Automatic Differentiation of Parallel Loops with Formal Methods

Jan Hückelheim, Laurent Hascoët

Automatic Differentiation of Parallel Loops with Formal Methods

Jan Hückelheim
jhueckelheim@anl.gov
Argonne National Laboratory
Lemont, Illinois, USA

Laurent Hascoët
Laurent.Hascoet@inria.fr
Inria Sophia Antipolis
Valbonne, France

ABSTRACT

This paper presents a novel combination of reverse mode automatic differentiation and formal methods, to enable efficient differentiation of (or backpropagation through) shared-memory parallel loops. Compared to the state of the art, our approach can reduce the need for atomic updates or private data copies during the parallel derivative computation, even in the presence of unstructured or data-dependent data access patterns. This is achieved by gathering information about the memory access patterns from the input program, which is assumed to be correctly parallelized. This information is then used to build a model of assertions in a theorem prover, which can be used to check the safety of shared memory accesses during the parallel derivative loops. We demonstrate this approach on scientific computing benchmarks including a lattice-Boltzmann method (LBM) solver from the Parboil benchmark suite and a Green's function Monte Carlo (GFMC) kernel from the CORAL benchmark suite.

KEYWORDS

Automatic Differentiation, OpenMP, Theorem Proving, Formal Methods, Data Flow Reversal

1 INTRODUCTION

Gradients, adjoints, and derivatives are useful in countless applications including weather and climate modeling, engineering, finance, and machine learning. Reverse-mode automatic differentiation (AD) or the closely related method of backpropagation are particularly efficient methods to obtain gradients or adjoints for large real-world applications. We provide some background on this in Section 4.1.

While reverse-mode AD and backpropagation are appealing due to their run time efficiency, they can be challenging to implement for parallel programs. One reason is the data flow reversal inherent to these methods, which turns read accesses in the original program (the "primal") into increment accesses in the derivative computation, which may result in data races if not done conservatively. Particularly on shared-memory parallel computers, it may therefore become necessary for an AD tool to make extensive use of atomic updates or privatized data copies that are later accumulated using reduction operations. This introduces time and memory overheads that can cause adjoint programs to have lower scalability than the primal program.

Previous work avoids this problem by performing differentiation on a high level of abstraction that hides the parallelization from the differentiation process, and results in derivatives that are expressed in terms of building blocks that are internally parallelized. This approach is very successful within domain specific languages and machine learning frameworks, but does not offer the flexibility that is needed for some scientific computing applications.

Instead, our work improves the code analysis of general-purpose AD tools by introducing a novel static analysis approach that combines formal methods and automatic differentiation to require fewer updates and reductions. We implemented this approach, which we will refer to as FormAD (Formal methods in AD), as an additional analysis pass within the existing AD tool Tapenade. Unlike many existing AD or automatic parallelization approaches, FormAD is able to parallelize loops even in the presence of unstructured and data-dependent memory access patterns, which are common in scientific applications. This is possible because FormAD exploits the relationship of primal programs with their corresponding gradients or adjoints.

2 RELATED WORK

Applying automatic differentiation to shared-memory parallel programs has been discussed extensively in the past. For example, [14] discusses manual detection of identical index expressions in the primal and adjoint program, followed by a postprocessing step to apply the same parallelization to both. Our work automates and generalizes this type of analysis. Other previous work [13, 15] focused on stencil kernels, where loop transformations or code generation from high-level specifications can be used to obtain efficient adjoint stencils. Our work is more general, as it can handle data-dependent index expressions, loop bounds, control flow, and other computations that can not be naturally expressed as regular stencils.

There is previous work discussing OpenMP support for AD tools, such as [2, 7, 8, 17], although it has been observed that parallel programs can be challenging to reverse-differentiate [1, 6], and that static analysis may be needed to ascertain the absence of increment-increment conflicts [6], which does not always succeed. Other papers use or discuss atomic or privatization/reduction constructs as a safe fallback [2, 8, 16], which causes a performance degradation that we try to avoid with our work.

Other work discusses knowledge derived from existing explicit parallelism [3]. While the idea is similar in spirit, they are interested in the absence of happens-before relationships to enable polyhedral transformations, while we are interested in the absence of overlapping memory addresses to enable AD.

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545089

3 CONTRIBUTIONS AND LIMITATIONS

We make the following contributions.

- We present a novel method to reason about the relationship between the memory access patterns in a primal and adjoint program. Compared to previous work, our theory extends beyond simple symmetry relationships, and is not restricted to particular access patterns such as stencils.
- We present an implementation of the above model (FormAD) within the AD tool Tapenade, which is to our knowledge the first combination of an automatic differentiation tool with a theorem prover. FormAD is publicly available under the same MIT open source license as Tapenade.
- We evaluate FormAD on a number of parallel scalability test cases, in which we improve upon the state of the art by factors ranging from 5× to over 13×.

We are aware of the following limitations of our work.

- FormAD currently only supports Fortran input programs. This is a consequence of the fact that Tapenade itself currently supports OpenMP only for Fortran programs. We expect C to be a straightforward extension, requiring only minor changes to the the Tapenade parser and scoping rules.
- FormAD is a prototype tool, and the OpenMP support in Tapenade is experimental. For example, synchronization barriers, tasks, offloading, and many other OpenMP features are not supported yet.
- We assume no aliasing between arrays (aliasing is forbidden in Fortran in many situations), and assume that multi-dimensional arrays are used "correctly", i.e. all indices are within the bounds of their dimension.
- FormAD assumes that the primal program is truly correctly parallelized. Some programs may include "benign" races, such as multiple threads writing the same value to a given location. These races are seldom truly benign, despite the expectation of some programmers. FormAD may in such cases produce derivative programs that break more frequently or more obviously than the (already incorrect) primal.

4 BACKGROUND

This section summarizes concepts that are essential to understand the remainder of this work.

4.1 Source-Transformation Adjoint AD

We present here only a small subset of Automatic Differentiation theory to introduce notation and make the paper more readable. A thorough introduction to AD can be found, for example, in [7, 9, 18]. We focus in this work on source-transformation AD, which transforms a given source code, referred to as the "primal', into a new code that computes derivatives of the primal's outputs with respect to its inputs. Furthermore, we focus on reverse-mode AD (implemented via source-transformation), which is a particularly efficient strategy when the primal has many inputs and few outputs. Applications with this property include shape optimization, inverse design, medical imaging, and machine learning, where reverse-mode AD is also known as back-propagation. Other application domains commonly refer to reverse-mode AD as "adjoint AD", and we use the terms gradients and adjoints interchangeably in this work. An inherent feature of reverse-mode AD is that it propagates derivatives backwards through the computation, starting from the outputs and finishing at the inputs. This property is the reason for its efficiency, and at the same time a source of implementation challenges.

