernel; runs on the GPU

```
CUDA_BFS_KERNEL(Va,Ea,Fa,Xa,Ca)
tid = getThreadID // retrieve ID, which is uniquely linked to a vertex
if Fa[tid] then // when frontier flag is true, vertex can perform operation
    Fa[tid] = false // set own frontier flag to false
    Xa[tid] = true // current vertex has now been visited
    for all neighbours nid of tid do // loop over neighbours using identifier nid
        if NOT Xa[nid] then // if neighbour not yet visited
            Ca[nid] = Ca[tid] + 1 // update the cost of the neighbour to 1 higher than
    current vertex
            Fa[nid] = true // frontier flag for neighbour enabled, for next iteration
        end if
    end for
end if
```

The algorithm uses two flags for each vertex, as well as an integer cost. The first flag, called F_a in the pseudo code, is used to indicate whether a vertex is in the frontier. The frontier consists of all the vertices that are currently being traversed by the algorithm. The second flag, called X_a in the pseudo code, is used to indicate whether a vertex has already been visited before. This flag is there so that the algorithm doesn't reprocess old vertices, which is unnecessary. Initially, the visited flag is set to false for all vertices, and the frontier flag is only set to true for the source vertex, since that needs to be processed first. The cost of the source vertex is by definition 0, and at first the cost of all other vertices is set to infinity.

The outer while loop, line 6 of part 1, will run iterations of the kernel. It is not included in the kernel so that each thread will have to wait until all threads have performed one iteration before the next one starts. This prevents race conditions. During each iteration the flags and costs are changed, and the while loop only terminates if there are no more vertices with the frontier flag set to true.

In the kernel, the thread ID is used as an identifier for the corresponding vertex. First a check is made to see if the vertex has the frontier flag enabled. If so, the frontier flag is disabled, and the visited flag enabled. Then a loop checks each neighbouring vertex to see if it has been visited before. If not, that vertex' cost is set to one higher than the cost of the current vertex, and the frontier flag is enabled for that neighbour, so it will be processed in the same way in the next iteration.

The data structure in [1] to store the graph is an adjacency list and is also used here. Since this algorithm only deals with directed graphs, every edge is only represented once in an adjacency list. Memory space is limited on the GPU, so this makes it more desirable than an adjacency matrix, where every edge is stored twice. The data structure consists of two arrays V_a and E_a. V_a has length equal to the number of vertices plus 1, while E_a has length equal to the number of edges. Every index in V_a corresponds to a vertex, and the entry at that index is an index for E_a. Entries in E_a correspond to vertices. For an example, see figure 3.1. Here all neighbours of vertex 1 can be found by first looking in V_a. The value at index 1 in V_a is 2, and the next value is 3. This means all neighbours of vertex 1 can be found in E_a starting at index 2 up to (but not including) 3. Consequently vertex 1 has only one neighbour, the value at index 2 in E_a, vertex 2, which matches the drawn graph. It is required to include an extra value at the end of V_a, so that this process also works for the last vertex. This value must then be equal to the number of edges so that all the neighbours of the last vertex are the last values of E_a.

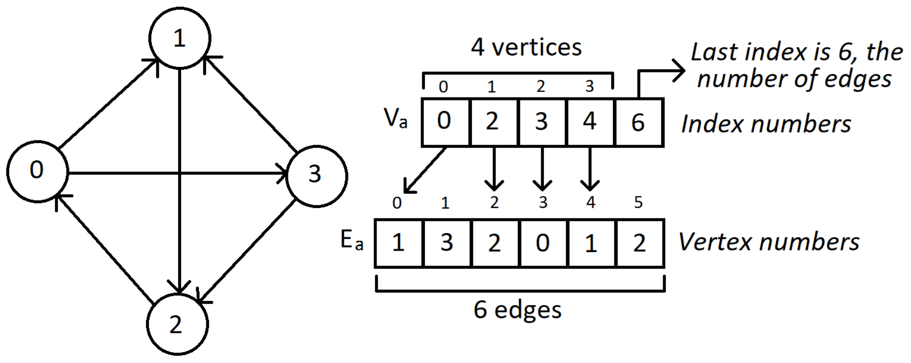

Figure 3.1: Example of the adjacency list data structure being used

3.2 Data Race

Unfortunately there are data race conditions present in the previously described algorithm. Since the order of execution of threads is not constant, there are two things that can go wrong.

3.2.1 Condition 1

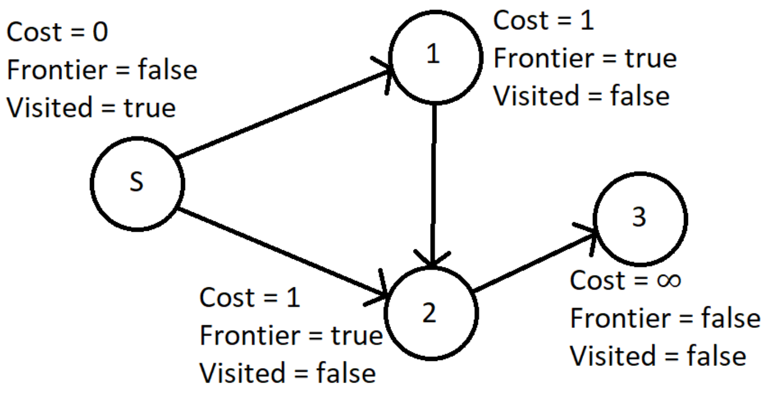

Figure 3.2: Example of first data race condition in Harish's BFS

In figure 3.2 a graph is seen after the first iteration of the kernel. The thread for vertex S (the source or start vertex) has set the cost and flags of vertices 1 and 2 correctly. In the upcoming iteration the threads for vertices 1 and 2 will be active. Note that vertex

2 is a neighbour of vertex 1. This means that should vertex 1 finish the kernel operation before vertex 2, the cost for vertex 2 would be set to 2, since the visited flag is still false. This is incorrect since the cost for vertex 2 should be 1. In the next iteration this fault would be carried to vertex 3. The problem here is that the algorithm only checks if the neighbours of a vertex are already visited, and not if they are currently in the frontier as well.
An easy solution for this problem would then be to not only check that the visited flag is false, but do the same for the frontier flag. However, with this solution in place the second data race condition still applies.

3.2.2 Condition 2

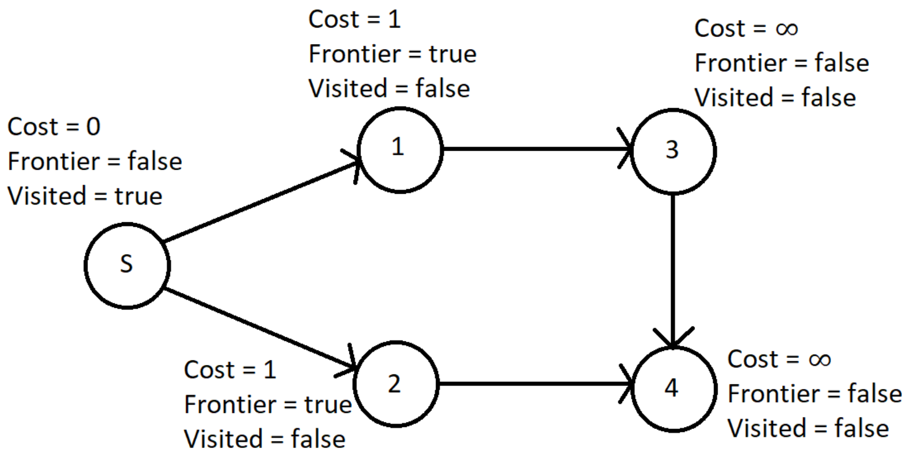

Figure 3.3: Example of second data race condition in Harish's BFS

It is assumed that the first data race condition has now been fixed in the manner described above. Figure 3.3 shows an example of the second data race condition. Much like figure 3.2 only 1 (correct) iteration of BFS has been performed here. This means that vertices 1 and 2 have the frontier flag set to true. The data race condition will occur in vertex 4. In the next iteration, vertices 1 and 2 will set the cost and frontier flag of vertex 3 and 4 respectively. However, the fact that the frontier flag for a vertex is false does not mean that there is no thread running for that vertex. The only thing that thread would do then is check for the frontier flag and, since it is false, terminate. If vertex 1 manages to set the frontier flag for vertex 3 to true before vertex 3 starts executing, then vertex 3 will also run the full algorithm in this iteration. If this were to happen then vertex 2 and 3 would contend for vertex 4. Since the cost of vertex 2 and 3 would not be the same, several things can happen. Vertex 3 can set the cost of vertex 4 to the wrong value of 3, and before the frontier flag is set to true vertex 2 can then also set the cost of vertex 4, since the frontier flag is still false. The final cost value would then be 2, which is correct. However it is also possible for a similar execution where vertex 3 sets the cost last, which would mean that the value would be 3. The problem here is that there is nothing stopping threads from interrupting each other when the same vertex is concerned. Checking whether or not the frontier flag is set to true for the neighbour vertex does not help in this case, since one thread can pass the check before the other thread can set the flag correctly. A solution to both data race conditions is presented in the next section.

3.3 Correction

There are two modifications that are needed to solve the data race conditions. First a comparison of costs is introduced as extra condition. This states that the cost of a neighbouring vertex will only be set if the cost of the current vertex plus 1 is lower than the cost of the neighbour. In other words, the cost value can only decrease, not increase. This will prevent vertices from setting the costs of their neighbours to an incorrect value. With this comparison, the visited and frontier flag are no longer a necessity.
However the operation of updating the cost can still be wrongfully interrupted. The second modification makes the operation atomic, meaning it can never be interrupted.

