Automated Verification of the Parallel Bellman–Ford Algorithm

Mohsen Safari[1], Wytse Oortwijn[2], and Marieke Huisman[1(✉)]

[1] Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{m.safari,m.huisman}@utwente.nl
[2] ESI (TNO), Eindhoven, The Netherlands
wytse.oortwijn@tno.nl

Abstract. Many real-world problems such as internet routing are actually graph problems. To develop efficient solutions to such problems, more and more parallel graph algorithms are proposed. This paper discusses the mechanized verification of a commonly used parallel graph algorithm, namely the Bellman–Ford algorithm, which provides an inherently parallel solution to the Single-Source Shortest Path problem.

Concretely, we verify an unoptimized GPU version of the Bellman–Ford algorithm, using the VerCors verifier. The main challenge that we had to address was to find suitable global invariants of the graph-based properties for automated verification. This case study is the first deductive verification to prove functional correctness of the parallel Bellman–Ford algorithm. It provides the basis to verify other, optimized implementations of the algorithm. Moreover, it may also provide a good starting point to verify other parallel graph-based algorithms.

Keywords: Deductive verification · Graph algorithms · Parallel algorithms · GPU · Bellman–Ford · Case study

1 Introduction

Graph algorithms play an important role in computer science, as many real-world problems can be handled by defining suitable graph representations. This makes the correctness of such algorithms crucially important. As the real-world problems that we represent using graphs are growing exponentially in size—think for

The first and third author are supported by the NWO VICI 639.023.710 Mercedes project.

C. Drăgoi et al. (Eds.): SAS 2021, LNCS 12913, pp. 346–358, 2021.
https://doi.org/10.1007/978-3-030-88806-0_17

example about internet routing solutions—we need highly efficient, but still correct(!), graph algorithms. Massively parallel computing, as supported on GPUs for example, can help to obtain the required efficiency, but also introduces extra challenges to reason about the correctness of such parallel graph algorithms.

In the literature, several verification techniques to reason about the correctness of massively parallel algorithms have been proposed, see e.g. [5,7,11,25,26]. This paper uses such a verification technique to develop a mechanized proof of a parallel GPU-based graph algorithm. Our verification is based on deductive program verification, using a permission-based separation logic for GPU programs [7] as implemented in the VerCors program verifier [6]. In VerCors, the program to be verified is annotated with a specification, as well as intermediate (invariant) properties. From these annotations, suitable proof obligations are generated, which can then be discharged with Z3. Given the annotated program, the verification process is fully automatic.

The concrete graph algorithm that we study here is the Bellman–Ford algorithm [3,15], a solution for the Single-Source Shortest Path (SSSP) problem. This algorithm computes the shortest distance from a specific vertex to all other vertices in a graph, where the distance is measured in terms of arc weights. Other solutions exist for this problem, such as Dijkstra's shortest path algorithm [14]. However, the Bellman–Ford algorithm is inherently parallel, which makes it suitable to be used on massively parallel architectures, such as GPUs.

In this paper, we prove race freedom, memory safety and functional correctness of a standard parallel GPU-based Bellman–Ford algorithm. This correctness proof can be used as a starting point to also derive correctness of the various optimized implementations that have been proposed in the literature [1,9,18,20,32,34]. Moreover, this work and the experiences with automated reasoning about GPU-based graph algorithms will also provide a good starting point to verify other parallel GPU-based graph algorithms.

To the best of our knowledge, there is no similar work in the literature on the automated mechanized verification of parallel GPU-based graph algorithms—Bellman–Ford in particular. Previous works on graph algorithm verification either target sequential algorithms, or abstractions of concurrent non-GPU-based algorithms. Furthermore, most previous works on GPU program verification focus on proving memory safety/crash and race freedom, but not on functional correctness. In contrast, we prove functional correctness of the GPU-based Bellman–Ford algorithm.

The main challenge that we had to address in this work, was to find the suitable global invariants to reason about the graph-based algorithm, and in particular to make those amenable to mechanized verification. Therefore, we first outline the manual correctness proof, and then discuss how we formalized this proof in VerCors. As mentioned before, the Bellman–Ford algorithm is inherently parallel, and our proof indeed demonstrates this.

Organization. Section 2 discusses the Bellman–Ford algorithm, and also gives a brief introduction to the VerCors verifier. Section 3 discusses the manual proof of the algorithm, in particular introducing all the necessary invariants. Section 4

```
1  int[] Bellman–Ford(G, s) {
2    int[] cost = new int[|V|]; // declare cost array
3    cost[s] := 0; // initialize cost array
4    for every v ∈ V \ {s} do cost[v] := ∞;
5    for (i = 0 up to |V| − 1) { // start |V| − 1 rounds of cost relaxations
6      for every a ∈ A { // check every arc for potential cost relaxations
7        if (cost[src(a)] + w(a) < cost[dst(a)])
8          cost[dst(a)] := cost[src(a)] + w(a); // perform a cost relaxation
9      }
10   }
11   return cost;
12 }
```

Fig. 1. Pseudo-code implementation of the (sequential) Bellman–Ford algorithm.

continues with the formal proof by encoding the informal proof into the VerCors verifier. Section 5 explains the evaluation and lessons learned from this case study. Section 6 discusses related work and Sect. 7 concludes the paper.

