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

Automatic differentiation (AD) is a technique to obtain derivatives of mathematical functions implemented in a given computer program. Reverse mode AD or backpropagation are efficient approaches for programs with many parameters, and are key components of machine learning, inverse design, data assimilation, and similar methods. This paper presents a novel combination of reverse mode AD and formal methods, which enables efficient differentiation of (or backpropagation through) shared-memory parallel loops. Compared to the state of the art, our approach can more often avoid the need for atomic updates or private data copies during the parallel derivative computation. This is achieved by gathering information about the memory access patterns from the given 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 computation. By leveraging information from the parallelized input program, we can automatically parallelize derivative computations even in the presence of unstructured or data-dependent data access patterns. We demonstrate this approach on scientific computing benchmarks.

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 write (or increment) accesses in the derivative or adjoint 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 scale less well than the primal program.

Some 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, machine learning frameworks, and similar contexts, but does not always offer the flexibility that is needed for scientific computing applications.

Instead, our work attempts to improve the code analysis of general-purpose AD tools, by introducing a novel static analysis approach. We implemented this approach in a new tool called "FormAD", which is developed as a plugin that is called by the existing AD tool Tapenade, and will be publicly released under the same MIT license as Tapenade. Our approach combines formal methods and automatic differentiation to more often avoid atomic updates and reductions. Unlike previous tools, FormAD is specifically designed to speed up the derivatives of programs with unstructured and data-dependent memory access patterns, by exploiting the relationship of primal programs with their corresponding reverse-mode AD or backpropagation computations.

2 Related Work

Applying automatic differentiation to shared-memory parallel programs has been discussed extensively in the past. For example, [12] 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 [11, 13] 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.

PPoPP'21, Feb 12–16, 2021, Seoul, South Korea
2021.

There is previous work discussing OpenMP support for AD tools, such as [2, 6, 7, 16], although it has been observed that parallel programs can be challenging to reverse-differentiate [1, 5], and that static analysis may be needed to ascertain the absence of increment-increment conflicts [5], which does not always succeed. Other papers use or discuss atomic or privatization/reduction constructs as a safe fallback [2, 7, 15].

3 Contributions and Limitations

To our knowledge, we make the following novel contributions.

- We present the first combination of an AD tool with a theorem prover to improve parallel efficiency of generated adjoint programs.
- We present the first tool that automatically detects the relationship of memory access patterns in a primal and adjoint program, with the same goal.
- Compared to previous work, our theory extends beyond simple symmetry relationships, and is not restricted to particular access patterns such as stencils.
- We present test cases where Tapenade with the FormAD plugin beats the parallel speedup of 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. We expect C to be a straightforward extension if the parser is changed and scoping rules adjusted slightly.
- FormAD is a prototype tool, and is known to break on some input programs due to limitations of the Fortran parser and our interaction with the parser. For example, we only support single-line OpenMP pragmas (but impose no length limit), and require that a true statement immediately precedes the OpenMP pragma.
- We assume no-alias, and that multi-dimensional arrays are used "correctly", i.e. without accessing consecutive rows/columns by going out of bounds in the current row/column.
- 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 almost never truly benign, but are expected by some programmers to succeed. FormAD is designed to be truly "garbage-in, garbage-out", and may in such cases deduce parallelizations that break more frequently or more visibly 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 aim to present the subset of Automatic Differentiation theory and the notations used in the rest of the paper. A complete introduction to AD can be found in e.g. [8] or [6]. Source-Transformation AD transforms a given source code (the "primal" code) into a new code that computes derivatives of the given code's outputs with respect to its inputs. Adjoint AD is one AD strategy which is particularly efficient when the given code has many inputs and few outputs. It is therefore the preferred strategy to compute gradients. To name a few, applications range from shape optimization (where many inputs define the geometry to optimize, and one or a few outputs define the cost function(s) to optimize), inverse problems (inputs being the parameters to estimate, output being the measure of discrepancy), to Machine Learning (inputs being the learned coefficients, output being the estimation error). In this last case, Adjoint AD is known as Backpropagation.

