Mostly Automated Formal Verification of Loop Dependencies with Applications to Distributed Stencil Algorithms

Thomas Grégoire[1] and Adam Chlipala[2]

[1] ÉNS Lyon, France
thomas.gregoire@ens-lyon.fr
[2] MIT CSAIL, Cambridge, MA, USA
adamc@csail.mit.edu

Abstract. The class of *stencil* programs involves repeatedly updating elements of arrays according to fixed patterns, referred to as stencils. Stencil problems are ubiquitous in scientific computing and are used as an ingredient to solve more involved problems. Their high regularity allows massive parallelization. Two important challenges in designing such algorithms are cache efficiency and minimizing the number of communication steps between nodes. In this paper, we introduce a mathematical framework for a crucial aspect of formal verification of both sequential and distributed stencil algorithms, and we describe its Coq implementation. We present a domain-specific embedded programming language with support for automating the most tedious steps of proofs that nested loops respect dependencies, applicable to sequential and distributed examples. Finally, we evaluate the robustness of our library by proving the dependency-correctness of some real-world stencil algorithms, including a state-of-the-art cache-oblivious sequential algorithm, as well as two optimized distributed kernels.

1 Introduction

Broadly speaking, in this paper we are interested in verifying, within a proof assistant, the correctness of a class of algorithms in which some matrices are computed, with some matrix cells depending on others. The aim is to check that these quantities are computed *in the right order*. This archetypical style of calculation arises in such situations as computing solutions of partial differential equations using finite-difference methods. The algorithms used in this setting are usually referred to as *stencil codes*, and they are the focus of the framework that we present in this paper.

A *stencil* defines a value for each element of a d-dimensional spatial grid at time t as a function of neighboring elements at times $t-1, t-2, \ldots, t-k$, for some fixed $k, d \in \mathbb{N}^+$. Figure 1 defines a stencil and gives its graphical representation.

Stencil problems naturally occur in scientific-computing and engineering applications. For example, consider the two-dimensional heat equation:

$$\frac{\partial u}{\partial t} - \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = 0$$

$$\begin{cases} \text{Space} = \{0, 1, \ldots, N\}^2, \\ u_0[x, y] = \alpha_{x,y}, \\ u_{t+1}[x, y] = F(u_t[x, y], u_t[x+1, y], u_t[x-1, y], \\ \qquad\qquad u_t[x, y+1], u_t[x, y-1]), \\ N \in \mathbb{N}, \quad \alpha_{x,y} \in \mathbb{R}, \quad F : \mathbb{R}^4 \to \mathbb{R} \end{cases}$$

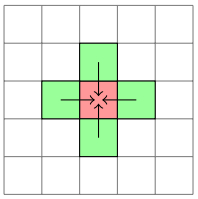

Fig. 1. A two-dimensional Jacobi stencil

We might try to solve it by discretizing both space and time using a finite-difference approximation scheme as follows:

$$\frac{\partial u}{\partial t} \approx \frac{u(t + \Delta t, x, y) - u(t, x, y)}{\Delta t}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(t, x + \Delta x, y) - 2u(t, x, y) + u(t, x - \Delta x, y)}{\Delta x^2}$$

Proceeding similarly for $\frac{\partial^2 u}{\partial y^2}$, we obtain the two-dimensional stencil depicted by Figure 1. More generally, stencil computations are used when solving partial differential equations using finite-difference methods on regular grids [11,4], in iterative methods solving linear systems [11], but also in simulations of cellular automata, such as Conway's game of life [8].

In this paper, we will focus on stencil codes where all the values of the grid are required at the end of the computation–a common situation when partial differential equations are used to simulate the behavior of a real-world system, and the user is interested not only in the final result but also in the dynamics of the underlying process.

Although writing stencil code might seem very simple at first glance–a program with nested loops that respects the dependencies of the problem is enough–there are in fact many different optimizations that have a huge impact on performance, especially when the grid size grows. For stencil code running on a single processor or core, changing the order in which computations are performed can significantly increase cache efficiency, hence dramatically lowering computation times. In the case of multicore implementations, reducing the number of synchronizations between the cores is important since communication is usually the main bottleneck. All these optimizations *reorder* computations. It is therefore crucial to check that these reorderings *do not break the dependencies implied by the underlying stencil*. This is the overall goal of this paper.

We describe a formal framework to define stencils, encode their sequential and distributed implementations, and prove their correctness. We show how using a domain-specific language adapted to stencil code allows a very effective form of *symbolic execution* of programs, supporting verification without annotations like loop invariants that are common in traditional approaches. As for distributed algorithms, we show how the *synchronousness* of stencil kernels impacts verification. In particular, we show how verification of distributed stencil code boils down to verification of *sequential* algorithms. Finally, we showcase the robustness

of our theory by applying it to some real-world examples, including a state-of-the-art sequential cache-oblivious algorithm, as well as a communication-efficient distributed kernel.

We have implemented our framework and example verifications within the Coq proof assistant[3]. Because everything is formalized in Coq, we will, in this paper, tend toward relatively informal explanations, to help develop the reader's intuition.

So, to summarize, our contributions are *the first mechanized proof of soundness of a dependency-verification framework for loopy programs over multidimensional arrays*, in addition to *a set of Coq tactics that support use of the framework with reasonable effort* plus a set of case-study verifications showing the framework in action for both sequential and distributed programs.

1.1 Related Work

To the authors' knowledge, this is the first attempt at designing a verification framework for dependencies in stencil code.

Stencils have drawn some attention in the formal-proof community, since finite-difference schemes are among the simplest (yet most powerful) methods available to solve differential equations in low dimension. Therefore, recent work has been more focused on proving stability and convergence of a given discretization scheme [2,3,9], rather than investigating different ways to solve a given stencil. In a different direction, we also mention Ypnos [16], a domain-specific language that enforces indexing safety guarantees in stencil code through type checking, therefore eliminating the need for run-time bounds checking.