2 Background

This section describes the (parallel) Bellman–Ford algorithm (Sect. 2.1), and gives a brief introduction on deductive code verification with VerCors (Sect. 2.2).

2.1 The Bellman–Ford Algorithm

A directed weighted graph $G = (V, A, w)$ is a triple consisting of a finite set V of vertices, an binary arc relation $A \subseteq V \times V$, and a weight function $w : A \to \mathbb{N}$ over arcs. In the remainder of this paper we assume that A is irreflexive, since we do not consider graphs that contain self-loops. The *source* and *destination* of any arc $a = (u, v) \in A$ is defined $\mathsf{src}(a) \triangleq u$ and $\mathsf{dst}(a) \triangleq v$, respectively. Any finite arc sequence $P = (a_0, a_1, \ldots, a_n) \in A^*$ is a *path in* G if $\mathsf{dst}(a_i) = \mathsf{src}(a_{i+1})$ for every $0 \leq i < n$. We say that P is an (u, v)*-path* if $\mathsf{src}(a_0) = u$ and $\mathsf{dst}(a_n) = v$, under the condition that $0 < n$. The *length of* P is denoted $|P|$ and defined to be $n+1$. The *weight* of any path P is denoted $w(P)$, with w overloaded and lifted to sequences of arcs $A^* \to \mathbb{N}$ as follows: $w(a_0, a_1, \ldots, a_n) \triangleq \Sigma_{i=0}^{n} w(a_i)$. Any path P is *simple* if all its vertices are unique. Finally, any (u, v)-path P is a *shortest* (u, v)*-path in* G if for every (u, v)-path Q in G it holds that $w(P) \leq w(Q)$.

The Bellman–Ford algorithm [3,15] solves the *Single-Source Shortest Path (SSSP) problem*: given any input graph $G = (V, A, w)$ and vertex $s \in V$, find for any vertex $v \in V$ reachable from s the weight of the shortest (s, v)-path. Figure 1 shows the algorithm in pseudo-code. It takes as input a graph G and starting vertex s. The idea is to associate a `cost` to every vertex v, which amounts to the weight of the shortest (s, v)-path that has been found up to round i of the algorithm. Initially, s has cost 0 (line 3), while all other vertices start with cost

∞ (line 4). Then the algorithm operates in $|V| - 1$ rounds (line 5). In every round i, the cost of vertex v is *relaxed* on lines 7–8 in case a cheaper possibility of reaching v is found. After $|V| - 1$ such rounds of relaxations, the weights of the shortest paths to all reachable vertices have been found, intuitively because no simple path can contain more than $|V| - 1$ arcs.

The Bellman–Ford algorithm can straightforwardly be parallelized on a GPU, by executing the iterations of the for-loop on line 6 in parallel, thereby exploiting that arcs can be iterated over in arbitrary order. Such a parallelization requires lines 7–8 to be executed atomically, and all threads to synchronize (by a possibly implicit barrier) between every iteration of the round loop.

This paper demonstrates how VerCors is used to mechanically verify *soundness* and *completeness* of the parallelized version of the Bellman–Ford algorithm. Soundness in this context means that, after completion of the algorithm, for any $v \in V$ such that $\texttt{cost}[v] < \infty$ it holds that there exists a shortest (s, v)-path P such that $\texttt{cost}[v] = w(P)$. The property of completeness is that, for any v if there exists an (s, v)-path P after completion of the algorithm, it holds that $\texttt{cost}[v] < \infty$. In addition to soundness and completeness we also use VerCors to verify memory safety and race-freedom of parallel Bellman–Ford.

2.2 The VerCors Verifier

VerCors [6] is an automated, SMT-based code verifier specialized in reasoning about parallel and concurrent software. VerCors takes programs as input that are annotated with logical specifications, and can automatically verify whether the code implementation adheres to these specifications. The specifications are formulated in a Concurrent Separation Logic (CSL) that supports permission accounting, and are annotated as pre/postconditions for functions and threads, and invariants for loops and locks [2,7]. However, to keep the paper accessible, this paper describes the formalization independent of specific knowledge of CSL, and explains any further necessary details whenever needed.