The computation of derivatives typically works on two levels: Locally, each instruction performed by the primal code is augmented by a derivative instruction to compute that instruction's local gradient. Then, the derivative instructions must be combined globally such that the gradient of the entire computation is correctly accumulated. The challenges related to this global accumulation have been discussed in previous work, and include the need to store and retrieve branch decisions and intermediate values that influence the derivatives, and to correctly handle communication, parallel schedules, and variable scoping whenever necessary. We refer to [12] for a discussion of these issues. The challenge addressed in this paper occurs locally, during the execution of individual gradient instructions, and only if the propagation happens within a parallel region and affects a variable that is shared between threads.

We will now take a detailed look at the reverse-mode differentiation of a single instruction. Subsection 4.2 will discuss the interaction of differentiated instructions that may happen concurrently in a parallel region. We use as a representative example an assignment that has a binary operation on the right hand side. The treatment of operations with more (or fewer) operands is similar.

$$z = x \; Op \; y$$

We assume here that Op is a differentiable operation, and x, y, and z are variable references. These references could in practice be scalar variables, array items, dereferenced pointers, etc. We further assume for ease of notation that the memory location of z does not overlap with any variable on the right-hand side. If this assumption is violated, it is straightforward to introduce a temporary variable and split the assignment into two statements to make the assumption valid for each of the two statements.

With these assumptions, reverse-mode AD of the above instruction can be implemented as

$$\begin{aligned} \overline{x} &= \overline{x} + \overline{z} * \frac{\partial Op}{\partial x}(x,y) \\ \overline{y} &= \overline{y} + \overline{z} * \frac{\partial Op}{\partial y}(x,y) \\ \overline{z} &= 0, \end{aligned}$$

where $\overline{x}$, $\overline{y}$, and $\overline{z}$ are the gradients of the final output with respect to the current x, y, and z respectively. Formal justification of this involves writing the 3×3 derivative (*Jacobian matrix*) of the function implemented by the primal instruction, from $\mathbb{R}^3$ to $\mathbb{R}^3$ i.e. from the values stored in $\{x, y, z\}$ before the instruction to the values stored in $\{x, y, z\}$ after the instruction, and to multiply it *on the right* of the gradient vector $(\overline{x}, \overline{y}, \overline{z})$. An intuitive way to understand this is to view $\overline{x}$ as the influence of the current x on the final output. This influence is the sum of contributions from each place where x is used. In particular here, the contribution is the influence of z on the final output, magnified by $\frac{\partial Op}{\partial x}$. Similar reasoning holds for y. Finally the influence of z *before the instruction*, onto the output, is set to zero as z is going to be overwritten and therefore has finished its life cycle.

Assignment example	Increment example
`u(i-1) = a*v(i,j) + 1.5`	`u(2*i) = u(2*i) + 2*a`

`vb(i,j) = vb(i,j)`
`  + a*ub(i-1)`
`ab = ab + v(i,j)*ub(i-1)`
`ub(i-1) = 0`
	`ab = ab + 2*ub(2*i)`

Figure 1: Top: Two examples of original (primal) instructions. Bottom: Corresponding adjoint instructions for both examples. In real code, we name adjoint variables after the primal variable with "b" appended (read as "bar").

```
!$omp parallel do
do i=1,n
 y(c(i))=x(c(i)+7)
end do
```

```
!$omp parallel do
do i=n,1,-1
 xb(c(i)+7)=xb(c(i)+7)+yb(c(i))
 yb(c(i))=0.0
end do
```

Figure 2: A primal parallel loop with indirect memory access (left), and its corresponding adjoint loop (right). Assuming that the primal was correctly parallelized, for any two loop iterations i and i' the write indices `c(i)` and `c(i')` can not be identical. From this we deduce that the adjoint indices `c(i)+7` and `c(i')+7` can also not be identical, and thus we may increment without atomic pragma.

Figure 1 shows two examples of adjoint instructions of an assignment, including the special case where the assignment actually increments its left-hand side. The general differentiation rules then boil down to a simpler adjoint code where the adjoint of the left-hand side is not overwritten.

4.2 Adjoint of OpenMP shared variables

We now extend the discussion to the interaction of instructions across multiple threads within the same parallel region. We are specifically interested in the situation where the primal code is shared-memory parallelized, and we wish to compute its derivatives in a similarly scalable fashion. Previous work [6, 12] explains how this can be achieved in most situations. A challenge identified in both papers is the efficient treatment of variables that are shared (in other words, visible to multiple threads) in the primal. Although we (and the cited papers) use OpenMP terminology and mostly focus on OpenMP, we believe that other thread-parallel languages can be treated similarly. Frequently, `shared` variables are accessed by multiple threads concurrently for reading, which we will call a *read-read* situation. In the primal code this is harmless and therefore not called a conflict. However, as visible on the left of figure 1, the adjoint variable is incremented, which may cause a *write-write* conflict or more precisely an *increment-increment* conflict. Two increments to the same memory address may be executed by different threads within the same parallel region only if increment operations are done with (often expensive) *atomic* updates, or if some other expensive synchronization mechanisms are used. On most systems, atomic updates are costly even in favourable situations where each given memory address is accessed by only one thread. Because of this, it is important to use atomic updates only sparingly and when necessary, which is precisely the goal of our new analysis technique.

If we fail to prove the absence of a *read-read* situation on the primal variable v, instead of using atomic updates we may also privatize $\overline{v}$ as a `reduction(+)`, which also harms performance and increases the memory footprint of the application. Even though experimental libraries have been proposed [11] to reduce the cost of such reduction operations, it is nevertheless profitable to detect the absence of *read-read* situations in the primal code whenever possible. Detection of *read-read* situations can be addressed by classical data-dependence analysis. In a sense, this amounts to running an automatic parallelization tool on the adjoint code, which may detect the absence of read-read situations, or use domain-specific approaches such as polyhedral transformations to avoid conflicts. Although these options work in some cases, they are approximate (just like FormAD), and do not exploit domain-specific knowledge about the assumed-correct parallelization of the primal (unlike FormAD).

5 THE FORMAD APPROACH

We explain in this section our approach to more precisely analyze whether atomic updates (or some other safeguard) are needed to prevent race conditions in an adjoint program. Since we assume that parallel loops in the primal code are correct, they may not contain any loop-carried dependency, be it *write-to-read*, *read-to-overwrite*, or *write-to-overwrite*. Therefore for every pair of references to a given array that occur in a given primal parallel loop nest, at least one of these references being a *write*, we know that the array indices used are disjoint. This piece of information is exclusively about the two sets of index expressions and about their relation with the enclosing loop counters. It is not bound to the array itself. We can use this information to reason about other arrays accessed with similar index expressions, and in particular we can use it for the adjoint arrays in the adjoint parallel loop nest that corresponds to the given primal loop nest.