As mentioned earlier in this introduction, stencil code can suffer from poor cache performance. This has led to intensive research on cache-oblivious stencil algorithms [6,7]. Writing such optimized stencil code can be very tedious and error-prone, and code might have to be rewritten from scratch when switching architecture. Recently, different techniques have been proposed to automate difficult parts of stencil implementation, including sketching [18], program synthesis [21], or compiling a domain-specific stencil language [19]. In our new work, we show how to verify *a posteriori* that such generated code respects dependencies, for infinite input domains.

One of the most natural approaches to generate efficient parallel stencil code is to use the *polyhedral model* (see [10,5]), to represent and manipulate loop nests. The goal of our framework is different from that of a vectorizing compiler, since we are not trying to generate code, but rather check that it does not violate any dependencies. Nevertheless, our approach is reminiscent of the line of work on the polyhedral model and parallelizing compilers (see, *e.g.*, [17,15,14]), in the sense that we produce an algebraic representation of the current state of the program and use it to check that dependencies are satisfied. Notice that, since we are working within a proof assistant, we can represent arbitrary mathematical

[3] `https://github.com/mit-plv/stencils`

sets, which implies that we are not limited to linear integer programs; and indeed our evaluation includes codes that employ nonlinear arithmetic.

Closely related to this paper, there has been recent work on the formal verification of GPU kernels. In this direction, we refer the reader to PUG [12], GKLEE [13], and GPUverify [1]. Our approach is somewhat orthogonal to this line of work: we study a smaller class of programs, which is also big enough to have many applications, supporting first-principles proofs at low human cost.

2 Verifying Stencil Code

2.1 Stencils and Their Representation

A *stencil* is defined on a set $\mathcal{G}$, which represents a spatial *grid*. Its elements are called *cells*, and most stencils of interest are based on $\mathcal{G} = \mathbb{Z}^d$ for some small integer d.

We assign a numerical value $u_t[c]$ to every cell $c \in \mathcal{G}$ at every time $t \in \mathbb{N}$ (or any time $0 \leq t \leq t_{\max}$, for some $t_{\max} \in \mathbb{N}$). A stencil is then defined by some initial conditions $u_0[c] = \alpha_c$, with $c \in \mathcal{G}$, $\alpha_c \in \mathbb{R}$, and a pattern

$$u_t[c] = F(u_{t-1}[d_1(c)], \ldots, u_{t-1}[d_k(c)]),$$

for some function F. The cells $d_1(c)$, $\ldots$, $d_k(c)$ are called c's *neighbors*. See Figure 1 for an example of such a formal stencil definition.

Let us turn to the representation of stencils within the proof assistant.

Definition 1. *A* stencil *is defined by:*

- *A type* `cell` *representing grid elements;*
- *A term* `space : set cell`*, indicating which finite subset of the grid we will be computing on;*
- *A term* `target : set cell`*, indicating which grid elements have to be computed by the end of the execution of the algorithm;*
- *A term* `dep : cell → list cell` *representing the dependencies of each cell, that is, its neighbors.*

Figure 2 shows how we formalize the Jacobi 2D stencil we introduced in Figure 1. We will conclude this section with a few comments on this definition:

Remark 1. Here, `set` A denotes "mathematical" sets of elements of type A, implemented in Coq as $A \to$ `Prop`.

Remark 2. Notice that this is an *abstract* notion of stencil. In particular, we do not specify the function applied at each step–it is seen as a black box–nor do we give the initial conditions. Moreover, the formulation is much more general than the schematic equation given above and encompasses *e.g.* Gauss-Seidel iterations or box-blur filtering.

```
Parameters T I J : Z.

Module Jacobi2D <: (PROBLEM Z3).
  Local Open Scope aexpr.

  Definition space := ⟦0, T⟧×⟦0, I⟧×⟦0, J⟧.
  Definition target := ⟦0, T⟧×⟦0, I⟧×⟦0, J⟧.
  Definition dep c :=
    match c with
      | (t,i,j) ⇒ [(t−1,i,j); (t−1,i−1,j); (t−1,i+1,j); (t−1,i,j−1); (t−1,i,j+1)]
    end.
End Jacobi2D.
```

Fig. 2. Coq representation of the two-dimensional Jacobi stencil

Remark 3. The `space` parameter is used to encode *boundary conditions*. For example, in the case of the Jacobi 2D stencil, we might want to ensure that $u_t[(i,j)] = 0$ as soon as $i < 0$, $i > I$, $j < 0$, or $j > J$, for some parameters I and J. In this case, we would pick[4] $\texttt{space} = [\![0, I]\!] \times [\![0, J]\!]$.

2.2 Programs: Syntax, Semantics, and Correctness

Now that we have a way to describe stencils, we turn to the representation and correctness of programs. Let us consider the following trivial program solving the Jacobi 2D stencil:

```
for t=0 to T do
  for i=0 to I do
    for j=0 to J do
      Compute u_t[i,j]
```

Verifying the correctness of this program amounts to proving that

1. It does not violate any dependency. That is, $u_t[i,j]$ is never computed before $u_{t-1}[i,j]$, $u_{t-1}[i+1,j]$, $u_{t-1}[i-1,j]$, $u_{t-1}[i,j+1]$, or $u_{t-1}[i,j-1]$;
2. It is complete, in the sense that it computes all the values of cells in `target`.

Therefore, we see programs as *agents*, having some *knowledge*. The state of a program is a *set of cells*, those with values known by the program.

For now, we will only discuss requirement 1, and we will see how to prove requirement 2 in the next section. Our starting point is a basic imperative language, which we extend with a **flag** c command, which adds cell c to the current state, as well as **assert** c, which checks that c belongs to the current state. If not, the program halts abnormally.