3 Approach

Our strategy for verifying parallel Bellman–Ford is to first construct an informal pen-and-paper proof of its correctness, and then to encode this proof in VerCors to mechanically check all proof steps. This section elaborates on the (informal) correctness argument of Bellman–Ford, after which Sect. 4 explains how this argument is encoded in, and then confirmed by, VerCors.

Postconditions. Proving correctness of parallel Bellman–Ford amounts to proving that the following three postconditions hold after termination of the algorithm, when given as input a graph $G = (V, A, w)$ and starting vertex $s \in V$:

$$\forall v \,.\, \texttt{cost}[v] < \infty \implies \exists P \,.\, \mathsf{Path}(P, s, v) \land w(P) = \texttt{cost}[v] \tag{PC1}$$

$$\forall v \,.\, (\exists P \,.\, \mathsf{Path}(P, s, v)) \implies \texttt{cost}[v] < \infty \tag{PC2}$$

$$\forall v \,.\, \texttt{cost}[v] < \infty \implies \forall P \,.\, \mathsf{Path}(P, s, v) \implies \texttt{cost}[v] \leq w(P) \tag{PC3}$$

The predicate $\mathsf{Path}(P, u, v)$ expresses that P is an (u, v)-path in G.

These three postconditions together express that `cost` characterizes reachable states as well as shortest paths. PC1 and PC2 imply soundness and completeness of reachability: $\texttt{cost}[v] < \infty$ if and only if v is reachable from s. PC3 additionally ensures that `cost` contains the weights of all shortest paths in G.

Invariants. Our approach for proving the three postconditions above is to introduce *round invariants*: invariants for the loop on line 5 in Fig. 1 that should hold at the start and end of every round i for each thread. The proposed (round) invariants are:

$$\forall v \,.\, \texttt{cost}[v] < \infty \implies \exists P \,.\, \mathsf{Path}(P, s, v) \wedge w(P) = \texttt{cost}[v] \qquad \text{(INV1)}$$

$$\forall v \,.\, (\exists P \,.\, \mathsf{Path}(P, s, v) \wedge |P| \leq i) \implies \texttt{cost}[v] < \infty \qquad \text{(INV2)}$$

$$\forall v \,.\, \texttt{cost}[v] < \infty \implies \forall P \,.\, \mathsf{Path}(P, s, v) \wedge |P| \leq i \implies \texttt{cost}[v] \leq w(P) \qquad \text{(INV3)}$$

One can prove that the round invariants imply the postconditions after termination of the round loop, as then $i = |V| - 1$. PC1 immediately follows from INV1 without additional proof. Proving that PC2 and PC3 follow from INV2 and INV3 resp. requires more work since these postconditions quantify over paths of unbounded length.

Therefore, we introduce an operation $\mathsf{simple}(P)$ that removes all cycles from any given (u, v)-path P, and gives a simple (u, v)-path, which makes it easy to establish the postconditions. The three main properties of `simple` that are needed for proving the postconditions are $|\mathsf{simple}(P)| \leq |V| - 1$, $|\mathsf{simple}(P)| \leq |P|$, and $w(\mathsf{simple}(P)) \leq w(P)$ for any P. The latter two hold since $\mathsf{simple}(P)$ can only shorten P. Here we detail the proof for PC3; the proof for PC2 is similar.

Lemma 1. *If $i = |V| - 1$ then* INV3 *implies* PC3.

Proof. Let v be an arbitrary vertex such that $\texttt{cost}[v] < \infty$, and P be an arbitrary (s, v)-path. Then $\texttt{cost}[v] \leq w(P)$ is shown by instantiating INV3 with v and $\mathsf{simple}(P)$, from which one can easily prove $\texttt{cost}[v] \leq w(\mathsf{simple}(P)) \leq w(P)$. □

Preservation of Invariants. However, proving that each round of the algorithm preserves the round invariants is significantly more challenging. It is non-trivial to show that validity of invariants INV1–INV3 at round $i + 1$ follows from their validity at round i combined with the contributions of all threads in round i. An additional difficulty is that cost relaxations are performed in arbitrary order.

Our approach was to first work out the proof details in pen-and-paper style, and to later encode all proof steps in VerCors. We highlight one interesting case:

Lemma 2. *Every iteration of the loop on lines 5–10 in Fig. 1 preserves* INV3.