Consider for instance the simple parallel loop on the left of figure 2. The corresponding adjoint loop, shown on the right of the figure, is also a parallel loop. As we assume the primal loop is correct, we know there can be no loop-carried dependency on array `y`. In other words for any pair of loop indices `i` and $\mathtt{i}'$, $\mathtt{i} \neq \mathtt{i}'$, we know `y(c(i))` and $\mathtt{y(c(i')})$ can not overlap, i.e. `c(i)` $\neq$ $\mathtt{c(i')}$. This knowledge binds the iteration space of the parallel loop to the array offsets defined by the array indices, independently of the particular array `y` itself. We can therefore use this knowledge to show that there is no loop-carried dependency in the adjoint loop due to accesses to `xb` nor to accesses to `yb`. As a consequence, `xb` and `yb` can be declared `shared` in the adjoint loop, and there is no need for atomic pragmas or similar constructs.

For each parallel region, this results in two phases:

(1) **Knowledge extraction:** for each `shared` array in the given parallel region, we collect all read references and all write references. For example, in case of the stencil code in Section 7.1 we collect references for the arrays `unew` and `uold`.

For `unew`, the reference set includes reads and writes to both `i` and `i-1`. For each pair of such references, at least one being a write, we extract the index expression(s), and create a set of assertions for this region that expresses the fact that these indices are disjoint, i.e. they have a different value if loop counters differ. We call this set of assertions our knowledge base. We must handle the case where some variables appearing in the index expressions are modified in the region: this will be detailed in section 5.2.

(2) **Knowledge exploitation:** just before creation of the adjoint region, and for each *active* shared variable, i.e. one that will have an adjoint variable, we collect all future read and write references to the adjoint variable in the adjoint region. For each pair of such references, at least one being a write, we ask the knowledge base to prove that their index expressions have a different value when loop counters differ. Note that by construction, loop counters and other integer variables appearing in index expressions have the same value in the corresponding primal and adjoint instructions (they may be modified or overwritten in between these instructions, but the generated adjoint code must recompute or restore such instructions regardless of our approach). If for all pairs of references considered we can prove that different loop counters imply different indices, then the adjoint variable causes no *increment-increment* conflict, and can be declared simply `shared`.

Knowledge exploitation for a given adjoint variable may use constraints extracted from any primal variable. For example primal variables of `integer` type, which are not differentiable and thus have no adjoint counterpart, can still provide useful index knowledge. Likewise, one can imagine an adjoint variable referenced at three locations, and each of the three resulting potential conflicts being proved inexistent using knowledge from three different primal variables. Conversely if no knowledge is available the proof system can still, in some cases, prove the absence of conflicts just like any automatic parallelization tool.

5.1 Dealing with control flow

In general, the body of the parallel loops in the parallel region need not be simple straight-line code. Since we are defining a static analysis and control is dynamic, we must distinguish *may*-information from *must*-information on the knowledge we extract and use. We handle this by defining a notion of control *context*, representing the control decisions that lead to executing a given instruction. The context C_2 attached to an instruction I_2 will be said *included* in the context C_1 of I_1 if any iteration of the parallel loop nest that executes I_2 necessarily executes I_1. If $C_2 \subset C_1$ and $C_1 \subset C_2$, then $C_1 = C_2$. The top-level of the body of the parallel region receives a *root* context. If the code is well structured, included contexts can be created recursively. For the general case of an arbitrary control-flow graph, we instead use a classical dominator analysis: I_1 *dominates* I_2 if it is a necessary predecessor of I_2, and I_1 *post-dominates* I_2 if it is a necessary successor of I_2. Each case implies that $C_2 \subset C_1$.

Knowledge extraction creates a different knowledge base for each context. A context included in another context `C` inherits all knowledge from `C`. Each time we consider a pair (R_1, R_2) of references to a given array and try to extract knowledge from it, their respective contexts (C_1, C_2) are compared. Knowledge is added only to contexts that must execute both R_1 and R_2. If $C_1 = C_2$, knowledge is added to this context (and consequently to all included contexts). If $C_1 \subset C_2$, then knowledge is added to C_1 and not to C_2, and likewise if $C_2 \subset C_1$. Otherwise no knowledge is added as no control certainly executes both R_1 and R_2.

Knowledge exploitation works in a dual way. When considering a pair $(\overline{R_1}, \overline{R_2})$ of references to one adjoint array, we take the contexts (C_1, C_2) of the corresponding references in the primal code. When trying to prove the absence of loop-carried dependency between $\overline{R_1}$ and $\overline{R_2}$, we are only allowed to use the knowledge attached to the common root of C_1 and C_2, which is the knowledge available from iterations that executed both $\overline{R_1}$ and $\overline{R_2}$.

5.2 Distinguishing overwritten index variables

Variables occurring in index expressions may be modified during execution of the parallel body (except for the loop index itself, as stated by the OpenMP standard) so that we cannot compare them just textually. We therefore complement them with a so-called *instance* number. Two uses of one variable will get the same instance number when they are reached by the same set of Def-Use chains. The proof system will be given index expressions in which each variable name is accompanied with its instance number.

Instance numbers are created as follows. Each time an instruction overwrites a variable, the current instance number for this variable is set to a new value. As our method is static, control flow introduces some blurring. When different control flows merge at some point, variables with an instance number that differs according to the flow must receive yet another new instance number. Similarly, at the entry into a loop that overwrites some variable, the instance number of this variable must be renewed, thus representing either the entry value or the value coming from the previous iteration.

5.3 Handling private variables

Each thread holds its own instance of every private variable. This includes variables appearing in `private`, `reduction`, `firstprivate`, or `lastprivate` clauses. When sending a pair of index expressions to the proof system, either as extracted knowledge or as a question during exploitation, the first expression is modified by substituting private variables with a sibling variable (typically by appending a ' to its name), to express that it is distinct to the similarly named variable in the second expression. A special case is the parallel loop counter, which is by default private, but unlike other private variables, provides additional knowledge since the instances on two threads must not have the same value.

5.4 Taking advantage of AD-specific structure

We can take advantage of knowledge specific to AD to reduce the number of reference pairs to consider for loop-carried independence. Not all variable references in the primal code give birth to a corresponding adjoint reference in the adjoint code. To begin with, only variables with a differentiable type (`real` or `complex`) may require a derivative variable, as well as variables of a structured type that contains fields of a differentiable type. Additionally, Tapenade and some other AD tools perform *activity* analysis to

detect instructions or variables that do not influence the derivatives. Thanks to activity analysis, only a subset of the variable references of the given parallel region have a corresponding adjoint variable reference, which reduces the number of array references that must be analyzed by FormAD.

Furthermore, we detect when a primal instruction exactly increments a variable. As shown on the right of figure 1, the corresponding adjoint will only read and not overwrite or increment. Since FormAD is only needed to detect pairs with at least one write, this reduces the number of reference pairs to consider.