To verify requirement 1 for the trivial program above, we would therefore have to prove the normal termination of the following program:

[4] In this paper, for all $a, b \in \mathbb{Z}$, we write $[\![a, b]\!]$ for $\{n \in \mathbb{Z} : a \leq n \leq b\}$.

$$
\begin{array}{l}
p ::= \mathbf{nop} \mid p; p \mid \mathbf{flag}\ c \mid \mathbf{assert}\ c \mid \mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ p \mid \mathbf{for}\ v = e\ \mathbf{to}\ e\ \mathbf{do}\ p \\
e ::= k \mid v \mid e + e \mid e - e \mid e \times e \mid e / e \mid e \bmod e \\
b ::= \epsilon \mid \mathbf{not}\ b \mid b\ \mathbf{or}\ b \mid b\ \mathbf{and}\ b \mid e = e \mid e \neq e \mid e \leq e \mid e \geq e \mid e < e \mid e > e \\
k \in \mathbb{Z},\ \epsilon \in \{\top, \bot\}
\end{array}
$$

Fig. 3. Syntax of arithmetic and Boolean expressions and programs

```
for t=0 to T do
  for i=0 to I do
    for j=0 to J do
      assert (t − 1, i, j); assert (t − 1, i + 1, j);
      assert (t − 1, i − 1, j); assert (t − 1, i, j + 1);
      assert (t − 1, i, j − 1); flag (t, i, j)
```

The precise syntax of expressions and programs is given in Figure 3. Notice that there is no assignment command $x := e$, but that a statement like $x := e; p$ can be simulated by the program $\mathbf{for}\ x = e\ \mathbf{to}\ e\ \mathbf{do}\ p$. Our programs are *effect-free*, in the sense that variables are bound to values functionally within loops. Moreover, our framework is parameterized by a type and evaluation function for cell expressions.

The operational semantics of our programming language is given by a judgment $\rho \vdash (C_1, p) \Downarrow C_2$, where ρ assigns an integer value to every variable, C_1 and C_2 are sets of cells, and p is a program. The intended meaning is that, in a state where the program knows the values of the cells in C_1 (and these values only), and where variables are set according to ρ, the execution of p terminates (without any assertion failing) and the final knowledge of the program is described by C_2.

The semantics is given in Figure 4. There, $[\![e]\!]_\rho$ denotes the evaluation of expression e in environment ρ, when e is an arithmetic, Boolean, or cell expression. $\rho[x \leftarrow i]$ denotes the environment obtained by setting x to $i \in \mathbb{Z}$ in ρ.

$$
\text{Nop:}\frac{}{\rho \vdash (C, \mathbf{nop}) \Downarrow C} \qquad \text{Seq:}\frac{\rho \vdash (C_1, p_1) \Downarrow C_2 \quad \rho \vdash (C_2, p_2) \Downarrow C_3}{\rho \vdash (C_1, p_1; p_2) \Downarrow C_3}
$$

$$
\text{If (1):}\frac{[\![b]\!]_\rho = \top \quad \rho \vdash (C_1, p_1) \Downarrow C_2}{\rho \vdash (C_1, \mathbf{if}\ b\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2) \Downarrow C_2} \qquad \text{If (2):}\frac{[\![b]\!]_\rho = \bot \quad \rho \vdash (C_1, p_2) \Downarrow C_2}{\rho \vdash (C_1, \mathbf{if}\ b\ \mathbf{then}\ p_1\ \mathbf{else}\ p_2) \Downarrow C_2}
$$

$$
\text{Assert:}\frac{[\![c]\!]_\rho \in C \vee [\![c]\!]_\rho \notin \texttt{space}}{\rho \vdash (C, \mathbf{assert}\ c) \Downarrow C} \qquad \text{Flag:}\frac{}{\rho \vdash (C, \mathbf{flag}\ c) \Downarrow C \cup \{[\![c]\!]_\rho\}}
$$

$$
\text{For:}\frac{\forall (S_k).\forall (U_k).\quad A = [\![a]\!]_\rho \quad B = [\![b]\!]_\rho \quad \forall i.\ U_i = C \cup \bigcup_{k \in [\![A,i]\!]} S_k \quad \forall A \leq i \leq B.\ \rho[x \leftarrow i] \vdash (U_{i-1}, p) \Downarrow U_i}{\rho \vdash (C, \mathbf{for}\ v = a\ \mathbf{to}\ b\ \mathbf{do}\ p) \Downarrow U_B}
$$

Fig. 4. Operational semantics of programs

Remark 4. As mentioned earlier, assertions check that a cell's value is known *only if this cell belongs to* `space`. Therefore, in the Jacobi 2D example above, $\mathbf{assert}\ (-1, 0, 0)$ would not fail.

Remark 5. In the loop rule, S_i represents the set of cells computed by the program at iteration i, while U_i represents the knowledge of the program since its execution started, until the beginning of iteration i.

2.3 Verification Conditions and Symbolic Execution

Proving the correctness of a program (at least for requirement 1) amounts to proving a statement of the form $\rho \vdash (C_1, p) \Downarrow C_2$. Can we automate this process?

The key insight here is that our domain-specific language is very simple. It is not Turing-complete. We take advantage of this fact to simplify the verification scheme.