Proof (outline). Suppose that INV1–INV3 *hold on round i, that $i < |V| - 1$, and that all cost relaxations have happened for round i (i.e., lines 6–9 have been fully executed). We show that* INV3 *holds for $i + 1$. We write $\mathsf{old}(\texttt{cost}[v])$ to refer to the "old" cost that any v had at the beginning of round i.*

We create a proof by contradiction. Suppose that there exists a vertex v and an (s, v)-path P such that $\texttt{cost}[v] < \infty$, $|P| \leq i + 1$, *and* $w(P) < \texttt{cost}[v]$. *It must be the case that* $|P| = i + 1$, *since otherwise, if* $|P| < i + 1$, *then* INV1 *and* INV2 *together would imply that* $\mathsf{old}(\texttt{cost}[v]) < w(P)$, *which is impossible since vertex costs can only decrease. So P consists of at least one arc. Let a be the last arc on P so that* $\mathsf{dst}(a) = v$, *and let P' be the path P but without a, so that P' is an* $(s, \mathsf{src}(a))$*-path of length i. Let us abbreviate* $\mathsf{src}(a)$ *as v'. Instantiating* INV2 *and* INV3 *with v' and P' gives* $\mathsf{old}(\texttt{cost}[v']) < \infty$ *and* $\mathsf{old}(\texttt{cost}[v']) < w(P')$.

Let us now consider what a's thread could have done in round i. When this thread got scheduled it must have observed that v' and v had some intermediate costs, which we refer to as $obs_{v'}$ *and* obs_v, *respectively, for which it holds that* $\texttt{cost}[v'] \leq obs_{v'} \leq \mathsf{old}(\texttt{cost}[v'])$ *and* $\texttt{cost}[v] \leq obs_v \leq \mathsf{old}(\texttt{cost}[v])$. *And since*

$$obs_{v'} + w(a) \leq \mathsf{old}(\texttt{cost}[v']) + w(a) \leq w(P') + w(a) = w(P) < \texttt{cost}[v] \leq obs_v$$

we know that a's thread must have updated the cost of v to be $obs_{v'} + w(a)$ *in its turn. Since v's cost might have decreased further in round i by other threads, we have* $\texttt{cost}[v] \leq obs_{v'} + w(a) \leq w(P)$, *which contradicts* $w(P) < \texttt{cost}[v]$. □

This proof outline emphasizes the non-triviality of verifying the Bellman–Ford algorithm using automated code verifiers. Interestingly, also all other invariant preservation proofs have been performed as a proof by contradiction.

4 Proof Mechanization

So far the correctness argument of parallel Bellman–Ford has been presented at the abstract level of mathematical definitions and pseudocode. This section discusses how this abstract reasoning translates to the GPU version of Bellman–Ford, by formalizing its correctness proof in VerCors. This required (i) encoding all specifications introduced in Sect. 3 into the VerCors specification language, (ii) adding additional permission specifications to guarantee memory safety, and (iii) using these specifications to formulate pre- and postconditions, as well as loop and lock invariants for the algorithm encoding. For step (i), the main challenge was to give these specifications in terms of concrete GPU data types, rather than mathematical structures (e.g., defining a graph representation in C arrays instead of mathematical sets). Furthermore, GPU memory (as a scarce resource) imposes more restrictions on how to represent large graphs in an efficient way (e.g., using a one-dimensional array instead of matrices, and assigning threads to arcs instead of vertices). For step (iii), the main challenge was to encode the lemmas and their proofs, as introduced in Sect. 3. We use *lemma functions* for this, which are pure functions whose function specification corresponds to the lemma property. The challenge was to encode the proofs of these lemmas in VerCors, as discussed in more detail below. The end result of our verification effort is the first machine-checked proof of a GPU version of parallel Bellman-Ford. The remainder of this section elaborates on the formalization of the informal specifications and proof outlines in VerCors, and on how these are used to verify the concrete GPU host and kernel code.

```
1  kernel_invariant \pointer((src, dst, w), A, read);
2  kernel_invariant \pointer(cost, V, write);
3  kernel_invariant ···; // encodings of INV1, INV2 and INV3
4  void Kernel-BF(int V, int A, int[] src, int[] dst, int[] w, int[] cost, int s) {
5    int tid = blockIdx.x × 512 + threadIdx.x;
6    atomicMin(&cost[dst[tid]], cost[src[tid]] + w[tid]);
7  }
8  context Graph(V, A, src, dst, w);
9  context \pointer((src, dst, w), A, read) ∧ \pointer(cost, V, write);
10 ensures ∀v . 0 ≤ v < V ∧ cost[v] < infty() ⇒ // encoding of PC1
11   ∃P . Path(V, A, src, dst, w, s, v, P) ∧ Weight(V, A, src, dst, w, P) = cost[v];
12 ensures ∀v . 0 ≤ v < V ⇒ // encoding of PC2
13   ∃P . Path(V, A, src, dst, w, s, v, P) ⇒ cost[v] < infty();
14 ensures ∀v . 0 ≤ v < V ∧ cost[v] < infty() ⇒ // encoding of PC3
15   ∀P . Path(V, A, src, dst, w, s, v, P) ⇒ cost[v] ≤ Weight(V, A, src, dst, w, P);
16 void Host-BF(int V, int A, int[] src, int[] dst, int[] w, int[] cost, int s) {
17   invariant Graph(V, A, src, dst, w);
18   invariant
     \pointer((src, dst, w), A, read) ∧ \pointer(cost, V, write);
19   invariant ∀v . 0 ≤ v < V ∧ cost[v] < infty() ⇒ // encoding of INV1
20     ∃P . Path(V, A, src, dst, w, s, v, P) ∧ Weight(V, A, src, dst, w, P) = cost[v];
21   invariant ∀v . (0 ≤ v < V ∧ ∃P . Path(V, A, src, dst, w, s, v, P) ∧ |P| ≤ i) ⇒
22     cost[v] < infty(); // encoding of INV2
23   invariant ∀v . 0 ≤ v < V ∧ cost[v] < infty() ⇒
24     ∀P . Path(V, A, src, dst, w, s, v, P) ∧ |P| ≤ i ⇒
25       cost[v] ≤ Weight(V, A, src, dst, w, P); // encoding of INV3
26   for (i = 0 up to V − 1) {
27     Kernel-BF(V, A, src, dst, w, cost, s);
28     lemma_inv_pres(...); // apply invariant preservation lemma functions
29   }
30   lemma_post_establ(...); // apply postcond. establishment lemma functions
31 }
```