The efficiency of Adjoint AD has been documented many times and we will refer the reader to the AD literature e.g. [8] for full justification. This efficiency comes from its reverse nature, that propagates gradients of the primal code's final output with respect to each intermediate variable of the computation, therefore backwards, until it finally obtains the gradient of the final output with respect to the program's inputs. Consequently, while ordinary AD ("Tangent AD") or similarly Finite Differences must compute the gradient by running once for each component of the input, Adjoint AD computes all components of the gradient in just one run. For the applications we mentioned earlier, this tremendous efficiency clearly outweighs the added difficulties inherent to its reverse nature.

For each instruction performed by the primal code, the adjoint code must perform corresponding adjoint instructions, which we are going to describe. These adjoint instructions must be executed in reverse order with respect to the primal code. Moreover, they often need to use the values that were used by the corresponding primal instruction, and therefore a form of communication must be organized from the primal code to the adjoint code, both to schedule the adjoint instructions in reverse order (*control-flow reversal*), and to provide them with the intermediate values of needed variables (*data-flow reversal*). These reversals can be quite complex and have been the subject of many years of AD research. However, they do not directly interfere with this work and we refer to [10] for a fuller description. The problem that we address here occurs strictly inside the *backward sweep*, i.e. the collection of the adjoint instructions, and is not related to the communication machinery from the primal code (the *forward sweep*) to the backward sweep.

Consider a representative statement (here an assignment) of the primal code, here in a slightly abstracted form:

```
u(i-1) = a*v(i,j) + 1.5
```

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

```
u(2*i) = u(2*i) + 2*a
```

```
ab = ab + 2*ub(2*i)
```

Figure 1. Adjoint instructions for an arbitrary assignment (left), and simplified adjoint instruction for an increment assignment (right). In real code, we name adjoint variables after the primal variable with "b" appended (read as "bar").

$$z = x\ Op\ y$$

with Op any differentiable (here binary) operation, and x, y, and z any variable references, possibly with indices or other access patterns. Let us assume that the memory location of z does not overlap with any variable in the right-hand side. If not, just introduce a temporary variable and split the assignment. The adjoint instructions for this assignment is the following sequence:

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

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 $\{x, y, z\}$ before to $\{x, y, z\}$ after the instruction, and to multiply it *on the right* of the gradient vector $(\overline{x}, \overline{y}, \overline{z})$. We will rather give an informal interpretation of these adjoint instructions: $\overline{x}$ is the influence of the current x on the final output. This influence is the sum of contributions from each place where this 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.

Figure 1 shows two concrete examples of adjoint instructions of an assignment, including the interesting 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 focus on thread-parallel codes. In the sequel, we will use the OpenMP formalism but we believe all can be transposed to other thread-parallel formalisms. A general study of Adjoint Source-Transformation AD of thread-parallel codes can be found in [10]. We will summarize it here only to introduce the biggest difficulty: the transformation of *read-read* situations on shared variables in the primal code into *increment-increment* conflicts in the adjoint code.

In our Source-Transformation model, the adjoint code consists basically of two successive sweeps, a *forward sweep* that runs a copy of the primal code augmented with storage of data for control-flow and data-flow reversal, and a *backward sweep* that runs the adjoint instructions, in the intended reverse order, and that progressively reads when needed the data stored in the forward sweep. Obviously now, the data structure used for storage is basically a stack. For each parallel region of the primal code, two new parallel regions are created, one for each sweep.

The new parallel region of the forward sweep is essentially a copy of the primal code, and can therefore retain all its inside parallel structure, as well as the variable distribution directives, called *scopings*, i.e. `private, firstprivate, lastprivate, reduction`, or `shared`. The stack used for communication between sweeps must be declared `threadprivate`.

The corresponding region in the backward sweep can also be declared parallel. Variables from the primal code still exist in the backward sweep, and it was shown that they can keep their original scoping. In addition, the backward sweep must declare and manipulate *adjoint* variables to hold the derivatives. The nontrivial question is the scoping one must attach to these adjoint variables to preserve correctness and hopefully parallel efficiency.