We can in fact perform some *symbolic execution* of programs. For an environment ρ and a program p, we define a set $\mathrm{Shape}_\rho(p)$ that intuitively corresponds to the knowledge that the program will acquire after its execution *if it does not fail*. The rules defining $\mathrm{Shape}_\rho(p)$ are purely *syntactic* and given in Figure 5.

$$\begin{aligned}
\mathrm{Shape}_\rho(\textbf{nop}) &:= \emptyset, \quad \mathrm{Shape}_\rho(\textbf{flag}\ c) := \{[\![c]\!]_\rho\} \\
\mathrm{Shape}_\rho(\textbf{if}\ b\ \textbf{then}\ p_1\ \textbf{else}\ p_2) &:= \begin{cases} \mathrm{Shape}_\rho(p_1) \text{ if } [\![b]\!]_\rho = \top \\ \mathrm{Shape}_\rho(p_2) \text{ otherwise} \end{cases} \\
\mathrm{Shape}_\rho(p_1; p_2) &:= \mathrm{Shape}_\rho(p_1) \cup \mathrm{Shape}_\rho(p_2) \\
\mathrm{Shape}_\rho(\textbf{assert}\ c) &:= \emptyset \\
\mathrm{Shape}_\rho(\textbf{for}\ x = a\ \textbf{to}\ b\ \textbf{do}\ p) &:= \textstyle\bigcup_{k \in [\![A,B]\!]} \mathrm{Shape}_{\rho[x \leftarrow k]}(p), \quad A = [\![a]\!]_\rho, B = [\![b]\!]_\rho
\end{aligned}$$

Fig. 5. Symbolic execution of programs

Note that $\mathrm{Shape}_\rho(p)$ will also allow us to prove requirement 2 from the correctness statement: the latter can be reformulated as $\texttt{target} \subseteq \mathrm{Shape}_\rho(p)$. To be more precise, let us now formalize the notion of correctness for sequential stencil algorithms.

Definition 2. *Consider a stencil* $(\texttt{space}, \texttt{target}, \texttt{dep})$. *For every cell* c, *let* $\textbf{fire}\ c \equiv \textbf{assert}\ d_1; \ldots; \textbf{assert}\ d_n; \textbf{flag}\ c$, *where* $\texttt{dep}(c) = \{d_1, \ldots, d_n\}$. *Here and after, it is assumed that the user uses exclusively the* **fire** c *command, and not* **flag** c.

Let ρ_0 *denote an empty environment. A program* p *is correct with respect to the aforementioned stencil if* $\rho_0 \vdash (\emptyset, p) \Downarrow \mathit{Shape}_{\rho_0}(p)$ *and* $\mathit{Shape}_{\rho_0}(p) \subseteq \texttt{target}$.

Now that we have a means to symbolically evaluate programs, we can write our verification-condition generator. The symbolic-execution step is very important, since it allows us to *synthesize loop invariants*, without the need for any human intervention. The verification-condition generator is defined in Figure 6.

Theorem 1 proves the correctness of the verification-condition generator.

Theorem 1. *Let* p *be a program,* ρ *an environment, and* C *a set of cells. If* $VC_{\rho,C}(p)$ *holds, then* $\rho \vdash (C, p) \Downarrow (C \cup \mathit{Shape}_\rho(p))$.

$$\mathrm{VC}_{\rho,C}(\textbf{nop}) := \top, \quad \mathrm{VC}_{\rho,C}(\textbf{flag}\ c) := \top$$
$$\mathrm{VC}_{\rho,C}(\textbf{if}\ b\ \textbf{then}\ p_1\ \textbf{else}\ p_2) := \begin{cases} \mathrm{VC}_{\rho,C}(p_1) \text{ if } [\![b]\!]_\rho = \top \\ \mathrm{VC}_{\rho,C}(p_2) \text{ otherwise} \end{cases}$$
$$\mathrm{VC}_{\rho,C}(p_1; p_2) := \mathrm{VC}_{\rho,C}(p_1) \wedge \mathrm{VC}_{\rho,C\cup\mathrm{Shape}_\rho(p_1)}(p_2)$$
$$\mathrm{VC}_{\rho,C}(\textbf{assert}\ c) := [\![c]\!]_\rho \in C \vee [\![c]\!]_\rho \notin \texttt{space}$$
$$\mathrm{VC}_{\rho,C}(\textbf{for}\ x = a\ \textbf{to}\ b\ \textbf{do}\ p) := \forall A \le i \le B.\ \mathrm{VC}_{\rho[x\leftarrow i],D}(p)$$
$$A = [\![a]\!]_\rho,\ B = [\![b]\!]_\rho,\ D = C \cup \mathrm{Shape}_\rho(\textbf{for}\ x = a\ \textbf{to}\ i-1\ \textbf{do}\ p)$$

Fig. 6. Verification-condition generator

3 Verifying Distributed Stencil Algorithms

We now turn to the problem of verifying *distributed* stencil algorithms. We will start with an informal description of our programming model.

3.1 Reduction to the Sequential Case

Stencil problems are inherently regular. Therefore, they are susceptible to substantial parallelization. More importantly, distributed stencil code is in general *synchronous*. The program alternates between *computation steps*, where each thread computes some cell values, and *communication steps*, during which threads send some of these values to other threads. Figure 7 gives a graphical representation of such an algorithm's execution, with three threads. Each of the three threads is assigned a strip in the plane, depicted by the dark dashed line. There is:

- A computation step, where each thread computes a "triangle";
- A communication step, during which each thread sends the edges of its "triangle" to its left and right neighbors;
- Another computation step, where each thread computes two "trapezoids."

We will use the phrase *time step* to refer to the combination of one computation step and one communication step. Our model, inspired by that of Xu et. al [21], also shares similarities with the Bulk-Synchronous Parallel (BSP) model [20].

Let us write a pseudo-code implementation of this simple algorithm, to give a flavor of what our formalization will eventually look like.