Fig. 2. The simplified GPU version of Bellman-Ford, annotated with VerCors specifications. The total number of threads (*tid*) is the same as the number of arcs (A).

Proof Outline and Specification Encoding. Figure 2 presents a simplified[1] overview of our specification of parallel Bellman–Ford. Lines 1–7 and lines 8–31 show the annotated CPU host code and GPU kernel code, respectively. Observe that the algorithm uses a representation of directed weighted graphs that is typical for GPU implementations: using three C arrays, `src`, `dst` and `w`.

On line 8 in the specification we require (and ensure)[2] that these three arrays indeed form a graph, by means of the predicate `Graph(V, A, src, dst, w)`,

[1] Various details have been omitted for presentational clarity. We highlight only the most interesting aspects of the specification. The full specification is available at [31].

[2] The keyword `context` is an abbreviation for both `requires` and `ensures`.

```
1 pure bool Graph(int V, int A, int[] src, int[] dst, int[] w) =
2   0 < V ∧ 0 < A ∧ |src| = A ∧ |dst| = A ∧ |w| = A ∧
3   (∀i . 0 ≤ i < A ⇒ 0 ≤ src[i] < V ∧ 0 ≤ dst[i] < V ∧ src[i] ≠ dst[i] ∧ 0 < w[i]) ∧
4   (∀i, j . 0 ≤ i < A ∧ 0 ≤ j < A ∧ i ≠ j ∧ src[i] = src[j] ⇒ dst[i] ≠ dst[j]);
```

Fig. 3. The `Graph` predicate, that determines whether `src`, `dst` and `w` form a graph.

as defined in Fig. 3. The integer `V` represents the total number of vertices, and `A` the total number of arcs. Then any index $a \in [0, \texttt{A})$ represents an arc from `src`$[a]$ to `dst`$[a]$ with weight `w`$[a]$. Similarly, any index $v \in [0, \texttt{V})$ represents a vertex in the graph such that `cost`$[v]$ is the current cost assigned to v by the algorithm. The integer `s`, with $0 \leq \texttt{s} < \texttt{V}$, is the starting vertex. This representation can handle large graphs on GPU memory, and by assigning threads to arcs more parallelism and hence more performance can be obtained.

Lines 10–15 and 19–25 contain the VerCors encoding of the postconditions and round invariants introduced in Sect. 3, respectively. These encodings are defined over various other predicates such as `Path` and `Weight`, whose definitions are the same in spirit as the one of `Graph`.

Verifying Memory Safety. Verifying data-race freedom requires explicitly specifying ownership over heap locations using *fractional permissions*, in the style of Boyland [8]. Fractional permissions capture which heap locations may be accessed by which threads. We use the predicate $\texttt{\textbackslash pointer}((S_0, \ldots, S_n), \ell, \pi)$ to indicate that all array references $S_0, \ldots, S_n$ have length ℓ, and that the current thread has permission $\pi \in (0, 1]$ for them[3]. We often use the keywords `read` and `write` instead of concrete fractional values to indicate read or write access.

Lines 9 and 18 indicate that initially and in each iteration of the algorithm we have read permission over all locations in `src`, `dst` and `w`. Moreover, we also have write permission over all locations in `cost`. Within the kernel, threads execute in parallel, meaning that the updates to `cost` have to be done atomically (line 6)[4]. The kernel invariants specify shared resources and properties that may be used by a thread while in the critical section. After leaving the critical section, the thread should ensure all the kernel invariants are re-established (see [2] for more details).