Modified kernel

```
tid = getThreadID
for all neighbors nid of tid do
    atomic {
        if Ca[tid] + 1 < Ca[nid] then
            Ca[nid] = Ca[tid] + 1
        end if
    }
end for
```

Although no longer necessary, flags do increase the efficiency of the algorithm. Using the modified kernel all neighbours will be checked regardless. This means the workload of a single thread is directly related to the degree of the concerned vertex. Since after each iteration all threads synchronise again, it can be assumed that the vertex with the highest degree will take the longest to process and is therefore the bottleneck. Therefore if flags are introduced again so that only the necessary vertices are processed the workload will generally decrease, increasing efficiency.
The two above data race conditions are now no longer present. However, it cannot be assumed that the modified kernel is data race free yet. The data race freedom of this modified kernel is verified in chapter 4

Chapter 4

Data Race Freedom

4.1 Initial attempt

The first attempt of verifying data race freedom uses the modified kernel described at the end of chapter 3, which does not have the data race conditions that were originally present. The source code can be found on page 17-18. This section explains the workings of this code.
In line 29 the method is declared and shows the arguments passed to the algorithm. Integer `V` is equal to the number of vertices, while integer `E` is equal to the number of edges. Integer arrays `vertices` and `edges` together form the adjacency list structure. Integer array `costs` holds the cost data for every vertex. Lastly the integer `source` represents the vertex number that is the starting vertex.
All preceding lines (lines 1-28) are method specifications for these variables. The requirements are used to specify the adjacency list data structure and its limits. The first 8 lines are invariants, which describe fundamental properties. None of the variables can be null, `V` and `E` must be greater than 0 so that the graph is not empty, the `source` integer must be a valid vertex number and the lengths of `vertices`, `edges` and `costs` must match `V` and `E`. The length of `vertices` is 1 longer, since the adjacency list structure requires that there always be a next value, including the last vertex.
As mentioned in Chapter 2, all edges in a graph are directed, there are no double edges and no self loops. When these properties are applied to the data structure of arrays the requirements on lines 20-23 emerge. Nested loops are required to access all the neighbours of a vertex, for all vertices. To ensure there are no double edges, the elements in `edges` are first sorted per vertex. Then no two consecutive elements can be the same. This way no vertex can have another vertex as neighbour twice. When these properties are combined it means that the vertex number of a neighbour must always be smaller than the number of the next neighbour.
Lines 24-28 ensure that `costs` is correctly initialised. A high value of 1 billion is used to reflect infinity. This number is sufficiently high enough, as it is assumed that the number of vertices in a graph stays below that number.
In the method contract all permissions that are needed are declared. Since the graph is not altered in any way in the algorithm, read-only permission is sufficient for `vertices` and `edges`. For `costs` write permission is required, since in each iteration of the algorithm the costs of vertices will change. In line 16-17 the read permission for `edges` is declared. This is done on a neighbour basis using a nested loop, instead of a simple loop over all the elements in `edges`, because in the algorithm the data will be accessed in a similar way, and declaring the permission like that helps the VerCors tool with verifying.
In line 30 the boolean flag `continue` is initialised to true. In the pseudo-code of the algorithm the outer while loop that runs on the CPU checks if there are still vertices

with the frontier flag set to true. Since this would require a loop to check all vertices and would not fit in one line of code, this flag is used instead. In line 49, at the start of each kernel iteration `continue` is set to false. The only way it can be set to true again so that the next iteration executes is if a vertex finds a neighbour whose cost can still be reduced. Then in line 94 `continue` is set to true again.
Line 31-47 lists all the loop invariants used for the outer while loop. These consist of repeating the properties and permissions defined before the method, except `costs` now is not initialised, but it is stated that the values are in between 0 and infinity.
The body of the outer loop starts by defining the resource invariant that is used in the atomic operation, line 50-53. All permissions for and properties of `costs` are needed for the atomic block and are thus included inside the invariant block. Although a thread only needs access to a few vertices, permission for the costs of all vertices is used here. On the GPU, when a thread runs the atomic block it will only use the permission for the vertices necessary. Meanwhile any other thread can enter the atomic block, if there is no overlap with another thread for the permissions needed. This way multiple threads can enter the atomic block, else much of the parallel execution would have been lost. The body of the invariant block contains the rest of the code.
Line 54 defines the parallel block that ensures the parallel execution of the kernel. An integer `tid` is defined in the range from 0 to V, so that there is one thread per vertex. In line 55-70 the contract for the parallel block is defined. These mainly follow the method contract, except that these are now vertex specific, instead of graph wide. Every thread only needs permission to read the current and next element in `vertices`, as seen in line 77, since these are necessary to find all the neighbours. In line 85 this is used to get read permission for all the elements in `edges` that contain neighbours. This way each thread has all the permission it needs, and nothing more.
Line 72-98 contains the body of the parallel block. First two integers are defined for the while loop that will loop over the neighbours of the current vertex. The same properties and permissions are then declared as loop invariants for this loop. In each iteration of the loop the atomic operation is performed. This consists of comparing the cost of the current vertex plus 1 with the cost of the neighbouring vertex. If that is the case the cost of the neighbour is set to the cost of the current vertex plus 1. Then the flag for the outer while loop is also set to true. Finally the loop iterator is incremented.

When VerCors tries to verify the above code, the verification process does not terminate. This is because the tool cannot prove that the algorithm is correct, but cannot find a counterexample to prove that it is incorrect either. Thus it keeps searching and finally if kept running for long enough, the tool will give an out of memory error. It is speculated that this is the fault of the data structure. It is possible that accessing the neighbours of a vertex in the current data structure is too complex for VerCors. When looping over the neighbours of a vertex, like in line 17, 19, 21 and 23, the nested loop uses elements of `vertices` as range, which means the tool first needs to read from `vertices` and check that the thread has permission to do so. It is uncertain whether this is the cause of the problem, but for the next attempt a new data structure is used to see if this helps.

Source code of initial attempt

```
invariant V > 0 && E > 0;
invariant source >= 0 && source < V;
invariant vertices != null;
invariant edges != null;
invariant costs != null;
invariant vertices.length == V + 1;
invariant edges.length == E;
invariant costs.length == V;
//Vertices
context (\forall* int i; 0 <= i && i <= V; Perm(vertices[i], read));
context (\forall int i; 0 <= i && i < V; vertices[i] >= 0 && vertices[i] < E);
context vertices[0] == 0;
context vertices[V] == E;
context (\forall int i; 0 <= i && i < V; vertices[i] <= vertices[i+1]);
//Edges
context (\forall* int i; 0 <= i && i < V;
    (\forall* int j; vertices[i] <= j && j < vertices[i + 1]; Perm(edges[j], 1/2)));
context (\forall int i; 0 <= i && i < V;
    (\forall int j; vertices[i] <= j && j < vertices[i + 1]; edges[j] >= 0 && edges[j] < V));
context (\forall int i; 0 <= i && i < V;
    (\forall int j; vertices[i] <= j && j < vertices[i + 1]; i != edges[j])); // not self-loop
context (\forall int i; 0 <= i && i < V;
    (\forall int j; vertices[i] <= j && j < vertices[i + 1] - 1; edges[j] < edges[j + 1])); // no double edges
//Costs
context (\forall* int i; 0 <= i && i < V; Perm(costs[i], write));
requires (\forall int i; 0 <= i && i < source; costs[i] == 1000000000); // initializations
context costs[source] == 0; // initializations
requires (\forall int i; source < i && i < V; costs[i] == 1000000000); // initializations
void Host_BFS(int V, int E, int[] vertices, int[] edges, int[] costs, int source) {
    boolean continue = false;
    loop_invariant (\forall* int i; 0 <= i && i <= V; Perm(vertices[i], read));
    loop_invariant (\forall int i; 0 <= i && i < V; vertices[i] >= 0 && vertices[i] < E);
    loop_invariant vertices[0] == 0;
    loop_invariant vertices[V] == E;
    loop_invariant (\forall int i; 0 <= i && i < V; vertices[i] <= vertices[i+1]);
    loop_invariant (\forall* int i; 0 <= i && i < V;
        (\forall* int j; vertices[i] <= j && j < vertices[i + 1]; Perm(edges[j], 1/2)));
    loop_invariant (\forall int i; 0 <= i && i < V;
        (\forall int j; vertices[i] <= j && j < vertices[i + 1]; edges[j] >= 0 && edges[j] < V));
    loop_invariant (\forall int i; 0 <= i && i < V;
        (\forall int j; vertices[i] <= j && j < vertices[i + 1]; i != edges[j])); // not self-loop
    loop_invariant (\forall int i; 0 <= i && i < V;
        (\forall int j; vertices[i] <= j && j < vertices[i + 1] - 1; edges[j] < edges[j + 1])); // no double
    edges
    loop_invariant (\forall* int i; 0 <= i && i < V; Perm(costs[i], write));
    loop_invariant (\forall int i; 0 <= i && i < source; costs[i] > 0 && costs[i] <= 1000000000);
    loop_invariant costs[source] == 0;
    loop_invariant (\forall int i; source < i && i < V; costs[i] > 0 && costs[i] <= 1000000000);
    while (continue) {
        continue = true;
        invariant inv((\forall* int i; 0 <= i && i < V; Perm(costs[i], write))
        ** ((\forall int i; 0 <= i && i < source; costs[i] > 0 && costs[i] <= 1000000000)
        && costs[source] == 0
        && (\forall int i; source < i && i < V; costs[i] > 0 && costs[i] <= 1000000000))) {
            par Device_BFS (int tid=0..V)
            context Perm(vertices[tid], read) ** Perm(vertices[tid + 1], read);
            context vertices[tid] >= 0 && vertices[tid] < A;
            context tid != V - 1 ==> vertices[tid + 1] >= 0 && vertices[tid + 1] < A;
            context tid == 0 ==> vertices[tid] == 0;
            context tid == V - 1 ==> vertices[tid + 1] >= 0 && vertices[tid + 1] == A;
            context vertices[tid] <= vertices[tid + 1];
            context (\forall* int j; vertices[tid] <= j && j < vertices[tid + 1]; Perm(edges[j], 1/2));
            context (\forall int j; vertices[tid] <= j && j < vertices[tid + 1]; edges[j] >= 0 && edges[j] < V);
            context (\forall int j; vertices[tid] <= j && j < vertices[tid + 1]; tid != edges[j]);
            context (\forall int j; vertices[tid] <= j && j < vertices[tid + 1] - 1; edges[j] != edges[j + 1]);
            {
            int start = vertices[tid];
            int end = vertices[tid + 1];

            loop_invariant start <= end;
            loop_invariant tid >= 0 && tid < V;
            loop_invariant Perm(vertices[tid], read) ** Perm(vertices[tid + 1], read);
            loop_invariant vertices[tid] >= 0 && vertices[tid] < A;
            loop_invariant tid != V - 1 ==> vertices[tid + 1] >= 0 && vertices[tid + 1] < A;
            loop_invariant tid == 0 ==> vertices[tid] == 0;
            loop_invariant tid == V - 1 ==> vertices[tid + 1] >= 0 && vertices[tid + 1] == A;
            loop_invariant vertices[tid] <= vertices[tid + 1];
            loop_invariant start >= vertices[tid] && start <= vertices[tid + 1];
            loop_invariant end == vertices[tid + 1];
            loop_invariant (\forall* int j; vertices[tid] <= j && j < vertices[tid + 1]; Perm(edges[j], 1/2));
            loop_invariant (\forall int j; vertices[tid] <= j && j < vertices[tid + 1]; edges[j] >= 0 && edges[j
    ] < V);
            loop_invariant (\forall int j; vertices[tid] <= j && j < vertices[tid + 1]; tid != edges[j]);
            loop_invariant (\forall int j; vertices[tid] <= j && j < vertices[tid + 1] - 1; edges[j] < edges[j +
     1]);
            while (start < end) {
                atomic(inv) {
                    if(costs[tid] + 1 < costs[edges[start]]) {
                        costs[edges[start]] = costs[tid] + 1;
                        continue = true;
                    }
                }
                start = start + 1;
                }
            }
        }
    }
}
```