After analysis, it turns out that for most scopings, we are able to provide an adequate scoping for the adjoint variable which preserves the parallel efficiency [10]. This is true for the `private`, `firstprivate`, `lastprivate`, and `reduction(+)` scopings. Reductions other than `(+)` generally require specific treatment for AD, out of the scope of this work. They are fortunately quite infrequent in applications, the only other differentiable reduction being `(*)`.

The situation is more delicate for the `shared` scoping. We still can provide a correct scoping for the adjoint variables, but often at the cost of a degraded parallel efficiency. Apart from favorable situations that are generally hard to detect, a frequent case is when `shared` variables are accessed only for reading, but possibly concurrently by several threads, 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 a *increment-increment* conflict. Two increments of the same variable may be executed in parallel only if increment operations are *atomic*.

Unless we can prove the absence of a *read-read* situation on the primal variable v, we have only two correct options for the adjoint variable $\overline{v}$:

- declare $\overline{v}$ as `shared`, but place an `atomic` pragma on each instruction that increments $\overline{v}$.

```
!$omp parallel do
!$omp+ shared(u),
!$omp+ shared(r)
do i=2,2*n,2
  r(i-1) = 2*u(i)
  r(i) = 3*u(i-1)
end do
```

```
!$omp parallel do
!$omp+ shared(ub), shared(rb)
do i=2*n,2,-2
  ub(i-1) = ub(i-1) + 3*rb(i)
  rb(i) = 0.0
  ub(i) = ub(i) + 2*rb(i-1)
  rb(i-1) = 0.0
end do
```

Figure 2. A simple parallel loop (left), and its corresponding adjoint loop (right). Adjoint variables are named after the primal variable with "b" appended. The reason why ub and rb can here be shared is the subject of this work.

- privatize $\overline{v}$ as a `reduction(+)`,

Both options harm performance. Even though experimental libraries have been proposed [14] to reduce the cost of this particular reduction operation, it is nevertheless profitable to detect sheer absence of *read-read* situations in the primal code.

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. Although this option should not be turned down, one must be aware of its limitations as a static and therefore approximate analysis. It is also limited when array accesses are irregular, using dynamic indirection methods.

We would like to note that some favorable cases allow for elaborate transformations to maintain a `shared` scoping for the adjoint variable. For regular accesses on a regular grid, one may shift adjoint iterations, realigning them on the adjoint increment to eliminate the *increment-increment* conflicts. This is application-dependent.

We propose to exploit another information source to detect the absence of *read-read* situations: 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 the same parallel loop nest, at least one of these references being a *write*, we know that the array indexes used are disjoint and cannot lead to a loop-carried dependency. 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 reuse it for any other array accessed with the same indices in any parallel loop nest of same iteration space, and in particular we can use it for the adjoint arrays in the adjoint parallel loop nests.

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 `r`. In other words for any pair of loop indices i_1 and i_2, $i_1 \neq i_2$, we know `r(`i_1`-1)` and `r(`i_2`)` can not overlap., i.e. $i_1 - 1 \neq i_2$ This knowledge binds the iteration space of the parallel loop to the array offsets defined by the array indices, independently of the particular array `r` itself. We can therefore use this knowledge to show that there is no loop-carried dependency in the adjoint loop due to accesses to ub nor to accesses to rb. As a consequence, ub and rb can be declared `shared` in the adjoint loop.

For each parallel region, this results in two phases:

1. **Knowledge extraction:** for each `shared` variable (actually array) in the given parallel region, we collect all read references and all write references. For each pair of such references, at least one being a write, we extract the index expression(s), and add into the knowledge base for this region that these indices are disjoint, i.e. they have a different value if loop counters differ. The knowledge base must handle the case where some variables appearing in the index expressions are modified in the region: this will be detailed in section 4.4.
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 consider the index expressions and ask the knowledge base to prove that these indices have a different value when loop counters differ. Note that by definition the index expressions of an adjoint reference are the same as those of its primal. If for all pairs of references considered, we could 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 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.

We will now focus on a few key points of the method, and then discuss the proof system for our special-purpose data-dependence analysis.