The kernel invariants on lines 1 and 2 specify that each thread within the critical section has read permission over all locations in `src`, `dst` and `w` (line 1) and write permission in `cost` (line 2). Note that the atomic operations execute in an arbitrary order, but as there always is at most one thread within the critical section, this is sufficient to guarantee data-race freedom.

Lemma Functions. As mentioned above, to show the preservation of INV1, INV2 and INV3 we apply the corresponding lemmas at the end of the loop (line 28).

[3] To specify permissions over a specific location *idx* of an array S we use $\texttt{\textbackslash pointer\_index}(S, idx, \pi)$, where *idx* is a proper index in S.

[4] atomicMin() is a built-in GPU function that compares its two arguments and assigns the minimum one to the first argument.

```
1  requires Graph(...) ∧ 0 ≤ i < V − 1 ∧ |P| ≤ i + 1;
2  requires cost[v] ≤ oldcost[v] < infty();
3  requires Weight(V, A, src, dst, w, P) < cost[v];
4  requires ∀a . 0 ≤ a < A ∧ oldcost[src[a]]+w[a] < oldcost[dst[a]] ⇒
5    cost[dst[a]] ≤ oldcost[src[a]] + w[a];
6  requires ∀v . 0 ≤ v < V ∧ cost[v] < infty() ⇒
7    ∃P . Path(V, A, src, dst, w, s, v, P)∧
8      Weight(V, A, src, dst, w, P) = oldcost[v]; // encoding of INV1
9  requires ∀v . (0 ≤ v < V ∧ ∃P . Path(V, A, src, dst, w, s, v, P) ∧ |P| ≤ i) ⇒
10   oldcost[v] < infty(); // encoding of INV2
11 requires ∀v . 0 ≤ v < V ∧ oldcost[v] < infty() ⇒
12   ∀P . Path(V, A, src, dst, w, s, v, P) ∧ |P| ≤ i ⇒
13     oldcost[v] ≤ Weight(V, A, src, dst, w, P); // encoding of INV3
14 ensures false;
15 pure bool lemma_2(int i, int v, int[] P, ...) {
16   assert 0 < |P| ∧ |P| = i + 1; a := P[|P| − 1]; P' := P[0..|P| − 2];
17   assert |P'| ≤ i; v' := src(a);
18   assert oldcost[v'] ≤ Weight(V, A, src, dst, w, P'); // from INV2 and INV3
19   assert cost[v'] ≤ oldcost[v'] ∧ cost[v'] ≤ Weight(V, A, src, dst, w, P');
20   assert lemma-transitivity(V, A, src, dst, w, s, v', v, P', a);
21   assert cost[v] ≤ Weight(V, A, src, dst, w, P); // contradiction
22 }
```

Fig. 4. The (simplified) VerCors encoding of Lemma 2.

Note that these invariants must hold in the kernel as well (line 3). Similarly, to establish PC1, PC2 and PC3 we apply the corresponding lemmas after termination of the loop when $i = |V| - 1$ (line 30).

All the proofs of the lemmas mentioned in Sect. 3 that show the preservation of the round invariants and establishment of postconditions are encoded in VerCors as *lemma functions* [16,35]. Lemma functions have specifications that capture the desired property, while the proof is encoded as a side effect-free imperative program. Most of our lemmas (e.g., the proof of Lemma 2) were proven by contradiction. Proving a property ϕ by contradiction amounts to proving $\neg\phi \Rightarrow$ false. Therefore, to show preservation of, e.g., INV3 (Lemma 2), we proved that (INV1$(i)\wedge$ INV2$(i)\wedge$ INV3$(i) \wedge \phi(i) \wedge \neg$ INV3$(i+1)) \Rightarrow$ false, with $\phi(i)$ describing the contributions of all threads in round i.

Lemma 4 shows how the VerCors encoding of Lemma 2 looks, where the lemma is implicitly quantified over the function parameters: iteration round i, vertex v, and path P. The function body encodes all proof steps. The main challenge was finding the precise assertions that explicitly describe all the steps from the informal proof. In particular, we had to prove various auxiliary lemmas such as `lemma-transitivity` (line 20), which models the transitivity property of paths along with its weight, and which required an induction over paths in its proof.

5 Evaluation and Discussion

Evaluation. The algorithm encoding and its specification consists of 541 lines of code. Of these 541 lines, 30 are for the encoding of the algorithm (5.5%) and the remaining 511 are specification (94.5%). The specification part can be subdivided further: of the 511 lines, 6.1% is related to permissions, 30.7% to invariant preservation proofs, 45.1% to proofs for establishing the postconditions, and 18.1% to definitions (e.g., of graphs and paths) and proving basic properties.