4.2 Different Graph Data Structure

In order to try to help VerCors, a different data structure is used in the second approach to verify data race freedom. Instead of the adjacency list structure described in chapter 3 now two arrays with a length equal to the number of edges is used, `E1` and `E2`. Since edges are directed, each edge has a tail and head, the vertex the edge comes from and goes to respectively. `E1` is used to hold the tails of all edges, whilst `E2` holds all the heads. The elements in the arrays correspond to vertex numbers. This way two elements at the same index in `E1` and `E2` describe a single edge. Like figure 3.1, an example is given.

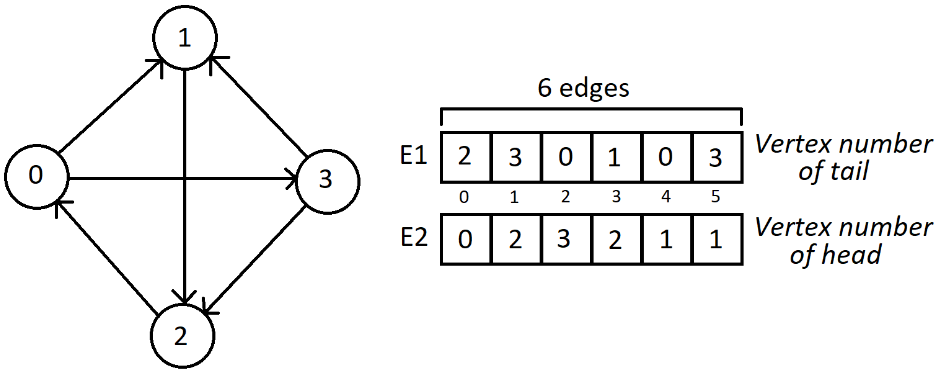

Figure 4.1: Example of the Edge lists data structure

For example the edge between vertex 1 and 2 is stored at index 3 in both arrays. In `E1` the tail, vertex 1, is stored there, while in `E1` the head, vertex 2 is stored there. The order in which edges are stored in this data structure is random. This is done because there is no advantage to any order, and there is no logical ordering in the first place.
With this new data structure the algorithm must also be modified. Finding all the neighbours of a vertex now requires a searching process. Therefore, the algorithm should no longer run one thread per vertex, but one thread per edge. That way every thread can compare the costs of the tail vertex with the head vertex and act accordingly. The `costs` array does not need to change for this. The source code for this approach can be found on the next page. The overall structure is similar to the first attempt, but modifications are made where the data structure is concerned. To ensure that there are no self-loops there simply cannot be an index where `E1` and `E2` have the same vertex number, line 14. Line 15-16 ensures there are no double edges. This is done with a nested loop that states that if the tails of two edges are the same, the heads can only be the same if it is the same edge. In other words, no two different edges must be the same. Again only read permission is needed for `E1` and `E2`. The outer while loop and invariant block have mostly not changed. The parallel block now defines the integer `tid` to range between 0 and `E`, ensuring every thread is linked to an edge using `tid`. In the contract for the parallel block the double edges property is not repeated, because it requires permission to read all elements of `E1` and `E2`, but one thread only receives access for one index. This does not pose a problem, since this property is also stated as a loop invariant in line 32-33.
The body of the parallel block is now shorter, as there is no need to loop over all the neighbours of a vertex. Instead the atomic operation compares the costs of the tail and head of the concerned edge, and if the cost of the tail plus 1 is lower than the cost of the head, the cost of the head is adjusted to the cost of the tail plus 1. Then the boolean flag `continue` is set to true, which still functions like in the initial attempt.

Source code of second attempt

```
invariant V > 0 && E > 0;
invariant E1 != null;
invariant E2 != null;
invariant costs != null;
invariant source >= 0 && source < V;
invariant E1.length == E;
invariant E2.length == E;
invariant costs.length == V;
//Edges
context (\forall* int i; 0 <= i && i < E; Perm(E1[i], read));
context (\forall* int i; 0 <= i && i < E; Perm(E2[i], read));
context (\forall int i; 0 <= i && i < E; 0 <= E1[i] && E1[i] < V);
context (\forall int i; 0 <= i && i < E; 0 <= E2[i] && E2[i] < V);
context (\forall int i; 0 <= i && i < E; E1[i] != E2[i]); // no self-loops
context (\forall int i; 0 <= i && i < E; (
    \forall int j; 0 <= j && j < E; i != j && E1[i] == E1[j] ==> E2[i] != E2[j]) ); // no double edges
//Costs
context (\forall* int i; 0 <= i && i < V; Perm(costs[i], write));
requires (\forall int i; 0 <= i && i < source; costs[i] == 1000000000);
context costs[source] == 0; // initializations
requires (\forall int i; source < i && i < V; costs[i] == 1000000000);
void Host_BFS(int V, int E, int[] E1, int[] E2, int[] costs, int source) {
    boolean continue = true;

    //Edges
    loop_invariant (\forall* int i; 0 <= i && i < E; Perm(E1[i], read));
    loop_invariant (\forall* int i; 0 <= i && i < E; Perm(E2[i], read));
    loop_invariant (\forall int i; 0 <= i && i < E; 0 <= E1[i] && E1[i] < V);
    loop_invariant (\forall int i; 0 <= i && i < E; 0 <= E2[i] && E2[i] < V);
    loop_invariant (\forall int i; 0 <= i && i < E; E1[i] != E2[i]); // no self-loops
    loop_invariant (\forall int i; 0 <= i && i < E; (
        \forall int j; 0 <= j && j < E; i != j && E1[i] == E1[j] ==> E2[i] != E2[j]) ); // no double edges
    //Costs
    loop_invariant (\forall* int i; 0 <= i && i < V; Perm(costs[i], write));
    loop_invariant (\forall int i; 0 <= i && i < source; 0 < costs[i] && costs[i] <= 1000000000);
    loop_invariant costs[source] == 0;
    loop_invariant (\forall int i; source < i && i < V; 0 < costs[i] && costs[i] <= 1000000000);
    while (continue) {
        continue = false;

        invariant inv( (\forall* int i; 0 <= i && i < V; Perm(costs[i], write))
                ** ((\forall int i; 0 <= i && i < source; costs[i] > 0 && costs[i] <= 1000000000)
                && costs[source] == 0
                && (\forall int i; source < i && i < V; costs[i] > 0 && costs[i] <= 1000000000)) ) {

            par Device_BFS (int tid = 0..E)

            context Perm(E1[tid], read);
            context Perm(E2[tid], read);
            context 0 <= E1[tid] && E1[tid] < V;
            context 0 <= E2[tid] && E2[tid] < V;
            context E1[tid] != E2[tid]; // no self-loops
            {
                atomic(inv) {
                    if (costs[E1[tid]] + 1 < costs[E2[tid]]) {
                        costs[E2[tid]] = costs[E1[tid]] + 1;
                        continue = true;
                    }
                }
            }
        }
    }
}
```

The efficiency of this approach is better compared to using threads for vertices. Every thread now has the same workload since only one comparison of costs is made per edge. The total workload is now balanced and spread over more threads. This way the parallel nature of a GPU is better utilised.
Contrary to the initial attempt, VerCors does verify this code. This could mean that using the new data structure did help VerCors with verifying. Since functionally both attempts are assumed to be the same, this means that the basic BFS algorithm is verified to be data race free. Of course the functional correctness of this approach must still be verified, which is attempted in chapter 5.
The remainder of this chapter focuses on adding flags to the verified algorithm. Doing this will not increase efficiency, since every thread already has very little work to do and there is no apparent bottleneck when it comes to the workload of all threads. In fact the adding of flags could decrease performance because of the extra need to read and write to memory for the flags. However this was realised only after they were already added. For the sake of this work the results are still included here (and it *is* used in chapter 5).