5.5 Proof system

The proof system is provided with sets of index expressions that are known to be disjoint. Each expression set is valid within a particular context of the original program. The proof system contains a procedure that, within a given context, builds a set of assertions that is known to hold. This assertion set is passed recursively to all child contexts. The assertion set is built through a pairwise combination of the expressions with the $\neq$ operator, where private variables on one side are replaced with sibling variables as explained above. The recursive procedure is started at the root context, corresponding to the body of the parallel loop, with the extra assertion $\mathtt{i} \neq \mathtt{i}'$, which expresses the fact that no two threads may have the same loop counter value, as is guaranteed by OpenMP. In pseudo code, this procedure is as follows:

```
buildModel(rootContext, i_prime != i)

def buildModel(context, assertions):
  model = z3.Solver()
  model.add(assertions)
  for var in variables:
    writeexprs = var.writeexprs(context)
    readexprs = var.readexprs(context)
    for expr0 in writeexprs:
      expr0p = expr0.putprimes()
      for expr1 in writeexprs+readexprs:
        model.add(expr0p != expr1)
        assert(model.check() == SAT)
  for child in context.children:
    buildModel(child, model)
```

One can see that the model size is proportional to the product of the number of write expressions times the number of write or read expressions. If the expressions contain variable references, they will be with the appropriate instance number. Note the assertion that checks after each addition whether the model is still satisfiable. While this increases the number of calls to the theorem prover (here called `model.check()`), we add this as a safeguard because a failing assertion means that the expressions found in the primal program can not all be disjoint, pointing at a data race in the primal program or a bug in our system. To improve run time, the assertion could be checked less frequently.

The knowledge exploitation phase tests, for each adjoint variable, the set of candidate conflicting index expressions given by Tapenade. This is done by following a similar quadratic procedure to build pairs of expressions. We know by construction (and from our assertion) that the existing model for each context is satisfiable. By adding a pair of expressions from the set of candidate expressions and asserting its equality, the model becomes unsatisfiable if the expressions are provably disjoint. If the model remains satisfiable or if the theorem prover fails to come to a conclusion, the tested expression pair may not be disjoint and we will assume that the parallel accesses to this adjoint variable are unsafe. For a given model (associated with a context) and some adjoint variable's candidate write and read index expressions, this can be written as follows:

```
def testVar(model, writeexprs, readexprs):
  for expr0 in writeexprs:
    expr0p = expr0.putprimes()
    for expr1 in writeexprs+readexprs:
      model.push()
      model.add(expr0p == expr1)
      if(model.check() != UNSAT):
        return False # unsafe pair
      model.pop()
  return True # all pairs are safe
```

We use a functionality of Z3 that allows us to push the model state onto a stack, then modify it (in this case, by adding another assertion), and later restoring the previous state.

6 IMPLEMENTATION

We implement FormAD[1] within the Source-Transformation AD tool Tapenade [10]. The implementation also uses the Z3 [5] model checker through its Java API. We use version 4.8.15-x64-glibc-2.31 of Z3 without modifications (although other versions will probably work as well), and only modify Tapenade to enable our approach. Prior to our work, Tapenade did not support theorem proving and did not analyze accesses to shared arrays. Therefore we implemented the following functionalities:

- Identification of the *contexts* (*cf* section 5.1) and of the *instances* (*cf* section 5.2). Contexts are detected on the control flow graph, based on the pre-existing dominator and post-dominator analysis of Tapenade. Instances are detected by a new data-flow analysis.
- Extraction, from the parallel regions of the primal code, of *knowledge* of non-existing conflicts as well as of *questions* about potential *increment-increment* conflicts on future shared adjoint variables. This is implemented by a single sweep on the control flow graph of parallel loop bodies of the primal code, that collects all reads and writes of arrays then accumulates the (non-)conflicting pairs of those. The existing activity analysis of the AD tool is instrumental here to get fewer questions (*cf* section 5.4).
- Translation of each elementary knowledge or question, which are pairs of array index expressions that are either known or requested independent, into the form of a Z3 assertion. Starting from the two expressions syntax tree, parallel loop indices are identified, and all other variables are equipped

[1]The version used for the experiments in this paper is available at https://gitlab.inria.fr/tapenade/tapenade/-/tags/ICPP2022

with their current instance number. Actual translation into a Z3 assertion uses a depth-first then bottom-up traversal of both syntax trees.

The central algorithm described in section 5.5 is also implemented in Tapenade, in the code section devoted to differentiation of a thread-parallel loop. For each context, the accumulated knowledge is passed as a set of assertions to Z3, then each question is posed separately to Z3. When all potential conflicts about an adjoint variable have been proven nonexistent, Tapenade can consequently declare these conflict-free adjoint variables as `shared` with no `atomic` pragma added.

The added analysis and algorithms are triggered if and only if thread-parallel loops are present. As Tapenade's internal representation for multithread loops was kept reasonably independent from OpenMP, we have good hope that this implementation effort can be reused for other multithread dialects. For instance a current development to extend Tapenade for Cuda addresses the same internal representation.

7 EXPERIMENTAL RESULTS

We evaluated FormAD on a number of test cases as discussed in the following subsections. We discuss the performance of the FormAD analysis itself in Section 7.5, after showing run times of the generated derivative programs. For some test cases, our method did not find any opportunities to remove atomic pragmas, in which case we do not actually compile and run the generated programs, since there would be no performance difference to the baseline.

For all other test cases we create the following program versions, all of which are compiled with the Intel Fortran compiler version 2021.2.0 and flags `-O3 -xHost -qopenmp`:

Primal, the original function.
Adjoint Serial, the function generated by reverse-mode AD without any OpenMP pragmas.
Adjoint FormAD, generated by reverse-mode AD with FormAD to avoid atomics whenever it is safe.
Adjoint Atomic, generated by reverse-mode AD using `atomic` pragmas to guard shared array increments.
Adjoint Reduction, generated by reverse-mode AD using reductions to guard increments to shared arrays.

Unless otherwise stated, when a serial time is reported, it is obtained using the serial program version, not by running one of the other versions with just one thread.

Our test system is a dual socket system with two Intel Xeon E5-2695v4 (Broadwell) processors. To avoid NUMA issues we pin our threads to one socket, which has 18 cores. The processor supports up to 36 threads using hyper-threading, but we did not see a benefit from this in preliminary test runs and therefore use a maximum of 18 threads in our benchmarks. We report average run times from 10 consecutive runs on the same system. We note that we only avoid using a NUMA system to simplify our experimental evaluation, since neither of our test cases was NUMA-aware to begin with, and our approach is unrelated to NUMA. If a given program uses NUMA-aware thread placement and memory allocation, an AD tool could imitate that same strategy for the derivative code regardless of whether or not FormAD is used.