Computation step	Communication step

if T=0 **then**
  for t=0 **to** 3 **do**
  for $i = t$ **to** $7 - t$ **do**
  fire $(8 \times \mathrm{id} + i, t)$
else *(∗ T=1 ∗)*
  for t=1 **to** 3 **do**
  for $i = -t$ **to** $t - 1$ **do**
  fire $(8 \times \mathrm{id} + i, t)$
  for $i = -t$ **to** $t - 1$ **do**
  fire $(8 \times \mathrm{id} + 8 + i, t)$
	if T=0 **then**
	  if to $= \mathrm{id} - 1$ **then**
	  for t=0 **to** 3 **do**
	  fire $(8 \times \mathrm{id} + t, t)$
	  else if to $= \mathrm{id} + 1$ **then**
	  for t=0 **to** 3 **do**
	  fire $(8 \times \mathrm{id} + 4 + t, 3 - t)$

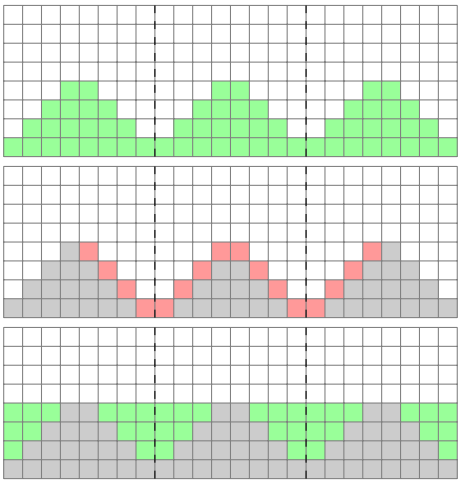

Fig. 7. Example of two computation steps (green) and one communication step (red)

On the left is the "computation kernel." Notice that every thread can access its unique identifier through the variable "id" and the current time step through variable "T". We also implicitly assume that "fire" commands involving a cell that is out of the rectangle depicted in Figure 7 have no effect. This is why we included `space` in our definition of stencils. The communication step is given on the right. This time, besides the variables "id" and "T", every thread has access to the variable "to", which contains the unique identifier of the thread it is currently sending data to. Now "fire" corresponds to sending a cell's local value to a neighbor thread.

For the reader familiar with verification of more complicated distributed code, we would like to emphasize that there is *no data race* here: threads have their own separate memories, and communication between threads only happens via message passing followed by barriers waiting for all threads to receive all messages sent to them. As a result, it does not make sense to talk about different threads racing on reads or writes to grid cells.

3.2 Distributed Kernels: Syntax, Semantics, and Correctness

The syntax of distributed code is given in Figure 8. We use the *same* syntax for both computation steps (in which case **fire** c means "compute the value of cell c") and for communication steps (in which case **fire** c means "send the value of cell c to the thread we are currently communicating with"). More formally, we give two different translations to the programs defined in Figure 8. The first one, Comp.denote, compiles **fire** c into the "fire" command of Definition 2, which *checks that all dependencies of* c *are satisfied*. The second semantics, Comm.denote, simply compiles it into a **flag** c command, *without checking any*

$$p ::= \textbf{nop} \mid p; p \mid \textbf{fire}\ c \mid \textbf{if}\ b\ \textbf{then}\ p\ \textbf{else}\ p \mid \textbf{for}\ v = e\ \textbf{to}\ e\ \textbf{do}\ p$$

Fig. 8. Syntax of distributed code

$$\begin{aligned}
F &\in \{\textbf{fire}\ , \textbf{flag}\ \} \\
\text{denote}_F(\textbf{nop}) &:= \textbf{nop}, \text{denote}_F(\textbf{fire}\ c) := F(c) \\
\text{denote}_F(\textbf{if}\ b\ \textbf{then}\ p_1\ \textbf{else}\ p_2) &:= \textbf{if}\ b\ \textbf{then}\ \text{denote}_F(p_1)\ \textbf{else}\ \text{denote}_F(p_2) \\
\text{denote}_F(p_1; p_2) &:= \text{denote}_F(p_1); \text{denote}_F(p_2) \\
\text{denote}_F(\textbf{for}\ x = a\ \textbf{to}\ b\ \textbf{do}\ p) &:= \textbf{for}\ x = a\ \textbf{to}\ b\ \textbf{do} \quad \text{denote}_F(p) \\
\text{Comp.denote}(p) := \text{denote}_{\textbf{fire}}\ (p), &\quad \text{Comm.denote}(p) := \text{denote}_{\textbf{flag}}\ (p)
\end{aligned}$$

Fig. 9. Translation of distributed code. The arguments are distributed programs, but the values returned by denote are sequential programs.

dependencies. This trick allows us to factor computation and communication steps into the same framework: a distributed kernel is a pair of sequential codes, one for each step. The translation is formally given in Figure 9.

Now that we are equipped with a language describing distributed stencil kernels and the associated semantics, we can formalize the correctness of such algorithms. A few intuitive comments are given right below the definitions, and the reader might find them useful in order to interpret our formalization.

Definition 3. *Suppose we fix a number id_{max}, such that threads are indexed over $[\![0, id_{max}]\!]$, and let us fix a maximum execution time T_{max}.*

A trace *is a triple* $(\texttt{beforeComp}, \texttt{afterComp}, \texttt{sends})$*, where* `beforeComp` *and* `afterComp` *have type* $\texttt{time} \times \texttt{thread} \to \texttt{set cell}$ *and* `sends` *has type* $\texttt{time} \times \texttt{thread} \times \texttt{thread} \to \texttt{set cell}$*.*

A distributed kernel *is a pair* $(\texttt{comp}, \texttt{comm})$ *of distributed codes. It is* correct *if there exists a trace satisfying the following properties:*

- *Initially, nothing is known:*

$$\forall 0 \leq i \leq id_{max}.\ \texttt{beforeComp}(0, i) = \emptyset;$$

- *We go from* $\texttt{beforeComp}(T, i)$ *to* $\texttt{afterComp}(T, i)$ *through a computation step:* $\forall 0 \leq i \leq id_{max}.\ \forall 0 \leq T \leq T_{max}.$

$$\rho_0[\text{“}id\text{”} \leftarrow i,\ \text{“}T\text{”} \leftarrow T] \vdash (\texttt{beforeComp}(T, i), \textit{Comp.denote}(\texttt{comp})) \Downarrow \texttt{afterComp}(T, i);$$