4.3 Adding 1 Flag

First one flag is added. This flag will serve a similar purpose to the frontier flag described in chapter 3. It will still be linked to a vertex, so that a thread for an edge will only compare costs if the tail vertex has this flag enabled. Initially only the source vertex will have the flag set to true, so in the first iteration all edges coming from the source vertex will be active. In order to add this flag to the source code, it requires write permission and needs to be initialised in the method contract to be false for all vertices except the source vertex. Then the permission is carried into the while loop with a loop invariant. The flag is set in the atomic block, so the permission to do so is also included in the invariant block. Then the atomic block is modified to only compare costs if the flag of tail of the edge is true. If the cost of the head vertex is adjusted, the flag for this vertex is also set to true, so that in the next iteration the adjustment can be carried further.

The atomic operation using one flag

```
atomic(inv) {
    if (flag[E1[tid]]) {
        if (costs[E1[tid]] + 1 < costs[E2[tid]]) {
            costs[E2[tid]] = costs[E1[tid]] + 1;
            flag[E2[tid]] = true;
            continue = true;
        }
    }
}
```

VerCors verifies the source code with this flag added. However, because a vertex can have multiple outgoing edges, the flag for a vertex cannot be set to false again at the end of the atomic operation. If this were to happen there exists a chance that the flag is set to false before other outgoing edges that rely on the same flag can compare costs, thus losing functionality. Therefore the flag must remain true for the rest of the runtime.

This means that once a thread has compared costs for the first time, it will continue to do so in every iteration afterwards, despite the fact it is no longer necessary. Thus the flags of all the vertices connected to the source will turn true, and afterwards the algorithm terminates.

4.4 Adding 2 Flags

The problem with using only one flag is that both reading from the flag and writing to the flag happens for the same variable. This problem can be solved by introducing a second flag. Now the first flag can be used by threads to read from to see whether they need to compare costs, whilst the second flag can be written to so that threads compare costs in the next iteration. In this way the first flag can be seen as a representation of the frontier flag in the current iteration, whilst the second flag contains the frontier flag for the next iteration.
In order to use the data contained in the second flag in the next iteration, it must be copied to the first flag beforehand. Then the second flag must be reset so that it can be used in the next iteration to store the flags for the iterations after that. For this purpose a second parallel block is introduced that runs after the first parallel block.

Parallel block to prepare the flags for the next iteration

```
par set_flags (int tid = 0..V)
invariant F0 != null;
invariant F0.length == V;
invariant F1 != null;
invariant F1.length == V;
context Perm(F0[tid], write);
context Perm(F1[tid], write);
{
    F0[tid] = F1[tid];
    F1[tid] = false;
}
```

Here `F0` and `F1` are the first and second flag accordingly. Both flags have a length equal to `V`, so that is also the range used for the parallel block.

The atomic operation using two flags

```
atomic(inv) {
    if (F0[E1[tid]]) {
        if (costs[E1[tid]] + 1 < costs[E2[tid]]) {
            costs[E2[tid]] = costs[E1[tid]] + 1;
            F1[E2[tid]] = true;
            continue = true;
        }
    }
}
```

The atomic block is very similar to the version with one flag, except now writing to the flag happens for `F1`, whilst reading is done from `F0`. The permissions for both flags is added similarly to the code as was done for the one flag. Write permission is required for both flags since they need to be adjusted for each next iteration in the second parallel

block.
VerCors verifies the source code with two flags, so it is data race free. This version seems more efficient than the one with only one flag, since the flags for threads can be disabled again after comparison of costs. However the second parallel block does add an inefficiency. As described in chapter 2, when a parallel block is executed, all necessary data must be copied from the host to the device, a relatively slow process. With the introduction of the second parallel block `F0` and `F1` must be copied to the memory of the GPU a second time, which happens every iteration. This could cost more time than is gained by using 2 flags instead of 1. A more efficient approach is possible with three flags.

4.5 Adding 3 Flags

With the introduction of a third flag there is no need for a separate parallel block to initialise flags for the next iteration. There are three stages for a flag. First it functions as a read flag, so that threads can see whether they need to compare costs. Secondly a flag can be written to so that it contains the data necessary for the next iteration. Finally a flag can be reset, in other words all booleans are false. This happens after the flag was written to and the data is copied to the read flag. These three stages can be contained in three separate flags and with each iteration the function of each flag can be rotated to the next. After three iterations each flag will perform the same function again.

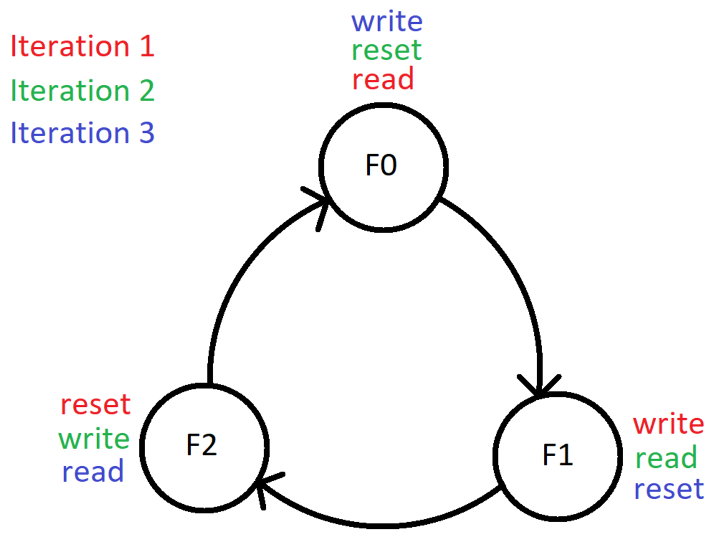

Figure 4.2: The rotational system of 3 flags

For this rotational system to work, there must be three parallel blocks where the flags rotate functions inside one iteration of the outer while loop. Each parallel block will then perform one iteration of BFS, so one iteration of the outer while loop will perform 3 iterations compared to previous versions. In order to rotate functions, the variable names are changed accordingly in each parallel block. These blocks can then be contained within the body of the invariant block. For each flag write permission is required, which is done in similar fashion as the code with two or one flags.

The atomic operation using three flags

```
1 atomic(inv) {
2     F2[E1[tid]] = false;
3     if (F0[E1[tid]] && costs[E1[tid]] + 1 < costs[E2[tid]]) {
4         costs[E2[tid]] = costs[E1[tid]] + 1;
5         F1[E2[tid]] = true;
6         continue = true;
7     }
8 }
```

First in the atomic blocks the elements in the reset flag are set to false, line 2. Then the atomic blocks combine the checking of the read flag and the comparison of costs into one if statement, line 3. This does not impede the efficiency, since the if statement requires both conditions to be true and if the read flag is false the comparison no longer needs to be checked. In line 5 the write flag is used. The actual flag number, `F0`, `F1` or `F2` is changed for these lines in the other two atomic blocks.
VerCors also verifies this approach. There is no need to copy memory from the host to the GPU in between execution of parallel blocks, as the data at the end of one block equals the data at the start of the next block. Alternatively just one parallel block could be used with barriers. The barriers function as points in the execution where all threads synchronise again, so one can be added after each atomic block. With each barrier specifications and permissions must then be reassigned. This approach has not been verified in this project.

Chapter 5

Functional Correctness

5.1 Definitions

In order to verify functional correctness of the BFS algorithm, the version that was verified by VerCors to be data race free in chapter 4, without flags, is used. This means that the data structure depicted in figure 4.1 is used here as well. The first step in verifying functional correctness is to formally write down several definitions in the context of graphs. These are contained in pure methods that return a boolean to indicate whether the arguments passed qualify for the definition. Sequences are used here instead of arrays, as arrays always require permissions to use, and although sequences cannot be edited, that is not necessary here.

5.1.1 Graph

First the definition of a graph itself is made, based on the data structure described in figure 4.1. A graph should be directed, have no self-loops or double edges, and should contain at least one vertex and edge.

Definition of a graph

```
pure boolean Graph(int V, int E, seq<int> E1, seq<int> E2) =
    V > 0 && E > 0 && |E1| == E && |E2| == E &&
    (\forall int i; 0 <= i && i < E; //loop over edges
        (0 <= E1[i] && E1[i] < V) && //valid vertex numbers
        (0 <= E2[i] && E2[i] < V) &&
        E1[i] != E2[i] && // no self loops
        (\forall int j; 0 <= j && j < E; (i != j && E1[i] == E1[j]) ==> E2[i] != E2[j])
    ); // no double edges
```

The arguments passed to this pure method are `V`, the number of vertices, `E`, the number of edges and `E1` and `E2`, that hold the data that describes the graph structure. In a graph the length of `E1` and `E2` is equal to `E`. Note that to access the length of a sequence in PVL, vertical bar characters are used, much like the cardinality of a set in set theory. Elements in `E1` and `E2` are vertex numbers and therefore should fall between 0 and `V`.

The requirements for no self-loops and no double edges are copied from chapter 4. There is no contract, as there are no pre- or postconditions that apply to the arguments.

5.1.2 Path

A path is defined as the possibility to travel or follow edges from vertex A to vertex B, directly or indirectly. Since graphs are directed, paths are one-way. A path is stored as a sequence of vertex numbers. The first number represents the starting vertex of the path. The next number depicts a neighbour of the starting vertex that is used in the path. The number after that is a neighbour of the previous number, etc. This repeats until the last number which is the destination of the path. The definition of a path therefore relies on the connection between elements in the sequence.