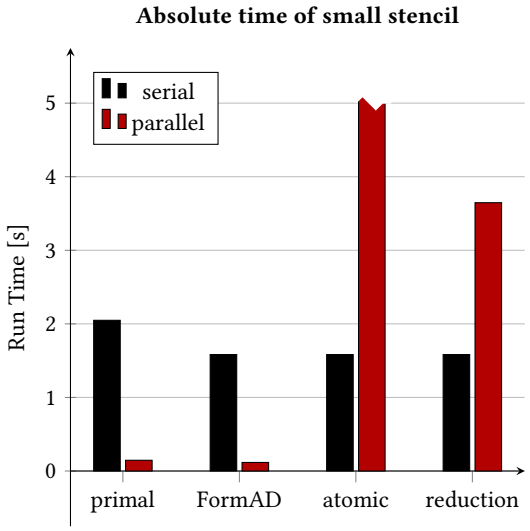

Figure 3: Absolute run time for small stencil test case. The primal serial program takes 2.05s**, whereas the parallel version takes** 0.146s**. The best-performing thread count for the parallel adjoint programs was** 1**, with** 40.7s **for the atomic and** 3.65s **for the reduction version, both much worse than the sequential adjoint at** 1.58s**. The FormAD version achieves a run time of** 0.116s **on 18 threads.**

Whenever we report parallel speedup numbers, we use the serial version (without any OpenMP pragmas) as the baseline. This is done to make a fair comparison with the times that a user would actually obtain without parallelism, and causes most of our speedup plots to start below 1 when only one thread is used, since reductions and atomics introduce an overhead even in single-threaded execution.

It should be noted that the program versions with reduction pragmas have a significantly larger memory footprint than the other program versions, due to the need to store privatized instances of the output data on each thread. On the other hand, whether or not atomics are used does not significantly affect the memory footprint of our test cases. While this can be seen as another advantage of using our approach, we do not provide an in-depth discussion or measurements of memory consumption in this paper.

7.1 Stencil

Stencils commonly occur in structured-mesh PDE solvers, image processing, and convolutional neural network layers. Previous work has proposed an implementation strategy that balances the number of load/store operations in stencil kernels where each output index depends on a large number of input indices [19]. The "compact" scheme introduced in that work has identical read and write sets in each iteration, which means that any parallelization scheme that is safe for the primal must also be safe for the reverse mode AD. As an example, we show the core computation (boundary treatment and some factors omitted) which is equivalent to a conventional three-point stencil here:

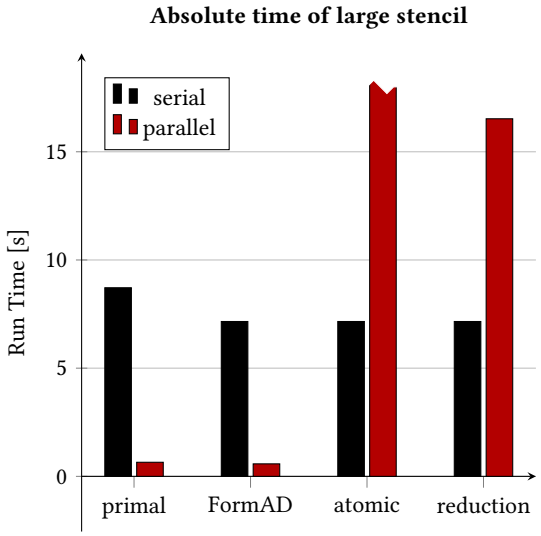

Figure 4: Absolute run time for large stencil test case. The primal serial program takes 8.72s, whereas the parallel version takes 0.651s. The best-performing thread count for the parallel adjoint programs was 1, with 95.8s for the atomic and 16.5s for the reduction version, both much worse than the sequential adjoint at 7.16s. The FormAD version achieves a run time of 0.578s on 18 threads.

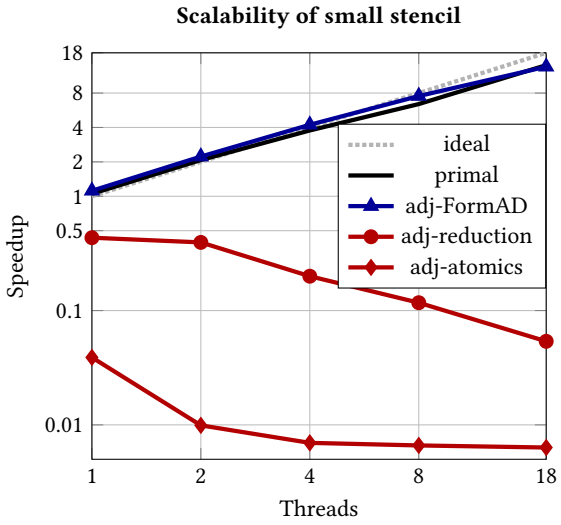

Figure 5: Parallel speedup for small stencil test case. The primal and FormAD programs scale well, with a maximum speedup on 18 threads of 13.4× and 13.6× respectively. The adjoint programs with atomics and reductions never exceed the serial performance, and actually slow down as more threads are added.

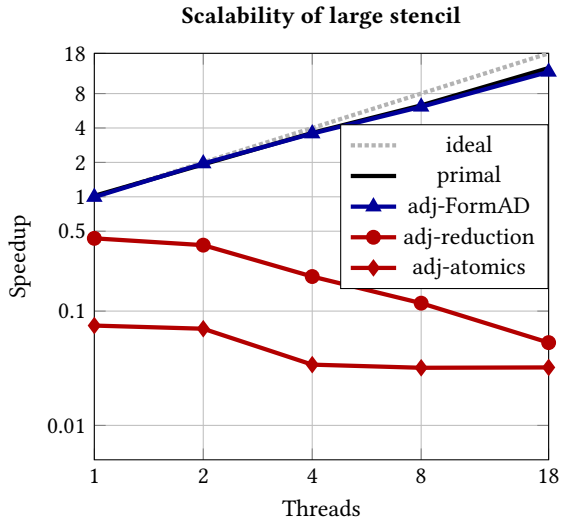

Figure 6: Parallel speedup for large stencil test case. The primal and FormAD programs scale well, with a maximum speedup on 18 threads of 13.12× and 12.4× respectively. The adjoint programs with atomics and reductions never exceed the serial performance, and actually slow down as more threads are added.

```
do offset=0,1
  from = 2*1+offset
  !$omp parallel do shared(unew,uold)
  do i = from, n-2, 2
    unew(i) = unew(i) + wl*uold(i-1)
    unew(i) = unew(i) + wc*uold(i)
    unew(i-1) = unew(i-1) + wr*uold(i)
  end do
end do
```

We apply FormAD to a compact 3-point and 17-point stencil-equivalent, respectively referred to as *small* and *large* stencil, where the output field `unew` is differentiated with respect to the input field `uold`. Without FormAD, all increments in the generated adjoint code are safeguarded with atomics, or alternatively, the derivative counterpart variable of the input array is placed in a reduction clause. Conversely, FormAD is able to prove the safety of the generated adjoint code, and no atomics or reduction constructs are used in the program generated with FormAD.