4.3 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 of the given code. If two instructions have the same context, then a given iteration of the parallel loop nest will always execute both instructions, or none of them. Concretely, the top-level of the body of the parallel region is labeled with a root context. Then recursively control-flow structure gives birth to a new context that is said *included* inside the context of the control structure itself. Specifically if a loop has context `C` as a whole, then top-level instructions in its body receive a new context `C.body` included in `C`. If a conditional statement has context `C`, then the top-level instruction of its *then* branch receive a new context `C.then`, and similarly `C.else` for the *else* branch.

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 because we cannot be sure that some control 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 $\overline{R_2}$.

4.4 Distinguishing overwritten variables in indexes

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. Anyway the loop index deserves a special treatment natively in our proof system. Other variables may be modified in the loop and cannot be compared 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.

4.5 Taking advantage of the AD model

We can take advantage of the AD model 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. More interestingly, AD generally comes with a special *activity* analysis, that detects locations where the derivative of a variable is trivially zero or is nontrivial but useless. Due to the reverse nature of adjoint AD, this may happen in two cases:

- the current variable has no differentiable influence on the output we want to differentiate. For instance, it is about to be reset to a new value. In this case the adjoint derivative, partial derivative of the said output with respect to current variable, is just zero.
- the current variable does not depend in a differentiable way on the input with respect to which we want to differentiate. For instance it was just reinitialized to a constant value. In that case the adjoint derivative may be nontrivial, but it will not be used and therefore need not be computed.

Thanks to activity analysis, only a subset of the variable references of the given parallel region give birth to an adjoint variable reference in the adjoint region. This increases the chances of discovering loop-carried independence.

Even further, we can detect situations where an instruction of the primal code exactly increments a variable reference. As shown on the right of figure 1, the corresponding adjoint variable will only be read in the adjoint instructions. This is the only situation where the adjoint variable is not overwritten. This is useful as the only pairs of references for which we need to prove loop-carried independence are pairs with at least one write. Turning a write into a read reduces the number of reference pairs to consider.

4.6 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 the loop counter i_1 on one side is replaced with the loop counter i_2. The recursive procedure is started at the root context, corresponding to the body of

the parallel loop, with the assertion $i_1 \neq i_2$, which expresses the fact that no two threads may have the same loop counter value, as is guaranteed by OpenMP.

In simplified pseudo code, this procedure is as follows:

```
buildModel(rootContext, i0 != i1)

def buildModel(context, assertions):
  model = z3.Solver()
  model.add(assertions)
  for var in variables:
    writeexprs = var.writeexprs(context)
    for expr0 in writeexprs:
      expr0s = expr0.substitute(i0, i1)
      for expr1 in writeexprs:
        model.add(expr0s != expr1)
        assert(model.check() == SAT)
  for child in context.children:
    buildModel(child, model)
```

The fact that each expression is combined with every other expression results in a model size that is quadratic in the number of unique write 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 added this as a safeguard because a failing assertion means that the write 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. If run time becomes an issue for larger test cases, this assertion could be checked less frequently.