Definition of a path in the context of the graph structure

```
requires Graph(V, E, E1, E2);
pure boolean Path(int V, int E, seq<int> E1, seq<int> E2, seq<int> P) =
    (|P| == 1 && 0 <= P[0] && P[0] < V)
    ||
    (|P| > 1 &&
    (\forall int i; 0 <= i && i < |P|; 0 <= P[i] && P[i] < V) && //all numbers in P are
     in bounds
    (\forall int i; 0 <= i && i < |P|-1; //All vertices in P are connected
        (\exists int j; 0 <= j && j < E; E1[j] == P[i] && E2[j] == P[i+1])) &&
    (\forall int i; 0 <= i && i < |P|; //All vertices in P are unique / no revisiting
    vertices in the path
        (\forall int j; 0 <= j && j < |P|; i != j ==> P[i] != P[j])) );
```

In order to check whether `P` is a path, the graph in which the path takes place must be included. It is also required that this is a graph by definition. First the vertex numbers in `P` must be between 0 and `V`. Next all vertices in the path must form a connected chain, i.e. for every two consecutive elements there must exist an edge that connects them. Lastly all vertices in the path must only occur once. This is done so that there exists a finite number of paths between two vertices. Should there be a loop in the graph then a path could use that to technically form an infinite number of unique new paths between two vertices. This is prevented by this requirement. A path of length one is also allowed as an edge case, provided that the vertex is valid.

5.1.3 Reachability

Once the definition of a path is done, reachability is not hard to define.

Definition of reachability

```
requires Graph(V, E, E1, E2);
requires 0 <= s && s < V;
requires 0 <= e && e < V;
pure boolean reachable(int V, int E, seq<int> E1, seq<int> E2, int s, int e) =
    s == e || (\exists seq<int> P; Path(V, E, E1, E2, P); P[0] == s && P[|P|-1] == e);
```

A start vertex `s` and end vertex `e` are passed as arguments. The method returns true if `e` is reachable from `s`. In other words, there should exist a path where the first element is s and the last element is e. If s and e happen to be the same vertex, reachability is trivially confirmed.

5.1.4 Neighbours

The next definition simply states whether two given vertices are neighbours.

```
requires Graph(V, E, E1, E2);
requires 0 <= s && s < V;
requires 0 <= e && e < V;
pure boolean isNeighbour(int V, int E, seq<int> E1, seq<int> E2, int s, int e) =
    (\exists int i; 0 <= i && i < E; E1[i] == s && E2[i] == e);
```

Vertex e is a neighbour of vertex s if there exists an index for E1 and E2 where E1 is equal to s and E2 to e. In other words, e is a neighbour of s if there exists and edge from s to e.

5.2 Shortest Path Algorithm

The purpose of the BFS algorithm being verified is to determine the minimum number of edges required to reach a vertex from the source vertex. Therefore, in order to verify the functional correctness of the result, it must be proven that this is indeed the case. Before ghost variables are introduced as described in chapter 2, this section tries to prove this using paths. If all the possible paths from the source vertex to an arbitrary vertex x can be found, then the shortest path from the source to x can be determined from that. The length of the shortest path (read: number of edges contained in it) should equal the final outcome of the BFS algorithm for vertex x. If that is the case it is proven that the outcome of the algorithm is correct.
The approach taken here is to first create an algorithm to find all the paths between two vertices. Then the shortest path is determined from this. This is then used to find the shortest path between the source and all other vertices. The data race free verified BFS algorithm with no flags from chapter 4 is used the results are compared to the found shortest paths to determine functional correctness.

5.2.1 Recursive Algorithm

In order to find all paths between two vertices a recursive algorithm called explore was created. There was an attempt to find the shortest path in a non-constructive way using specifications, but this failed since VerCors deemed this not equal to the result of BFS.

The method contract for 'explore'

```
requires Graph(V, E, E1, E2);
requires Path(V, E, E1, E2, P);
requires 0 <= e && e < V;
requires !(e in P);
requires (\forall int i; 0 <= i && i < |paths|; Path(V, E, E1, E2, paths[i]));
requires (\forall seq<int> path; path in paths; |path| > 1 && path[0] == P[0] && path[|path|-1] == e);
requires (\forall int i; 0 <= i && i < |neighbours|; 0 <= neighbours[i] && neighbours[i] < V);
requires (\forall int i; 0 <= i && i < |neighbours|; isNeighbour(V, E, E1, E2, P[|P|-1], neighbours[i]));
ensures (\forall int i; 0 <= i && i < |\result|; Path(V, E, E1, E2, \result[i]));
ensures (\forall seq<int> path; path in \result; |path| > 1 && path[0] == P[0] && path[|path|-1] == e);
```

The recursive algorithm 'explore' to find all paths

```
1  seq<seq<int>> explore(int V, int E, seq<int> E1, seq<int> E2, int e, seq<seq<int>> paths, seq<int> P, seq<int> neighbours) {
2      int current = P[|P|-1];
3      int j = 0;
4      loop_invariant 0 <= j && j <= |neighbours|;
5      loop_invariant (\forall int i; 0 <= i && i < |paths|; Path(V, E, E1, E2, paths[i]));
6      loop_invariant Path(V, E, E1, E2, P);
7      loop_invariant (\forall seq<int> path; path in paths; |path| > 1 && path[0] == P[0] && path[|path|-1] == e);
8      while (j < |neighbours|) {
9          int neighbour = neighbours[j];
10         if (neighbour == e) {
11             seq<int> found = addToPath(V, E, E1, E2, P, e);
12             paths = addToPaths(V, E, E1, E2, paths, found);
13         } else if (!(neighbour in P)) {
14             seq<int> next = addToPath(V, E, E1, E2, P, neighbour);
15             seq<int> nextNeighbours = getAllNeighbours(V, E, E1, E2, next[|next|-1]);
16             paths = explore(V, E, E1, E2, e, paths, next, nextNeighbours);
17         }
18         j = j + 1;
19     }
20     return paths;
21 }
```

This algorithm is a form of depth-first search and requires several arguments as input. First a graph is passed. Second, a vertex `e` in that graph is given, which is the destination of the paths and the vertex being sought for. `paths` is a sequence of integer sequences that holds all the paths found so far that lead to `e`. `P` is a partial path to `e` originating from the source vertex. Lastly a sequence of all the neighbouring vertices of the last element in the path `P` is given.
Since the algorithm is recursive, many of the arguments are passed along to the next level of recursion. Initially the algorithm must be called with an empty sequence for `paths`, a path containing only the source vertex for `P` and the neighbours of the source vertex in a sequence for `neighbours`.
The algorithm always focuses on the last element in `P`, and saves it as the current vertex in line 2. Then a loop over the neighbours of `current` is defined in lines 8-19 preceded by loop invariants. The purpose of the loop is to check whether any of the neighbours are vertex `e`, which is done in line 10. If that is the case, then a new path from the source to `e` has been found and is added to `paths` in line 12. If the neighbour is not

vertex e, then the neighbourhood of that vertex must also be searched by recursively calling the algorithm. First the arguments for the recursive call are initialised. In line 14 a new path is defined by adding the neighbour to the end of P. In line 15 the sequence of neighbours of that neighbour is retrieved. In line 16 the recursive call is made with these variables. This process of checking is done for all neighbours of current. All found paths to vertex e are stored in paths and passed along in the recursion until ultimately it returns it. By then it will contain all possible paths starting from the source vertex originally passed in P to e. Path P simply remembers the chain of vertices that has been followed so far. The source vertex is not passed as argument directly, but is contained at the front of P. In line 13 a condition ensures that any vertex that is added to P for the next recursive call was not already contained in P. This is required for the definition of a path, as is described earlier in this chapter. In lines 11, 12, 14 and 15 other methods are called to add a vertex to a path, a paths to paths or to find all the neighbours of a vertex. This is done in a separate method to have more control over the contract for such an operation so that it can be proven that it performs its task correctly.
The operation of adding a vertex to a path simply does this by addition of the current sequence and a new sequence containing only the concerned vertex. The contract ensures that the resulting sequence is still a path using the path definition. A similar approach is used to add a path to a sequence of paths.
Finding all the neighbours of a vertex is done by looping over E1 to see if any edges start at the vertex. Then the head of any such edges is added to a sequence which in the end is returned. The neighbour definition is used in the contract to ensure all elements in the result are neighbours of the concerned vertex, and that no neighbours have been omitted.
The contract of the algorithm itself both requires and ensures that all collected paths in paths are in fact paths and that they start at the first element in P (the source vertex) and end in e (line 5-6 and 9-10 of the method contract). Furthermore the neighbour sequence is required to be valid, line 7-8 of the contract.

5.2.2 getMinCost

In order to use this algorithm, another method called getMinCost is defined.

Method getMinCost to call the recursive algorithm

```
requires Graph(V, E, E1, E2);
requires 0 <= s && s < V;
requires 0 <= e && e < V;
ensures 0 <= \result && \result <= 1000000000;
ensures s == e ==> \result == 0;
int getMinCost(int V, int E, seq<int> E1, seq<int> E2, int s, int e) {
    if (s == e) {
        return 0;
    }
    seq<seq<int>> paths = seq<seq<int>> {};
    seq<int> P = seq<int> {s};
    seq<int> neighbours = getAllNeighbours(V, E, E1, E2, s);
    paths = explore(V, E, E1, E2, e, paths, P, neighbours);
    int minimum = getSmallestPath(paths);
15      return minimum;
16  }
```

This method is called with the source vertex `s`, another vertex `e` and will return the length of the shortest path between `s` and `e`. It does this by initialising the variables needed to call `explore` in line 10-12. Then all paths are found between `s` and `e` using `explore`. In line 14 operation `getSmallestPath` is called with `paths`. This will return the length of the smallest path in the sequence. Should `paths` be empty infinity is returned as vertex `e` is then unreachable. This minimum path length is then returned to the caller of `getMinCost`. Lines 7-9 are used for the trivial case where `s` and `e` are the same, so that the result is 0. The result will always be a number between or equal to 0 and infinity.