The total verification effort was about six weeks. Most of this time was spent on the mechanization aspects: spelling out all the details that were left implicit in the the pen-and-paper proof. The fully annotated Bellman–Ford implementation takes about 12 min to verify using VerCors on a Macbook Pro (early 2017) with 16 GB RAM, and an Intel Core i5 3.1 GHz CPU.

Discussion. In order to understand what verification techniques are suitable and effective for verifying parallel algorithms, we need the experience from different non-trivial case studies such as the one in this paper. Therefore, the value of this case study is more than just the verification of Bellman–Ford.

This case study confirms the importance of lemma functions in verifying non-trivial case studies, and in particular for encoding proofs by contradiction, which are common in the context of graphs. This paper also gives a representation of graphs that is suitable for GPU architectures, and can form the foundation of other verifications. Finally, we learned that deductive code verifiers are powerful enough to reason about non-trivial parallel algorithms—but they cannot do this yet without the human expertize to guide the prover.

6 Related Work

The work that is closest to ours is by Wimmer et al. [39], who prove correctness of a sequential version of the Bellman–Ford algorithm using Isabelle. Their proof strategy is different from ours: they use a framework from Kleinberg and Tardos [21] to refine a correct recursive function into an efficient imperative implementation. They first define Bellman–Ford as a recursive function that computes the shortest distances between all vertices using dynamic programming, and then use Isabelle to prove that it returns the shortest path. Then this recursive function is refined into an efficient imperative implementation (see the proof in [36]). However, this imperative implementation cannot be naturally parallelized. Moreover, because of the refinement approach, their correctness arguments are different from ours and do not depend on property preservation, which makes them unsuitable for standard deductive code verification.

In the literature there is ample work on the verification of other sequential graph algorithms. Some of these verifications are fully automatic, while others are semi-automatically done by interactive provers. Lammich et al. [23,24] propose a framework for verifying sequential DFS in Isabelle [19]. Chen et al. [10] provide a formal correctness proof of Tarjan's sequential SCC algorithm using

three (both automated and interactive) proof systems: Why3 [38], Coq [12] and Isabelle. There is also a collection of verified sequential graph algorithms in Why3 [37]. Van de Pol [29] verified the sequential Nested DFS algorithm in Dafny [13]. Guéneau et al. [17] improved Bender et al.'s [4] incremental cycle detection algorithm to turn it into an online algorithm. They implemented it in OCaml and proved its functional correctness and worst-case amortized asymptotic complexity (using Separation logic combined with Time Credits).

In contrast, there is only limited work on the verification of concurrent graph algorithms. Raad et al. [30] verified four concurrent graph algorithms using a logic without abstraction (CoLoSL), but their proofs have not been automated. Sergey et al. [33] verified a concurrent spanning tree algorithm using Coq.

As far as we are aware, there is no work on automated code verification of massively parallel GPU-based graph algorithms. Most similar to our approach is the work by Oortwijn et al. [27,28], who discuss the automated verification of the parallel Nested Depth First Search (NDFS) algorithm of Laarman et al. [22]. Although they are the first to provide a mechanical proof of a parallel graph algorithm, their target is not massively parallel programs on GPUs.

7 Conclusion

Graph algorithms play an important role in solving many real-world problems. This paper shows how to mechanically prove correctness of the parallel Bellman–Ford GPU algorithm, with VerCors. To the best of our knowledge, this is the first work on automatic code verification of this algorithm.

Since we prove the general classic Bellman–Ford algorithm without applying GPU optimization techniques, we plan to investigate how to reuse the current proof for the optimized implementations. Moreover, we also would like to investigate how we can generate part of the annotations automatically.

References

1. Agarwal, P., Dutta, M.: New approach of Bellman Ford algorithm on GPU using compute unified design architecture (CUDA). Int. J. Comput. Appl. **110**(13) (2015)
2. Amighi, A., Darabi, S., Blom, S., Huisman, M.: Specification and verification of atomic operations in GPGPU programs. In: SEFM, vol. 9276 (2015)
3. Bellman, R.: On a routing problem. Q. Appl. Math. **16**, 87–90 (1958)
4. Bender, M.A., Fineman, J.T., Gilbert, S., Tarjan, R.E.: A new approach to incremental cycle detection and related problems. ACM Trans. Algorith. (TALG) **12**(2), 1–22 (2015)
5. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: OOPSLA, pp. 113–132. ACM (2012)
6. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 102–110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7