- $\texttt{sends}(T, i, j)$ *represents what is sent by thread i to thread j at step T:*

$$\forall 0 \leq i, j \leq id_{max}.\ \forall 0 \leq T \leq T_{max}.$$
$$\rho_0[\text{“}id\text{”} \leftarrow i,\ \text{“}to\text{”} \leftarrow j,\ \text{“}T\text{”} \leftarrow T] \vdash (\emptyset, \textit{Comm.denote}(\texttt{comm})) \Downarrow \texttt{sends}(T, i, j);$$

- *A thread cannot send a value it does not know:*

$$\forall 0 \leq i, j \leq id_{max}.\ \forall 0 \leq T \leq T_{max}.\ \texttt{sends}(T, i, j) \subseteq \texttt{afterComp}(T, i);$$

– *"Conservation of knowledge": what a thread knows at time $T+1$ comes from what it knew at time T and what other threads sent to it:*

$$\forall 0 \leq i \leq id_{max}.\ \forall 0 \leq T \leq T_{max}.$$

$$\texttt{beforeComp}(T+1, i) \subseteq \texttt{afterComp}(T, i) \cup \bigcup_{j \in [\![0, id_{max}]\!]} \texttt{sends}(T, j, i);$$

– *Completeness: when we reach time step $T_{max}+1$, all the required values have been computed:*

$$\texttt{target} \subseteq \bigcup_{i \in [\![0, id_{max}]\!]} \texttt{beforeComp}(T_{max}+1, i).$$

Remark 6. $\texttt{beforeComp}(T, i)$ represents the set of cells whose values are known by thread i at the beginning of the T^{th} time step. Similarly, $\texttt{afterComp}(T, i)$ represents the knowledge of thread i right after the computation step of time step T. Finally, $\texttt{sends}(T, i, j)$ represents the set of cells whose values are sent from thread i to thread j at the end of time step T.

3.3 Trace Generation

The definition of correctness involves proving a lot of different properties. Nevertheless, we will now show how the tools developed in the previous section, the symbolic-execution engine and the verification-condition generator, can be used to support mostly automated verification. In particular, we will *synthesize the trace that the program would follow if it does not fail*. The trace generator is given in Figure 10.

$$\begin{aligned}
[\texttt{sends}](k, \mathrm{id_{max}}, T, i, j) &:= \mathrm{Shape}_{\rho_0[\text{“id”} \leftarrow i, \text{“to”} \leftarrow j, \text{“T”} \leftarrow T]}(\mathrm{Comm.denote}(k)) \\
[\texttt{computes}](k, \mathrm{id_{max}}, T, i) &:= \mathrm{Shape}_{\rho_0[\text{“id”} \leftarrow i, \text{“T”} \leftarrow T]}(\mathrm{Comp.denote}(k)) \\
[\texttt{beforeComp}](k, \mathrm{id_{max}}, T, i) &:= \bigcup_{t \in [\![0, T-1]\!]} [\texttt{computes}](k, \mathrm{id_{max}}, t, i) \cup \bigcup_{\substack{t \in [\![0, T-1]\!] \\ j \in [\![0, \mathrm{id_{max}}]\!]}} [\texttt{sends}](k, \mathrm{id_{max}}, t, j, i) \\
[\texttt{afterComp}](k, \mathrm{id_{max}}, T, i) &:= \bigcup_{t \in [\![0, T]\!]} [\texttt{computes}](k, \mathrm{id_{max}}, t, i) \cup \bigcup_{\substack{t \in [\![0, T-1]\!] \\ j \in [\![0, \mathrm{id_{max}}]\!]}} [\texttt{sends}](k, \mathrm{id_{max}}, t, j, i)
\end{aligned}$$

Fig. 10. Trace generator

Of course, the trace generator would be useless without a proof of its correctness. Theorem 2 is the key result of this paper: it shows that, thanks to the trace generator, verification of distributed kernels *boils down to verifying two sequential programs*, proving a law of "conservation of knowledge" and a set inclusion that encodes completeness.

Theorem 2. *Let $k = (\texttt{comp}, \texttt{comm})$ be a kernel. If the following conditions hold:*

– $\forall 0 \leq i \leq id_{max}.\ \forall 0 \leq T \leq T_{max}.\ VC_{\rho_{i,T},D}(Comp.denote(\texttt{comp}))$, *where* $\rho_{i,T} = \rho_0[\text{“}id\text{”} \leftarrow i, \text{“}T\text{”} \leftarrow T]$ *and* $D = [\texttt{beforeComp}](k, id_{max}, T, i)$;
– $\forall 0 \leq i, j \leq id_{max}.\ \forall 0 \leq T \leq T_{max}.\ VC_{\rho_{i,j,T},\emptyset}(Comm.denote(\texttt{comm}))$, *where* $\rho_{i,j,T} = \rho_0[\text{“}id\text{”} \leftarrow i, \text{“}to\text{”} \leftarrow j, \text{“}T\text{”} \leftarrow T]$;
– $\forall 0 \leq id, to \leq id_{max}.\ \forall 0 \leq T \leq T_{max}.\ \ [\texttt{sends}](k, id_{max}, T, id, to) \subseteq [\texttt{afterComp}](k, id_{max}, T, id)$;
– $\texttt{target} \subseteq \bigcup_{i \in [\![0, id_{max}]\!]} [\texttt{beforeComp}](k, id_{max}, T_{max} + 1, i)$.

Then, k is correct, with trace

$$([\texttt{beforeComp}](k, id_{max}), [\texttt{afterComp}](k, id_{max}), [\texttt{sends}](k, id_{max}))\,.$$

Notice that we get very close here to the way a human being would write the proof of correctness: we have to prove that the dependencies are satisfied at any point, and the current state of the program is synthesized for us. Then, we need to prove that the final set of values contains all those that had to be computed.

4 Implementation and Experimental Results

The framework described in this paper has been implemented in Coq. In this section, we show how our library can be used to prove a very simple stencil algorithm for the two-dimensional Jacobi stencil introduced in Figure 1, and whose Coq definition is given in Figure 2.