5.2.3 BFS and Main

Next the BFS algorithm itself is included. It is equal to the implementation verified to be data race free with no flags in chapter 4, except the cost labels of vertices are now stored in a matrix of integers. Each column matches a vertex and each row represents an iteration. The matrix is returned at the end of the algorithm. This way not only the final result but all intermediate results are stored. This is done so that an important loop invariant can be introduced for the outer while loop in the BFS algorithm. That loop invariant states that the cost of every vertex at that time is always less than or equal to the number of loop iterations that have passed, or equal to infinity. This was added to prove this property formally and it could help VerCors when comparing the results of `getMinCost` to the results of BFS.
The final method used is the main method. This will perform BFS and find the length of the shortest path to each vertex from the source and compare both findings. If they are equal then BFS works correctly.

Main method used to compare results

```
requires Graph(V, E, E1, E2);
requires 0 <= s && s < V;
requires |costs| == 1;
requires |costs[0]| == V;
requires (\forall int i; 0 <= i && i < s; costs[0][i] == 1000000000);
requires costs[0][s] == 0;
requires (\forall int i; s < i && i < V; costs[0][i] == 1000000000);
void main(int V, int E, seq<int> E1, seq<int> E2, int s, seq<seq<int>> costs) {
    seq<seq<int>> outcome = runBFSSeq(V, E, E1, E2, s, costs); //run BFS
    seq<int> finalCosts = outcome[|outcome|-1]; //The final result is the last element in outcome
    int i = 0;
    loop_invariant 0 <= i && i <= V;
    while (i < V) { // comparing the outcome of BFS and getMinCost
        int minCost = getMinCost(V, E, E1, E2, s, i);
        if (i != s) {
            assert finalCosts[i] == minCost - 1;
        } else {
            assert finalCosts[i] == minCost;
        }
        i = i + 1;
        }
}
```

The preconditions for this method are used to create and initialise all the variables that are needed. Lines 3-7 initialise the costs matrix used for the modified BFS algorithm which is performed in line 9. Line 10 extracts the final costs sequence from the matrix. Then a loop is used to calculate the shortest path length and compare this to BFS for each vertex. Line 16 performs this comparison with an assertion. Note that the BFS result must equal `minCost` minus 1. This is because the number of edges in a path is equal to the number of vertices used in a path minus 1. This property does not hold for the case that the cost of the source vertex is compared since `getMinCost` then returns 0. Hence the if statement in line 15 and the comparison in line 18.

5.2.4 Result

When verifying all the methods above combined using VerCors the assertion in line 16 in `main` fails because it returns false. This means that this attempt to verify functional correctness has failed. This is most likely caused by underspecification of the methods used. The recursive algorithm `explore` should return *all* unique paths to a vertex, but this property is never formally proven. Attempts to add this as a postcondition were made but VerCors did not successfully verify them. Consequently the method `getMinCost` is also underspecified. It can only guarantee that the length of the shortest path is a value between 0 and infinity, which is rather trivial. Since the return value of `getMinCost` is directly compared to the outcome of BFS, VerCors cannot prove that these values should equal. However, the rest of the methods did verify correctly. This means that the property added to BFS that the cost of a vertex is less than or equal to the number of passed iterations, or is infinity, holds. In the next section a different approach to verification of functional correctness is tried.

5.3 Reverse Strategy

Because of the negative result of the preceding code, a new approach is tried here using ghost variables. In order to mimic the BFS algorithm more, a 'reverse' algorithm is made. This algorithm will consist of operations that will be able to perform an iteration of BFS alongside the already data race free BFS algorithm with no flags. The major difference is that a vertex X will inspect all vertices for which X is a neighbour instead. These vertices are considered reverse neighbours of X. In a sense this approach looks 'backwards' whilst the original BFS algorithm looks 'forward'. The verified algorithm is modified to include ghost variables, which are then used with the operations and then compared to the result of BFS to verify functional correctness.

5.3.1 Operations

This approach uses several operations to perform the 'backward' operation. All these operations are recursive. Since an operation is a pure method in code, only a simple task can be performed. By recursively calling itself with different arguments an operation can perform more complicated work. Another way to do this is by calling other operations. The main operation used here is `update_cost` and does just that.

Main operation used for updating costs

```
1  requires Graph(V, E, E1, E2);
2  requires 0 <= i && i <= V;
3  requires |costs| == V;
4  requires (\forall int i; 0 <= i && i < |costs|; 0 <= costs[i] && costs[i] <= 1000000000);
5  ensures |\result| == V - i;
6  ensures (\forall int j; 0 <= j && j < |\result|; 0 <= \result[j] && \result[j] <= 1000000000);
7  ensures (\forall int j; 0 <= j && j < |\result|; \result[j] <= costs[i+j]);
8  ensures (\forall int j; 0 <= j && j < |\result|;
9      getMinCost(reverseNeighbours(V, E, E1, E2, j+i, 0), costs, 1000000000) + 1 < costs[j+i] ==>
10     \result[j] == getMinCost(reverseNeighbours(V, E, E1, E2, j+i, 0), costs, 1000000000) + 1);
11 ensures (\forall int j; 0 <= j && j < |\result|;
12     getMinCost(reverseNeighbours(V, E, E1, E2, j+i, 0), costs, 1000000000) + 1 >= costs[j+i] ==>
13     \result[j] == costs[j+i]);
14 pure seq<int> update_cost(int V, int E, seq<int> E1, seq<int> E2, seq<int> costs, int i) =
15     i < V ?
16         (getMinCost(reverseNeighbours(V, E, E1, E2, i, 0), costs, 1000000000) + 1 < costs[i] ? //The comparison
17             seq<int> {getMinCost(reverseNeighbours(V, E, E1, E2, i, 0), costs, 1000000000) + 1} + update_cost(V, E, E1, E2, costs, i+1)
18             : seq<int>{costs[i]} + update_cost(V, E, E1, E2, costs, i+1))
19     : seq<int>{};
```

The purpose of this operation is to perform an iteration of BFS, but by 'looking' backwards instead of forwards. As arguments it receives a graph and a sequence of the current costs as well as iterator `i`. `i` is used to indicate which vertex or element in `costs` is currently being compared. It returns a sequence of integers that will be the new cost values. When executing it is first checked that `i` is smaller than `V` in line 15, if that is the case then in line 16 a comparison is made using other operations. First all reverse neighbours of vertex `i` are retrieved using `reverseNeighbours`. Then the return value of that is used in `getMinCost`, which will return the lowest cost amongst the reverse neighbours. If this cost plus 1 is smaller than the cost of vertex `i` then much like BFS in chapter 4 this becomes the new cost value for vertex `i`. This is done in line 17 by creating a new sequence using that value and then adding that to the result of the recursive call. This calls the pure method with the same arguments except `i` is incremented by 1. This way the next vertex is checked. The idea is that this method is initially called with `i` equal to zero after which all vertices are checked. Then when `i` becomes larger than `V` an empty sequence is returned in line 19. Since this is a recursive method, the return value will not be the same for each level of recursion. Furthermore it is theoretically possible to call this method initially with a value higher than zero. The method contract keeps this into account. For example line 5 describes a formula for the length of the return sequence, `V` minus `i`. Line 8-13 describe the operation as a postcondition for all elements in the result, to ensure that it works as intended.

The next operation is `reverseNeighbours`, which finds all the reverse neighbours of a vertex.

Lemma for finding all reverse neighbours of a vertex

```
requires 0 <= i && i <= E;
requires Graph(V, E, E1, E2);
requires 0 <= vertex && vertex < V;
ensures (\forall int j; 0 <= j && j < |\result|; 0 <= \result[j] && \result[j] < V);
ensures (\forall int j; 0 <= j && j < |\result|; isNeighbour(V, E, E1, E2, \result[j], vertex)); //verifies all found ARE reverse neighbours
ensures (\forall int j; i <= j && j < E && E2[j] == vertex; E1[j] in \result); // verifies ALL reverse neighbours are found
pure seq<int> reverseNeighbours(int V, int E, seq<int> E1, seq<int> E2, int vertex, int i) =
    i < E ?
        (E2[i] == vertex ?
            seq<int> {E1[i]} + reverseNeighbours(V, E, E1, E2, vertex, i+1)
            : reverseNeighbours(V, E, E1, E2, vertex, i+1))
    : seq<int> {};
```

Much like `update_cost` this pure method uses recursion and `i` as an iterator. Here `i` iterates over all edges in the provided graph. It checks for each edge whether the head is equal to `vertex`, which would mean the tail is a reverse neighbour of `vertex`. If found, the reverse neighbour is added to the resulting sequence. The method contract uses the definition of a neighbour to check that all vertices found are reverse neighbours, and another postcondition ensures that all possible reverse neighbours are found.

The next operation uses the result of `reverseNeighbours` to find the value of the lowest cost amongst the found reverse neighbours.

Lemma used for finding cheapest reverse neighbour

```
1  requires 0 <= minimum && minimum <= 1000000000;
2  requires (\forall int i; 0 <= i && i < |reverse_neighbours|; 0 <= reverse_neighbours[i] && reverse_neighbours[i] < |costs|);
3  requires (\forall int i; 0 <= i && i < |costs|; 0 <= costs[i] && costs[i] <= 1000000000);
4  ensures \result <= minimum;
5  ensures |reverse_neighbours| > 0 ==> \result <= costs[head(reverse_neighbours)];
6  ensures (\forall int i; 0 <= i && i < |reverse_neighbours|; \result <= costs[reverse_neighbours[i]]);
7  ensures \result >= 0;
8  pure int getMinCost(seq<int> reverse_neighbours, seq<int> costs, int minimum) =
9      |reverse_neighbours| > 0 ?
10         (costs[head(reverse_neighbours)] < minimum ?
11             getMinCost(tail(reverse_neighbours), costs, costs[head(reverse_neighbours)]) :
12             getMinCost(tail(reverse_neighbours), costs, minimum))
13         : minimum;
```

Instead of using an iterator to go through the reverse neighbours, the sequence is made shorter in each level of recursion, where in each level the head of the sequence is inspected. An integer `minimum` is used to track the smallest cost value found so far. Initially this value must be infinity. If a cost is found lower than `minimum` that value becomes the new `minimum` in the next recursion level. In the end the final `minimum` is returned through all levels of recursion back to the initial caller. The method contract uses line 6 to ensure that the result is indeed the smallest cost value amongst the reverse neighbours.