7. Blom, S., Huisman, M., Mihelčić, M.: Specification and verification of GPGPU programs. Sci. Comput. Prog. **95**, 376–388 (2014)
8. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_4
9. Busato, F., Bombieri, N.: An efficient implementation of the Bellman-Ford algorithm for Kepler GPU architectures. IEEE Trans. Parall. Distrib. Syst. **27**(8), 2222–2233 (2016)
10. Chen, R., Cohen, C., Lévy, J.J., Merz, S., Théry, L.: Formal proofs of Tarjan's algorithm in Why3, Coq, and Isabelle. arXiv preprint arXiv:1810.11979 (2018)
11. Collingbourne, P., Cadar, C., Kelly, P.H.J.: Symbolic testing of openCL code. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 203–218. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34188-5_18
12. The Coq proof assistant. https://coq.inria.fr/
13. Dafny program verifier, https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/
14. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische mathematik **1**(1), 269–271 (1959)
15. Ford, L.R., Jr.: Network flow theory. Tech. rep, DTIC Document (1956)
16. Grov, G., Tumas, V.: Tactics for the Dafny program verifier. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 36–53. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_3
17. Guéneau, A., Jourdan, J.H., Charguéraud, A., Pottier, F.: Formal proof and analysis of an incremental cycle detection algorithm. In: Interactive Theorem Proving. No. 141, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
18. Hajela, G., Pandey, M.: Parallel implementations for solving shortest path problem using Bellman-Ford. Int. J. Comput. Appl. 95(15) (2014)
19. Isabelle interactive theorem prover. http://isabelle.in.tum.de/index.html
20. Jeong, I.K., Uddin, J., Kang, M., Kim, C.H., Kim, J.M.: Accelerating a Bellman-Ford routing algorithm using GPU. In: Frontier and Innovation in Future Computing and Communications, pp. 153–160. Springer (2014)
21. Kleinberg, J., Tardos, E.: Algorithm design. Pearson Education India, New Delh (2006)
22. Laarman, A., Langerak, R., van de Pol, J., Weber, M., Wijs, A.: Multi-core nested depth-first search. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 321–335. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24372-1_23
23. Lammich, P., Neumann, R.: A Framework for Verifying Depth-First Search Algorithms. In: CPP, pp. 137–146. ACM (2015)
24. Lammich, P., Wimmer, S.: IMP2-simple program verification in Isabelle/HOL. Archive of Formal Proofs (2019)
25. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: SIGSOFT FSE 2010, Santa Fe, pp. 187–196. ACM (2010)
26. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: ACM SIGPLAN Notices. vol. 47, pp. 215–224. ACM (2012)
27. Oortwijn, W.: Deductive techniques for model-based concurrency verification. Ph.D. thesis, University of Twente, Netherlands (2019). https://doi.org/10.3990/1.9789036548984

28. Oortwijn, W., Huisman, M., Joosten, S.J.C., van de Pol, J.: Automated verification of parallel nested DFS. In: TACAS 2020. LNCS, vol. 12078, pp. 247–265. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45190-5_14
29. van de Pol, J.C.: Automated verification of nested DFS. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 181–197. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19458-5_12
30. Raad, A., Hobor, A., Villard, J., Gardner, P.: Verifying concurrent graph algorithms. In: Igarashi, A. (ed.) APLAS 2016. LNCS, vol. 10017, pp. 314–334. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-47958-3_17
31. Safari, M., Oortwijn, W., Huisman, M.: Artifact for automated verification of the parallel bellman-ford algorithm. In: SAS (2021). https://github.com/Safari1991/SSSP-Verification
32. Safari, M., Ebnenasir, A.: Locality-based relaxation: an efficient method for GPU-based computation of shortest paths. In: Mousavi, M.R., Sgall, J. (eds.) TTCS 2017. LNCS, vol. 10608, pp. 43–58. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68953-1_5
33. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: PLDI, pp. 77–87 (2015)
34. Surve, G.G., Shah, M.A.: Parallel implementation of bellman-ford algorithm using CUDA architecture. In: 2017 International conference of Electronics, Communication and Aerospace Technology (ICECA), vol. 2, pp. 16–22. IEEE (2017)
35. Volkov, G., Mandrykin, M., Efremov, D.: Lemma functions for Frama-c: C programs as proofs. In: 2018 Ivannikov Ispras Open Conference (ISPRAS), pp. 31–38. IEEE (2018)
36. A Theory of Bellman-Ford, in Isabelle. https://www.isa-afp.org/browser_info/current/AFP/Monad_Memo_DP/Bellman_Ford.html. Accessed Jan 2021
37. Why3 gallery of formally verified programs. http://toccata.lri.fr/gallery/graph.en.html
38. Why3 program verifier. http://why3.lri.fr/
39. Wimmer, S., Hu, S., Nipkow, T.: Verified memoization and dynamic programming. In: Avigad, J., Mahboubi, A. (eds.) ITP 2018. LNCS, vol. 10895, pp. 579–596. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94821-8_34