We run this test case for a domain size of 1M grid points and perform 1000 sweeps over the domain. Figures 5 and 6 show the parallel speedup for the small and large stencil, whereas Figures 3 and 4 show absolute run times. The FormAD-generated adjoint programs generally scale as well as the corresponding primal programs, and outperform the adjoint programs with atomics or reductions by more than a factor of 10× on our system when executed in parallel.

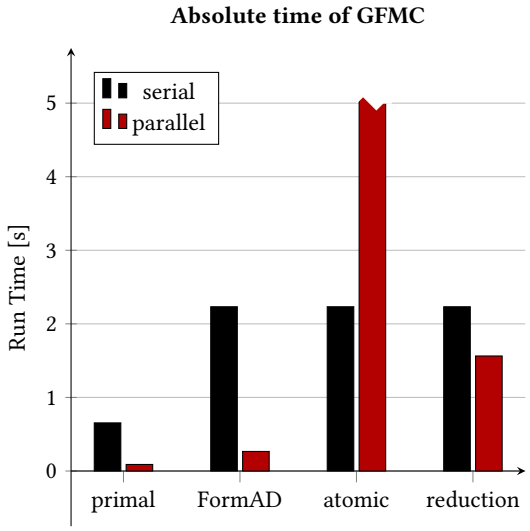

Figure 7: Absolute run times for GFMC. The sequential primal and adjoint take 0.655s and 2.23s. The FormAD adjoint performs best on 18 threads with a run time of 0.266s, whereas the adjoint with reductions performs best on 4 threads with 1.56s, which is 5.88× slower. The atomic adjoint version requires at least 33.9s.

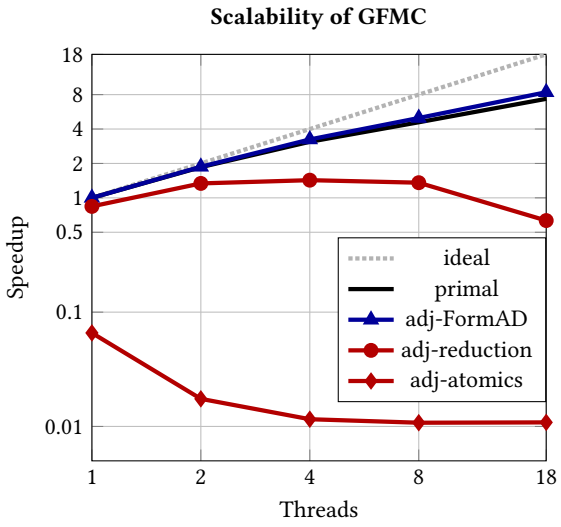

Figure 8: Parallel speedup for GFMC. The primal and FormAD adjoint achieve up to 7.35× and 8.39× respectively. The adjoint with reductions peaks at 1.43× on 4 threads. The version with atomics is between 10× and 100× slower than the serial version.

7.2 GFMC

The Green's function Monte Carlo kernel (GFMC)[2] is part of the CORAL performance benchmark suite [21]. While the original benchmark contains just one parallel loop, we create an alternative version in which we split the computation across two parallel loops, the first one containing a dynamic part with large load imbalance labeled as *spin exchange*, the other part with a more regular workload labeled as *spin flip*. We will refer to this version with two separate parallel loops as GFMC, and refer to the original version as GFMC*. We differentiate both versions by using both `cl` and `cr` as active input and output variables.

When applied to GFMC*, FormAD correctly identifies a read access during the spin exchange that yields an unsafe increment access in the adjoint, which needs to be safeguarded with an atomic pragma for parallel adjoint execution. Since both parts of the computation are inside the same parallel loop, this makes all increment accesses to the affected array potentially unsafe and requires Tapenade to guard them all.

In contrast, when applied to GFMC, FormAD can prove the absence of data races in the adjoint of the spin exchange computation, and we obtain a parallel adjoint for this part of the code that does not require atomics or reductions. Some of the most interesting memory accesses from the primal are shown in the following listing.

```
idd=mss(1,ig,k12)
iud=mss(2,ig,k12)
idu=mss(3,ig,k12)
iuu=mss(4,ig,k12)
...
cl(idd,j) = xee*cr(idd,j) + ...
cl(iuu,j) = xee*cr(iuu,j) + ...
cl(iud,j) = xmm*cr(iud,j) + ...
cl(idu,j) = xmm*cr(idu,j) + ...
```

The shared array `cr` is overwritten within the parallel loop at data-dependent indices `(idd,j)`, `(iuu,j)`, `(iud,j)`, `(idu,j)`. Assuming that the primal code contains no data races, we can conclude that the index expressions yield different values on different threads. The array `cr` is read at the same indices, which will cause its corresponding array in the adjoint code to be incremented at the same indices. We can thus conclude that the corresponding adjoint increments to `cr` are safe to perform without atomics.

We run 500 repetitions of this kernel, except for the atomic adjoint version, where we run only 5 iterations and multiply the obtained time with 100 to reduce experimentation time, since this program version is otherwise very slow. Figure 8 and 7 illustrate the run time and scalability of this part of the computation. Because this function is nonlinear, the adjoint computation requires saving and restoring intermediate values and is overall more complicated, which explains the larger run time of the adjoints compared to the primal. The FormAD adjoint scales as well as the primal and outperforms atomics or reductions by more than 5×.

7.3 LBM

This test case uses a Fortran translation of the Lattice-Boltzmann Method [4] (LBM) solver from the Parboil benchmark suite [20]. This is a structured-mesh application that operates on a three-dimensional grid of spatial points and performs a streaming operation that communicates data between neighboring grid points, as well as a collision operation that computes point-local values.

[2] https://asc.llnl.gov/coral-benchmarks#gfmcmk

FormAD (correctly) concludes that atomics or reductions are necessary, and no change to the code and thus no speedup is achieved by using FormAD. We therefore discuss this test case only briefly, as an example in which FormAD is ineffective.