5.3.2 BFS with Ghost Variables

The BFS version verified to be data race free with no flags from chapter 4 is used here. The ghost variables are introduced in the method contract using the keyword `given` and include two sequences to represent the graph like `E1` and `E2`, a sequence to mimic the array `costs` and finally a sequence matrix, a sequence of sequences, that is used to keep track of all values of the costs sequence across iterations of the algorithm, just like in the implementation of BFS in the first half of this chapter. The preconditions are used to initialise these ghost variables to have the same values as the arrays they mimic. The sequence matrix contains only the costs sequence at first.
After each iteration of BFS using the invariant and parallel block, the ghost variables are used by calling the operation `update_cost` along with integers `V` and `E` to run an iteration of the reverse strategy. The result of this is stored as the new value for the costs sequence. This is then also added to the sequence matrix.
The loop invariants for the outer while loop are used to state properties that hold for both the ghost variables and arrays after each iteration. First it is checked that the ghost variables for the graph are still equal to `E1` and `E2`. Next it is asserted that the current values of both `costs` and the costs sequence are smaller than or equal to the old values, that is the values before the last iteration. This says that the cost values are always decreasing with each iteration. Then it is checked that the sequence matrix has been correctly updated with the latest cost sequence. Finally it is checked whether the values in `costs` are equal to the costs sequence. If this is the case then that means the ghost variables successfully copied the results of the algorithm using the operations and functional correctness is verified.

5.3.3 Initial Results

When verifying the operations together with the BFS algorithm with ghost variables using VerCors everything passes except the loop invariant that states that the costs sequence updated using the operation `update_cost` equals `costs` which is used in BFS. Thus this attempt to verify functional correctness failed. The cause for this is explained here. First of all there is an important difference between the BFS algorithm and reverse strategy. The former focuses on edges and performs comparisons according to an edge. The latter focuses on vertices and compares with all reverse neighbours for a vertex. The order in which both algorithms compare costs is different and random. For example in a graph which just consists of a chain of vertices it is possible based on the order of execution that the cost of more than one vertex in the chain is updated in a single iteration for one algorithm, whilst the other algorithm does not do this. This way a difference is created between the ghost variable and costs array and thus their values do not match. In an attempt to solve this the reverse algorithm can be modified to also work with edges instead of vertices. This way both algorithms compare costs in the same order. Then in the reverse algorithm all reverse neighbours of the head of an edge are checked for costs. However, even then there are discrepancies between both algorithms. An example of this problem is given here.

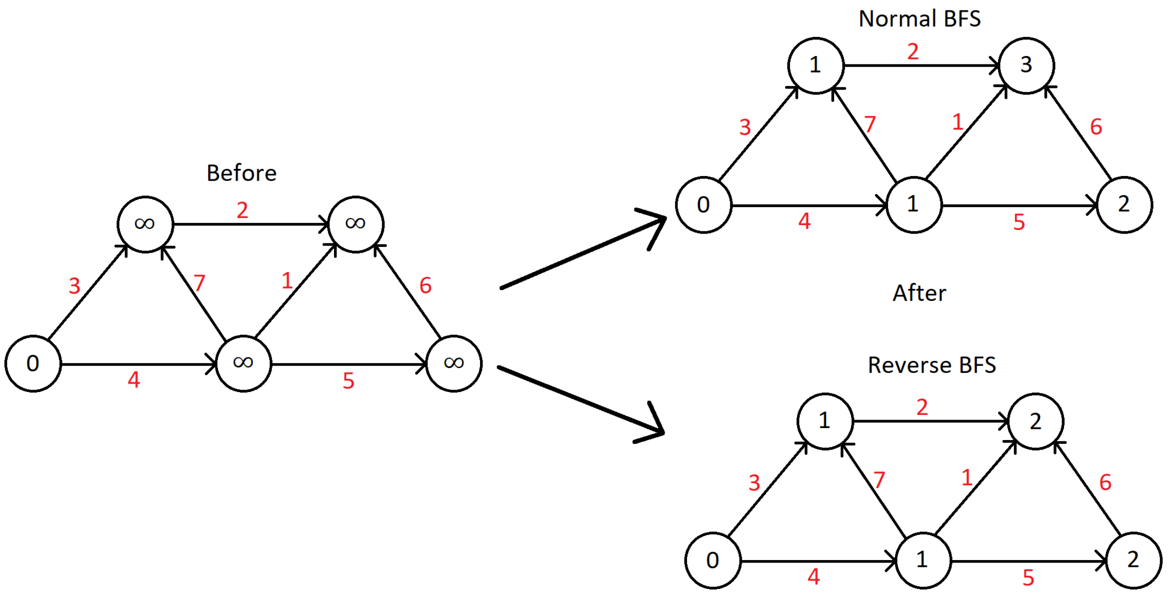

Figure 5.1: The critical difference between both algorithms

The red numbers next to edges indicate the order in which edges are executed. The numbers in vertices indicate the cost label of that vertex. The discrepancy in this example lies in the top right vertex. The edge with label 1 will be executed first, however this will have no impact on the cost values as at that moment the costs of both tail and head are infinity. The same goes for the edge labelled 2. It is not until edge 6 that the cost of the top right vertex is updated. In the case of normal BFS the cost will be set to 3, since the bottom right vertex has cost 2. This is incorrect now but will be corrected in the next iteration. In the case of reverse BFS the cost is set correctly the first time, because one of the reverse neighbours of the top right vertex already had cost 1. This is the fundamental difference between both algorithms. The normal BFS will only compare the cost of the tail of the current edge, whilst reverse BFS will compare the costs of all reverse neighbours of the head of the current edge. This problem can be fixed by using flags. If two flags are used like in chapter 4, There is always a predetermined number of edges that will be executed in an iteration. Then the operations used to perform the reverse BFS need to also keep track of and update flags. Furthermore the operation for finding the minimum cost among the reverse neighbours of a vertex, `getMinCost`, should then also hold flags into account so that only reverse neighbours with a flag enabled are considered. This way the example of figure 5.1 is prevented. This modification is used in the next attempt.

5.3.4 Using Two Flags

In order to solve the issues found and described above reverse BFS is modified to use two flags. Then the BFS algorithm using 2 flags from from chapter 4 is used with the ghost variables. Two extra ghost variables are introduced to mimic the behaviour of the two flag arrays present in that version. They are initialised using preconditions much like the other ghost variables. Now the cost of a reverse neighbour is only considered if the ghost flag for that vertex is set to true. In order to update the ghost flags alongside updating the cost a new operation is introduced. This operation is used at the same time as `update_cost` to update the read ghost flag `F0` for the next iteration. Another operation is used to reset the ghost variable for `F1` to false.

Lemma used to update the ghost flag

```
1  requires Graph(V, E, E1, E2);
2  requires 0 <= counter && counter < V;
3  requires 0 <= i && i <= V;
4  requires |costs| == V;
5  requires |F0_seq| == V;
6  requires (\forall int i; 0 <= i && i < |costs|; 0 <= costs[i] && costs[i] <= 1000000000);
7  requires (\forall int j; 0 <= j && j < V; costs[j] <= counter || costs[j] == 1000000000);
8  ensures |\result| == V - i;
9  ensures (\forall int j; 0 <= j && j < |\result|;
10     readFlag(V, reverseNeighbours(V, E, E1, E2, j+i, 0), F0_seq, 0) && getMinCost(V,
       reverseNeighbours(V, E, E1, E2, j+i, 0), costs, 1000000000, counter, 0, F0_seq) + 1 < costs[j+i]
        ==>
11     \result[j] == true);
12 ensures (\forall int j; 0 <= j && j < |\result|;
13     readFlag(V, reverseNeighbours(V, E, E1, E2, j+i, 0), F0_seq, 0) && getMinCost(V,
       reverseNeighbours(V, E, E1, E2, j+i, 0), costs, 1000000000, counter, 0, F0_seq) + 1 >= costs[j+i
       ] ==>
14     \result[j] == false);
15 ensures (\forall int j; 0 <= j && j < |\result|; !readFlag(V, reverseNeighbours(V, E, E1, E2, j+i,
       0), F0_seq, 0) ==> \result[j] == false);
16 ensures (\forall int j; 0 <= j && j < E && F0_seq[E1[j]];
17     (i <= E2[j] && E2[j] < V && (costs[E1[j]] + 1 < costs[E2[j]])) ==> \result[E2[j]-i] == true );
18 ensures (\forall int j; 0 <= j && j < E && F0_seq[E1[j]];
19     (i <= E2[j] && E2[j] < V && !(costs[E1[j]] + 1 < costs[E2[j]])) ==> \result[E2[j]-i] == false );
20 pure seq<boolean> update_flag(int V, int E, seq<int> E1, seq<int> E2, seq<boolean> F0_seq, seq<int>
       costs, int i, int counter) =
21     i < V ?
22         (readFlag(V, reverseNeighbours(V, E, E1, E2, i, 0), F0_seq, 0) && getMinCost(V,
       reverseNeighbours(V, E, E1, E2, i, 0), costs, 1000000000, counter, 0, F0_seq) + 1 < costs[i] ?
       //The comparison
23             seq<boolean> {true} + update_flag(V, E, E1, E2, F0_seq, costs, i+1, counter)
24             : seq<boolean> {false} + update_flag(V, E, E1, E2, F0_seq, costs, i+1, counter))
25     : seq<boolean>{};
```

Like `update_cost` this operation makes the same comparison of costs by using the reverse neighbours, but now instead of returning the new costs it returns a sequence of booleans that is equal to the write flag or `F1`. When the cost of a vertex would have been updated the flag is set to true, false otherwise. The method contract has been expanded. Line 15 makes use of operation `readFlag`. This operation checks whether one of the reverse neighbours of a vertex has its flag enabled and will return true then, otherwise it returns false. Line 15 states that if none of the reverse neighbours of a vertex have their flag enabled, then the resulting flag for that vertex should also be false. Line 16-19 are important. Line 16-17 check for all edges that if the flag is true for the tail vertex, and the cost of the head would be updated, then the flag for that vertex should be true. The logical inverse of that statement is written in line 18-19. What is special about this postcondition is that it reasons like the normal BFS algorithm, and not in reverse. Thus it is checked that the operation produces the same results as the normal algorithm would have.