4.1 A Simple Example

Let us come back to our straightforward sequential program:

```
Definition naive_st :=
  (For "t" From 0 To T Do
     For "i" From 0 To I Do
       For "j" From 0 To J Do
         Fire ("t":aexpr, "i":aexpr, "j":aexpr))%prog.
```

Let us start by stating the correctness of this algorithm:

```
Fact naive_st_correct : correct naive_st.
Proof.
  split.
```

We obtain two subgoals. The first corresponds to the verification conditions and can be simplified by using the symbolic-execution engine. We then clean up the goal.

```
  * decide i=0; [bruteforce | bruteforce' [i-1; i0; i1]].
```

The case $i = 0$ is special and deserves special treatment. Both cases are handled by our automation. The `bruteforce` tactic discharges the first subgoal, while its sister, `bruteforce'`, which takes as argument a list of candidates for existential-variable instantiation, discharges the second one.

The four other subgoals are handled similarly.

```
  * decide i=0; [bruteforce | ].
    decide i0=0; [bruteforce | bruteforce' [i-1; i0-1; i1]].
  * decide i=0; [bruteforce | ].
    decide i0=I; [bruteforce | bruteforce' [i-1; i0+1; i1]].
  * decide i=0; [bruteforce | ].
    decide i1=0; [bruteforce | bruteforce' [i-1; i0; i1-1]].
  * decide i=0; [bruteforce | ].
    decide i1=J; [bruteforce | bruteforce' [i-1; i0; i1+1]].
```

Although syntactically different, the computer-checked proof is very similar to the one a human being would write: case analysis to tackle boundaries, and the remaining proofs are "easy."

The second part of the proof of correctness is completeness:

$$\mathrm{Shape}_{\rho_0}(\texttt{naive\_st}) \subseteq \texttt{target},$$

which is easily discharged by our automation, this time with the `forward` tactic. Contrary to `bruteforce`, the latter tries to "make progress" on the goal, without failing if it cannot discharge it completely.

```
  - unfold target; simpl; simplify sets with ceval.
    forward. subst; forward. simpl; forward.
Qed.
```

4.2 Automation: Sets and Nonlinear Arithmetic

In this section, we describe the different tactics that we designed to reduce the cost of verifying stencil code.

Sets are represented as predicates: given a universe $\mathcal{U}$, a set has type $\mathcal{U} \longrightarrow$ `Prop`. The empty set is $\emptyset : u \mapsto \bot$, while for example the union of two sets A and B is defined as $A \cup B : u \mapsto A(u) \vee B(u)$. These definitions are here to give an experience to the user as close as possible to a handwritten proof, and they are automatically unfolded and simplified though first-order reasoning when using the tactic library. This process is implemented by two tactics, `simplify_hyps` and `simplify_goal`, which respectively clean up the current context and goal.

Programs are Coq terms. Therefore, the symbolic execution and trace synthesis are purely *syntactic*. We provide a tactic `symbolic execution` that unfolds all the required definitions.

This choice brings one inconvenience: their output has to be cleaned up. Some variables may be inferred automatically thanks to symbolic execution, which leads to expressions of the form **if** $0 = 1$ **then** A **else** B, where A and B are sets. Or, some of the verification conditions may look like $c \in \emptyset \cup \emptyset \cup \emptyset \cup \{c\}$, where c is a given cell. While the first one is pretty harmless, the second one can reduce the efficiency of automation: to prove that an element belongs to the union of two sets, we have to try and prove that it belongs to the first one, and if that fails, that it belongs to the second one. Therefore, we have implemented a *rewriting system* that tries to simplify the goal heuristically. For example, we use the following simplification rules:

$$A \cup \emptyset = \emptyset \cup A = A \qquad \bigcup_{c \in [\![a,b]\!]} A \times \{c\} = A \times [\![a,b]\!]$$

Moreover, we proved that set-theoretic operations are "morphisms," which in Coq's jargon means that we can apply the simplification rules to *subterms*. The rewriting system is implemented as a tactic called `simplify sets`.

The goals we obtain are set-theoretic: we usually have to prove that an element belongs to a set (*e.g.*, to show that a cell's value has already been computed) or that a set is a subset of another one (*e.g.*, what we need at this step was already known from the previous step). Most of this is first-order reasoning and is dealt with by the `forward` tactic, which repeatedly applies `simplify_goal`, `simplify_hyps`, and some first-order reasoning to the goal and context until no progress is made.

The next obstacles are goals of the following forms: $x \in A \cup B$ and $x \in \bigcup_{t \in [\![a,b]\!]} A_t$. We have already mentioned how the first one could be handled, contingent on the number of unions being "not too large." The second one can be tackled by taking as input a list of candidate variables, which we use to instantiate unknown parameters like t in the above expression. This more aggressive automation is implemented as the `bruteforce'` tactic, which takes a list of variables as input. `bruteforce` is a shortcut for `bruteforce'` $\emptyset$.

The challenge for full automation in stencil-code verification is that set-theoretic reasoning has to be followed up by a final arithmetic step. Indeed, a goal like $t \in [\![a, b]\!]$ is equivalent to $a \leq t \leq b$. But most of the time, stencil code acts on *blocks*, or subregions within grids, which are parameterized by some integers. For example, a typical goal might be $t \in [\![N \cdot a, (N+1) \cdot a - 1]\!]$. This leads to very nonlinear systems of inequalities. In our experience, most of these goals can be discharged using Coq's `nia` tactic, an (incomplete) proof procedure for integer nonlinear arithmetic.

Unfortunately, in our experience, `nia` is somewhat slow to fail, when given an unprovable goal. This is the main obstruction to a fully automated framework. We built a tactic to accumulate a list of all variables available in the current context and use it to enumerate expression trees that could be used as instantiation candidates in goals involving parameterized unions. However, in practice for interactive proving, we found it unusable with `bruteforce` due to the combinatorial explosion, combined with `nia`'s slowness to fail. It may still be cost-effective in overnight proof-search runs, for a program that is not likely to need much further debugging. In that case, we achieve full automation for a variety of stencil algorithms.