The primal code contains long index expressions that are the result of a sequence of preprocessor macros. FormAD simplifies the expressions and builds a set of known safe write expressions:

```
(w_0 + n_cell_entries_0*-1 + i_0)
(se_0 + n_cell_entries_0*-119 + i_0)
(c_0 + n_cell_entries_0*0 + i_0)
(nb_0 + n_cell_entries_0*-14280 + i_0)
(s_0 + n_cell_entries_0*-120 + i_0)
(sb_0 + n_cell_entries_0*-14520 + i_0)
(eb_0 + n_cell_entries_0*-14399 + i_0)
(et_0 + n_cell_entries_0*14401 + i_0)
(nt_0 + n_cell_entries_0*14520 + i_0)
(t_0 + n_cell_entries_0*14400 + i_0)
(ne_0 + n_cell_entries_0*121 + i_0)
(b_0 + n_cell_entries_0*-14400 + i_0)
(wb_0 + n_cell_entries_0*-14401 + i_0)
(wt_0 + n_cell_entries_0*14399 + i_0)
(sw_0 + n_cell_entries_0*-121 + i_0)
(e_0 + n_cell_entries_0*1 + i_0)
(st_0 + n_cell_entries_0*14280 + i_0)
(nw_0 + n_cell_entries_0*119 + i_0)
(n_0 + n_cell_entries_0*120 + i_0)
```

At least one index expressions that is used to increment an adjoint variable is not contained in this set:

```
eb_0 + n_cell_entries_0*0 + i_0
```

FormAD thus considers the access to `srcgrid` as unsafe and does not remove any safeguards from the generated code.

7.4 Green Gauss Gradients

Computing or approximating spatial gradients, which measure the variation of a physical property in space, is an essential step in partial differential equation (PDE) solvers. A popular approach for Finite Volume based PDE solvers is given by the Green Gauss theorem, which allows computing the gradients of a volume by integrating an expression over the surface of that volume. Green Gauss gradients were also used in previous work that discussed the application of AD tools to unstructured PDE solvers[14].

Our Green Gauss Gradient implementation iterates over a set of edges in a mesh. At each edge, values from both vertices connected by this edge are retrieved and updated. To parallelize this loop over edges and avoid conflicts on the nodes, we use a coloring approach [14] to split the edges into groups that can be treated concurrently. The parallel loop looks as follows:

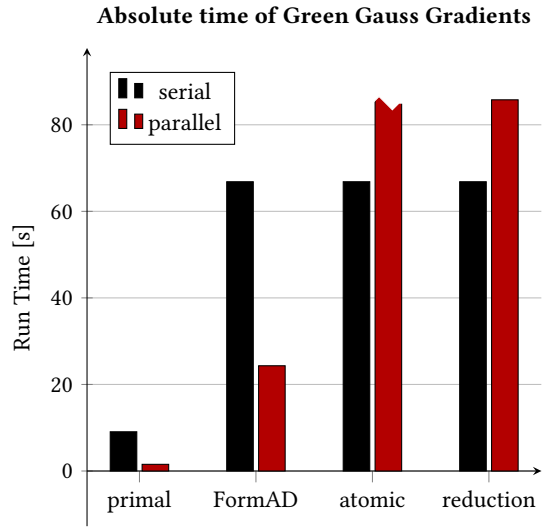

Figure 9: Absolute run times for Green Gauss Gradients. For sequential execution, the primal takes 9.064s, whereas the adjoint takes 66.84s. Using FormAD, we reduce this time to 24.32s on 18 threads (although 8 threads are nearly as fast). Meanwhile, reductions are significantly slower with their best performance on 8 threads still taking 85.77s. Atomics are slower still; even with just one thread execution time increases to 386s, and slowing down further as more threads are added.

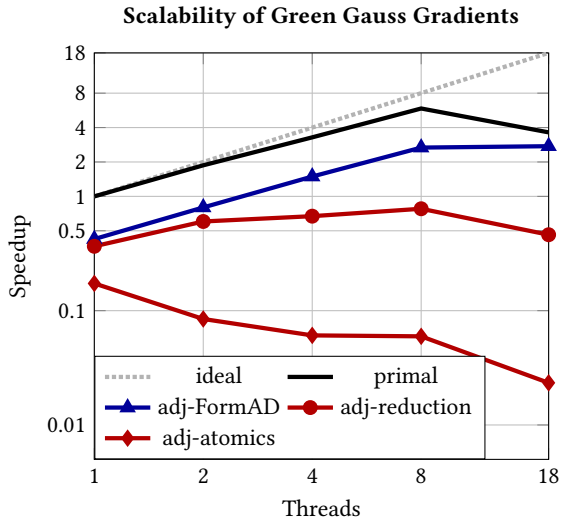

Figure 10: Parallel speedup for Green Gauss Gradients. This test case performs very few flops and has unstructured memory access, making it highly memory bound and explaining some of the overall poor scalability. Still, we observe some parallel speedup in the primal and adjoint using FormAD, while atomics and reductions perform poorly.

```
do ic=1,size(color_ia,1)-1
  !$OMP PARALLEL DO PRIVATE(ie, i, j, dvFace)
  do ie=color_ia(ic),color_ia(ic+1)-1
    i = edge2nodes(1,ie)
    j = edge2nodes(2,ie)
    if (i .ne. j) then
      dvFace=0.5d0*(dv(i)+dv(j))
      grad(i) = grad(i) + dvFace*sij(ie)
      grad(j) = grad(j) - dvFace*sij(ie)
    end if
  end do
end do
```

We note the data-dependent array indices to `grad`, which are dependent on the unstructured mesh connectivity given in `edge2nodes` and are therefore hard to analyze statically. `size(color_ia)` reflects the number of colors, and the loop bounds `color_ia(ic)` and `color_ia(ic+1)-1` reflect the first and last edge belonging to a particular color. Both depend on the input mesh as well as the coloring strategy, while `edge2nodes` reflects the two nodes connected by a given edge. Nevertheless, FormAD detects the equivalence of primal and adjoint index expressions in this test case, and produces an adjoint parallel loop without requiring safeguards.

We test the run time of this kernel by applying it 10,000 times to a mesh with 100,000 nodes. Our test mesh was generated with a simple, linear structure requiring only 2 colors to simplify the experimental setup. The resulting absolute times and scalability are shown in Figures 9 and 10. FormAD-generated adjoints are faster than sequential by a factor of 2.75, whereas the other approaches we tested can not achieve any parallel speedup.

7.5 Performance of FormAD analysis

Run times and relevant statistics for the differentiation aided by FormAD are shown in Table 1. In all our test cases, FormAD takes less than 5 seconds to run. The number e of unique index expressions detected in the input code is between 2 and 19. As expected, the number of assertions that are sent to the Z3 theorem prover is $1+e^2$, which in our test cases is between 5 and 362. A higher number of expressions tends to also increase the number of queries that is sent to Z3, as well as the run time of FormAD. The analysis can stop and indicate a potential conflict as soon as an unsafe index expression is found in the adjoint, whereas a response indicating safe (disjoint) accesses requires exploring the full set of index expressions. This explains why the GFMC test case (which is safe to parallelize without atomics) takes more time and more queries than the GFMC* and LBM test cases (which are both rejected as unsafe). FormAD is run only once during compile time, which means that even significantly longer times in larger test cases might be acceptable since they can be amortized over many executions of the optimized programs. It may become necessary to offer a user-configurable timeout in future implementations if Z3 fails to answer certain queries in a reasonable time.

problem	time	Z3 size	queries	exprs	loc
stencil 1	0.677	5	3	2	3
stencil 8	1.033	82	82	9	17
GFMC	4.145	65	772	8	54
GFMC*	3.125	65	261	8	65
LBM	3.938	362	364	19	82
GreenGauss	0.621	5	3	2	7

Table 1: Execution time in seconds for FormAD, size (number of assertions) of the generated model, number of queries answered by the proof system, number of unique index expressions included in the model, and number of lines of code within the parallel region that was analyzed. Note that the statistics are from a (now replaced) previous implementation, which wrote the knowledge to a file and used a standalone Python tool to call Z3, which made it easier to obtain timings for just the Z3 part.