For the outer while loop in normal BFS new loop invariants are added. Most notably it is checked that the ghost variables for the flags equal the flag arrays. This is an important property that must hold in order for the `update_cost` operation to perform correctly alongside BFS.

When verifying this approach with VerCors, several things go wrong. First of all lines 18-19 in `update_flag` fail. This means that there are cases where the normal BFS algorithm would set the flag for a vertex false, but the operation does not do this. Therefore the loop invariant that checks that the ghost variables for the flags matches the flag arrays also fails, and if the flags do not match, neither will the costs. A cause for this problem has not been found and further attempts to solve this have not been made due to time constraints.

Chapter 6

Related Work

There are many publications surrounding the programming and verification of GPGPU code for graph algorithms and in general.
Zhong et al [8] propose a general framework for performing graph algorithms on a GPU in order to achieve higher performance. Amongst others they use an Edge-Vertex model to solve the problems with mapping graph computations to hardware threads and lightweight approximate sorting to tackle load imbalance between threads.
Kojima et al [9] present an automated verification method for functional correctness of race free GPGPU programs in order to improve the efficiency of the verification process.
Che et al [10] have created a library of software building blocks for programming GPGPU graph applications to ease the workload of developers and improve portability and cross-platform performance. It is built on the OpenCL framework.
Dinneen et al [11] explore two different GPGPU implementation techniques for graph algorithms. They use these to implement a BFS algorithm and compare the performance on large graphs to other implementations, including Harish's algorithm.
Xu et al [12] analysed the performance of graph algorithms for GPGPU, including BFS, and compared it to non-graph programs to determine the bottlenecks on the hardware level. The results indicate that graph algorithms use more kernel function calls, leading to overuse of the PCI connection between GPU and CPU and ineffective use of caches.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In order to perform algorithms on large graphs, GPGPU programming is used. The highly parallel hardware of a GPU allows for many threads to perform an algorithm. However when implementing such an algorithm special care must be taken that it is done efficiently and does not contain data races. The functional correctness must be proven afterwards to ensure the results produced are correct.
The goal of this project was to formally verify data race freedom and functional correctness for Harish's algorithm, an implementation of breadth-first search for GPGPU published in [1]. In chapter 3 this algorithm was extensively described and two data race conditions were found in it. These were solved by introducing a comparison of costs and making the operation atomic. This modified algorithm was then used in the first attempt to formally verify data race freedom in chapter 4. The first attempt failed, possibly due to the complex adjacency list data structure used originally in Harish's algorithm. A new data structure was created so that there would be a thread running for every edge instead of vertex. This was verified to be data race free. Flags were also successfully added to this. For functional correctness first an attempt was made using paths. BFS should give the minimal number of edges to reach a vertex from the source, so if the shortest path was found this should match the BFS result. An algorithm was created to find the shortest path, but problems were encountered when trying to formally prove that the shortest path is found. The next attempt used ghost variables alongside the verified BFS algorithm. Operations were made that performed BFS in a backwards way, and these were used with the ghost variables. The resulting ghost variables were then compared with the results from BFS. Initially the differences between both approaches was too great so 2 flags were added to synchronise. However this also failed and due to time constraints no further attempts were made.
When it comes to efficiency the version of BFS with one thread for each edge and no flags is most efficient. Using one thread per edge means that the parallel nature of the GPU is better used and the total workload is more evenly divided over threads than when using a thread per vertex. While it seemingly increases efficiency, adding flags

to this could only complicate the algorithm more with read and write operations for these flags. Since the workload is already quite evenly divided the flags do not increase efficiency.

7.2 Future Work

Since the verification of functional correctness has not been successful, finishing this would be the first priority. There are several options to complete this. First the attempt to do this using the operations and BFS algorithm using two flags can be completed. More lemmas are needed to prove that the reverse strategy equals the normal BFS. A first step in this direction has already been made with line 16-17 in `update_flag`. These lines describe a postcondition that states that the result of the reverse strategy is equal to the result when using the normal BFS strategy. This was verified using VerCors and creates a first bridge in the gap between normal and reverse algorithms. Additionally the definitions already created here can be used.
Secondly another approach to verify functional correctness could be to add postconditions to the data race free implementation with no flags from chapter 4. These postconditions must then captivate the desired functional outcome of the algorithm and be verified by VerCors.
Alternatively another data structure can be used. Instead of two arrays with a length equal to the number of edges, a singe variable contains all data. This would be a sequence of arrays. Every vertex number would correspond to an index in this sequence, where an array of vertex numbers is stored. The array lists all the neighbours of the concerning vertex. This way the length of the sequence would be equal to `V`, whilst integer `E` becomes obsolete, as edges are not explicitly stored here. An example like figure 3.1 and 4.1 is given.

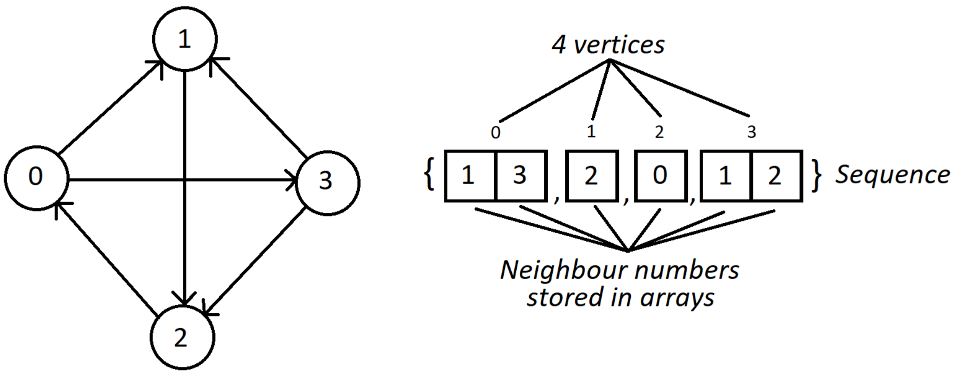

Figure 7.1: Example of the single variable adjacency list data structure

The vertex numbers in the arrays should be sorted, so that the desired properties of the graph, namely no self loops and no double edges, can easily be formulated into a precondition, much like the data structure of figure 3.1. Before trying to verify functional correctness using this it is necessary to first prove data race freedom. With this data structure there will be one thread running per vertex again. Since the outer structure is a sequence, only permissions for the arrays contained are necessary. A version with read and write flags is recommended if a similar approach as was done here with lemmas and ghost variables is used the definitions will also have to be adapted to the new data structure.

Bibliography

[1] Harish, Pawan, and P. J. Narayanan. "*Accelerating large graph algorithms on the GPU using CUDA.*" International conference on high-performance computing. Springer, Berlin, Heidelberg, 2007.

[2] Formal Methods and Tools (FMT), University of Twente. VerCors Verification Toolset. `https://github.com/utwente-fmt/vercors`

[3] Formal Methods and Tools (FMT), University of Twente. Prototypal Verification Language (PVL) Syntax. `https://github.com/utwente-fmt/vercors/wiki/PVL-Syntax`

[4] Rafia Inam, *An Introduction to GPGPU Programming - CUDA Architecture* Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden

[5] Aziz, Adnan; Prakash, Amit (2010). "4. Algorithms on Graphs". *Algorithms for Interviews.* p. 144. ISBN 978-1453792995.

[6] Blom, Stefan, Marieke Huisman, and Mihelcic Matej. 2014. "*Specification and Verification of GPGPU Programs.*" Science of Computer Programming 95: 37688.

[7] Kim, Hyesoon. 2012. "*Performance Analysis and Tuning for General Purpose Graphics Processing Units (Gpgpu)*" Synthesis Lectures on Computer Architecture, #20. San Rafael, Calif.: Morgan & Claypool.

[8] Zhong W, Sun J, Chen H, Xiao J, Chen Z, Cheng C, and Shi X. 2017. "*Optimizing Graph Processing on GPUs*" Ieee Transactions on Parallel and Distributed Systems 28 (4): 114962.

[9] Kojima, Kensuke, Akifumi Imanishi, and Atsushi Igarashi. 2018. "*Automated Verification of Functional Correctness of Race-Free Gpu Programs.*" Journal of Automated Reasoning 60 (3): 27998.

[10] Che, Shuai, Bradford M Beckmann, and Steven K Reinhardt. 2017. "*Programming GPGPU Graph Applications with Linear Algebra Building Blocks.*" International Journal of Parallel Programming 45 (3): 65779.

[11] Michael J. Dinneen, Masoud Khosravani and Andrew Probert. 2011. "*Using OpenCL for Implementing Simple Parallel Graph Algorithms*" Department of Computer Science, University of Auckland, Auckland, New Zealand

[12] Qiumin Xu, Hyeran Jeon and Murali Annavaram. 2014. "*Graph Processing on GPUs: Where are the Bottlenecks?*". Ming Hsieh Department of Electrical Engineering, University of Southern California Los Angeles, CA