4.3 More Examples

We have implemented a few stencil algorithms and proved their correctness. They come from different areas, including simulation of a differential equation, computational finance, and computational biology. Table 1 shows the number of lines of code needed to prove their correctness. The framework scales well and allows to prove optimized and optimal algorithms of various kinds.

Examples come in four different groups.

- The *two-dimensional Jacobi kernel* was introduced at the beginning of this paper (see Figure 1). We verified a naive sequential algorithm that is often

used as a textbook example for finite-difference methods, applied to the Heat Equation.
- We verified a cache-oblivious sequential algorithm as well as a communication-efficient distributed kernel for *three-point stencils*.
- We also verified a cache-oblivious sequential algorithm and a communication-efficient distributed kernel for *American put stock options pricing*. This example is interesting since dependencies go backward in time: the price of an option depends on the price of the underlying asset *in the future*.
- The *Pairwise Sequence Alignment* problem is different from the other examples. It shows that our framework can be used to prove the correctness of algorithms based on *dynamic programming*.

	Type	Lines of Proof
Heat Equation, 2D	Naive	30
American Put Stock Options	Naive	25
American Put Stock Options	Optimized	25
Distributed American Put Stock Options	Naive	65
Distributed American Put Stock Options	Optimized	150
Pairwise Sequence Alignment	Dynamic programming	20
Distributed Three-Point Stencil	Naive	60
Distributed Three-Point Stencil	Optimized	160
Universal Three-Point Stencil Algorithm	Optimal	300

Table 1. Stencils implemented using our framework

5 Conclusions and Future Work

In this paper, we have shown how dependencies for both sequential and distributed stencil algorithms could be formally verified, and how to design automation to drastically reduce the cost of proving the correctness of such programs.

By focusing on a restricted class of problems and working with a domain-specific language adapted to this class, we were able to *symbolically execute* algorithms, which allowed us to *synthesize* program states, therefore avoiding the need to manually write any kind of loop invariants. A natural and interesting extension of this work could be to add symbolic tracking of cache-relevant behavior.

We also showed how to verify *distributed* stencil algorithms. Here, the key result is that verifying *synchronous* algorithms, when their program states can be *synthesized*, actually boils down to the verification of *several sequential programs*. An interesting extension to this work would be to design an extraction mechanism able to translate our DSL into MPI code or conversely, to get a program in our DSL from MPI code.

Acknowledgments. This work was supported in part by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under Award Number DE-SC0008923; and by National Science Foundation grant CCF-1253229. We also thank Shoaib Kamil for feedback on this paper.

References

1. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: Proc. OOPSLA. pp. 113–132. ACM (2012)
2. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Formal proof of a wave equation resolution scheme: the method error. In: Interactive Theorem Proving, pp. 147–162. Springer (2010)
3. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave equation numerical resolution: a comprehensive mechanized proof of a C program. Journal of Automated Reasoning 50(4), 423–456 (2013)
4. Epperson, J.F.: An introduction to numerical methods and analysis. John Wiley & Sons (2014)
5. Feautrier, P.: Automatic parallelization in the polytope model. In: The Data Parallel Programming Model, pp. 79–103. Springer (1996)
6. Frigo, M., Strumpen, V.: Cache oblivious stencil computations. In: Proc. Supercomputing. pp. 361–366. ACM (2005)
7. Frigo, M., Strumpen, V.: The cache complexity of multithreaded cache oblivious algorithms. Theory of Computing Systems 45(2), 203–233 (2009)
8. Gardner, M.: Mathematical games – The fantastic combinations of John Conway's new solitaire game "Life". Scientific American 223(4), 120–123 (1970)
9. Immler, F., Hölzl, J.: Numerical analysis of ordinary differential equations in Isabelle/HOL. In: Interactive Theorem Proving, pp. 377–392. Springer (2012)
10. Kelly, W., Pugh, W.: A unifying framework for iteration reordering transformations. In: Proc. ICAPP. vol. 1, pp. 153–162. IEEE (1995)
11. LeVeque, R.J.: Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems, vol. 98. SIAM (2007)
12. Li, G., Gopalakrishnan, G.: Scalable SMT-based verification of GPU kernel functions. In: Proc. FSE. pp. 187–196. ACM (2010)
13. Li, G., Li, P., Sawaya, G., Gopalakrishnan, G., Ghosh, I., Rajan, S.P.: GKLEE: concolic verification and test generation for GPUs. In: PPOPP. pp. 215–224 (2012)
14. Loechner, V.: PolyLib: A library for manipulating parameterized polyhedra (1999), `http://camlunity.ru/swap/Library/Conflux/Techniques%20-%20Code%20Analysis%20and%20Transformations%20(Polyhedral)/Free%20Libraries/polylib.ps`
15. Maydan, D.E., Hennessy, J.L., Lam, M.S.: Efficient and exact data dependence analysis. In: Proc. PLDI. pp. 1–14. PLDI '91, ACM (1991)
16. Orchard, D., Mycroft, A.: Efficient and correct stencil computation via pattern matching and static typing. arXiv preprint arXiv:1109.0777 (2011)
17. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proc. Supercomputing. pp. 4–13. ACM (1991)
18. Solar-Lezama, A., Arnold, G., Tancau, L., Bodik, R., Saraswat, V., Seshia, S.: Sketching stencils. In: Proc. PLDI. pp. 167–178. ACM (2007)
19. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.K., Leiserson, C.E.: The Pochoir stencil compiler. In: Proc. SPAA. pp. 117–128. ACM (2011)
20. Valiant, L.G.: A bridging model for parallel computation. Communications of the ACM 33(8), 103–111 (1990)
21. Xu, Z., Kamil, S., Solar-Lezama, A.: MSL: a synthesis enabled language for distributed implementations. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 311–322. IEEE Press (2014)