8 CONCLUSION

We presented FormAD, a new type of static analysis plugin for AD tools that is specifically designed for reverse-mode automatic differentiation of shared-memory parallel programs. By harnessing information from the assumed-correct parallelization of the original program, we can significantly improve the run time and parallel speedup of generated derivative programs.

Our method and implementation could be improved in various ways in the future, including support for input languages other than Fortran. We also hope to apply this method to more, and larger, applications in the future. Finally, our experiments are all using OpenMP input programs. We postulate that the method should in principle also apply to other shared-memory-parallel systems including GPUs, where avoidance of reductions or atomic updates could be even more beneficial. Investigating this is future work.

ACKNOWLEDGEMENTS

This work was supported by the Applied Mathematics activity within the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under contract number DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided on Bebop, a high-performance computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

REFERENCES

[1] Gilbert Bernstein, Michael Mara, Tzu-Mao Li, Dougal Maclaurin, and Jonathan Ragan-Kelley. 2020. Differentiating a Tensor Language. *arXiv preprint arXiv:2008.11256* (2020).

[2] Christian Bischof, Niels Guertler, Andreas Kowarz, and Andrea Walther. 2008. Parallel Reverse Mode Automatic Differentiation for OpenMP Programs with ADOL-C. In *Advances in Automatic Differentiation*, Christian H. Bischof, H. Martin Bücker, Paul D. Hovland, Uwe Naumann, and J. Utke (Eds.). Springer, 163–173. https://doi.org/10.1007/978-3-540-68942-3_15

[3] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. 2015. Polyhedral optimizations of explicitly parallel programs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 213–226.

[4] Shiyi Chen and Gary D Doolen. 1998. Lattice Boltzmann method for fluid flows. *Annual review of fluid mechanics* 30, 1 (1998), 329–364.

[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[6] Michael Förster. 2014. *Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP*. Ph.D. Dissertation. RWTH Aachen.

[7] Ralf Giering and Thomas Kaminski. 1996. *Recipes for Adjoint Code Construction*. Technical Report 212. Max-Planck-Institut für Meteorologie. http://www.mpimet.mpg.de/en/web/science/a_reports_archive.php?actual=1996

[8] Ralf Giering, Thomas Kaminski, Ricardo Todling, Ronald Errico, Ronald Gelaro, and Nathan Winslow. 2005. Tangent Linear and Adjoint Versions of NASA/GMAO's Fortran 90 Global Weather Forecast Model. In *Automatic Differentiation: Applications, Theory, and Implementations*, H. Martin Bücker, George F. Corliss, Paul D. Hovland, Uwe Naumann, and Boyana Norris (Eds.). Springer, 275–284.

[9] A. Griewank and A. Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (2nd ed.). Number 105 in Other Titles in Applied Mathematics. SIAM, Philadelphia, PA. http://www.ec-securehost.com/SIAM/OT105.html

[10] L. Hascoët and V. Pascual. 2013. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Softw.* 39, 3 (2013), 20:1–20:43. https://doi.org/10.1145/2450153.2450158

[11] Jan Hückelheim and Johannes Doerfert. 2021. Spray: Sparse Reductions of Arrays in OpenMP. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 475–484. https://doi.org/10.1109/IPDPS49936.2021.00056

[12] J. Hückelheim and L. Hascoët. 2021. Source-to-Source Automatic Differentiation of OpenMP Parallel Loops. *ACM Trans. Math. Softw.* accepted for publication (2021).

[13] J.C. Hückelheim, P.D. Hovland, M.M. Strout, and J.-D. Müller. 2018. Parallelizable adjoint stencil computations using transposed forward-mode algorithmic differentiation. *Optimization Methods and Software* 33, 4-6 (2018), 672–693. https://doi.org/10.1080/10556788.2018.1435654 arXiv:https://doi.org/10.1080/10556788.2018.1435654

[14] Jan Hückelheim, Paul D. Hovland, Michelle Mills Strout, and Jens-Dominik Müller. 2017. Reverse-mode algorithmic differentiation of an OpenMP-parallel compressible flow solver. *International Journal for High Performance Computing Applications* (2017).

[15] Jan Hückelheim, Navjot Kukreja, Sri Hari Krishna Narayanan, Fabio Luporini, Gerard Gorman, and Paul Hovland. 2019. Automatic Differentiation for Adjoint Stencil Loops. In *Proceedings of the 48th International Conference on Parallel Processing* (Kyoto, Japan) *(ICPP 2019)*. Association for Computing Machinery, New York, NY, USA, Article 83, 10 pages. https://doi.org/10.1145/3337821.3337906

[16] Tim Kaler, Tao B. Schardl, Brian Xie, Charles E. Leiserson, Jie Chen, Aldo Pareja, and Georgios Kollias. 2021. PARAD: A Work-Efficient Parallel Algorithm for Reverse-Mode Automatic Differentiation. In *Proceedings of the Symposium on Algorithmic Principles of Computer Systems (APOCS)*, Schapira Michael (Ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, 144–158. https://doi.org/10.1137/1.9781611976489.11 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611976489.11

[17] Benjamin Letschert, Kshitij Kulshreshtha, Andrea Walther, Duc Nguyen, Assefaw Gebremedhin, and Alex Pothen. 2012. Exploiting Sparsity in Automatic Differentiation on Multicore Architectures. In *Recent Advances in Algorithmic Differentiation*, Shaun Forth, Paul Hovland, Eric Phipps, Jean Utke, and Andrea Walther (Eds.). Lecture Notes in Computational Science and Engineering, Vol. 87. Springer, Berlin, 151–161. https://doi.org/10.1007/978-3-642-30023-3_14

[18] Uwe Naumann. 2012. *The art of differentiating computer programs: an introduction to algorithmic differentiation*. Vol. 24. SIAM.

[19] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2014. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 65–76.

[20] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-mei W Hwu. 2012. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing* 127 (2012).

[21] Sudharshan S Vazhkudai, Bronis R de Supinski, Arthur S Bland, Al Geist, James Sexton, Jim Kahle, Christopher J Zimmer, Scott Atchley, Sarp Oral, Don E Maxwell, et al. 2018. The design, deployment, and evaluation of the CORAL pre-exascale systems. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 661–672.