In the knowledge exploitation phase, we need to test the set of potentially 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 scope is satisfiable. By adding a pair of expressions from the set of candidate expressions and asserting its equality, the model will become unsatisfiable if the expressions are provably disjoint. If the model will remain 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 entire candidate expression set from which this pair was taken is unsafe. For a given model (associated with a context) and a candidate expression set, this can be written as follows:

```
def checkExpressions(model, checkexprs):
  for expr0 in checkexprs:
    expr0s = expr0.substitute(i0, i1)
    for expr1 in checkexprs:
      model.push()
      model.add(expr0 == 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 popping the previous state from the stack.

5 Implementation

We implement our prototype as a collaboration between a special-purpose code analysis written in Python for knowledge extraction, the Source-Transformation AD tool Tapenade [9], and our proof system based on the Z3 [4] tool. In our implementation, the knowledge extraction phase is done by the Python tool. So are the identification of *contexts* (*cf* section 4.3) and of *instances* (*cf* section 4.4). This could also have been done within the AD tool itself, but Python was chosen for ease of development. The Python tool uses the fparser2 and Z3Py packages for Fortran parsing and theorem proving.

The collection of possible *increment-increment* conflicts on adjoint shared variables is done within the AD tool, using its activity analysis to reduce the number of possible conflicts (*cf* section 4.5). The AD tool then sends this set of possible conflicts to the proof system, and expects in return the list of adjoint variables for which all possible conflicts have been proven nonexistent. Finally upon creation of the differentiated code, these conflict-free adjoint variables are simply declared as `shared` with no `atomic` pragma added. Communication between the three tools use line numbers in the primal code for synchronization.

6 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 6.4, 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 change in performance compared 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 Atomic, generated by reverse-mode AD using `atomic` pragmas (atomics) to safeguard increments to shared arrays.

Adjoint FormAD, generated by reverse-mode AD where FormAD has been used to avoid atomics whenever it is safe.

Adjoint Reduction, generated by reverse-mode AD using a reduction clause to safeguard 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.

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 OpenMP 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 could 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.

6.1 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 [17]. The "compact" scheme introduced in this 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:

```
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)
```

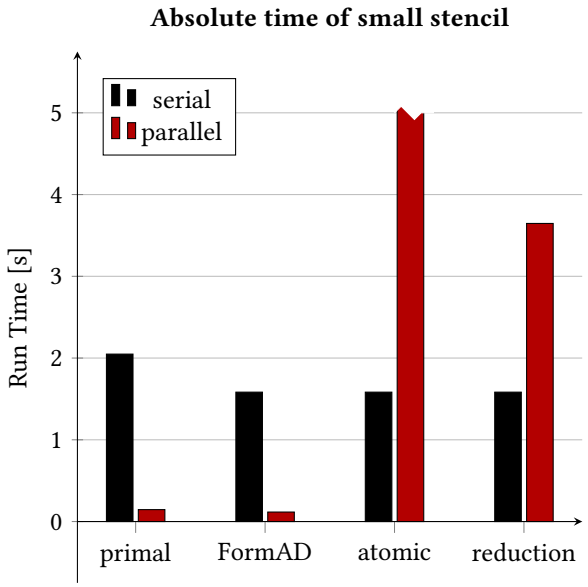

Figure 3. Absolute run time for small stencil test case. The primal serial program takes 2.05*s*, whereas the parallel version takes 0.146*s*. The best-performing thread count for the parallel adjoint programs was 1, with 40.7*s* for the atomic and 3.65*s* for the reduction version, both much worse than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

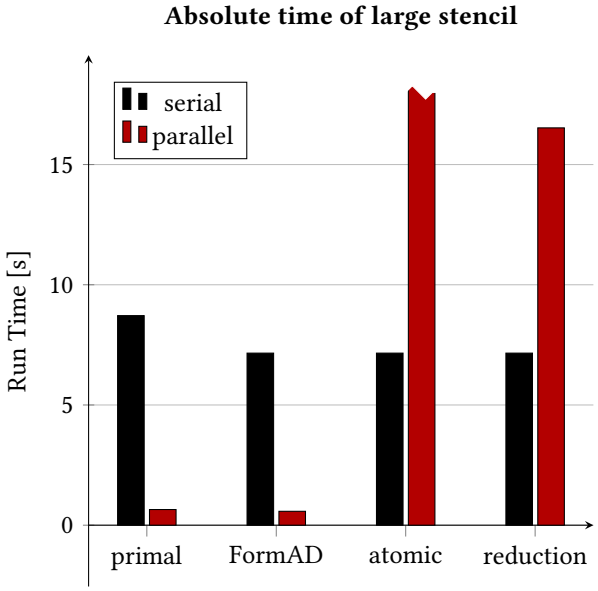

Figure 4. Absolute run time for large stencil test case. The primal serial program takes 8.72*s*, whereas the parallel version takes 0.651*s*. The best-performing thread count for the parallel adjoint programs was 1, with 95.8*s* for the atomic and 16.5*s* for the reduction version, both much worse than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

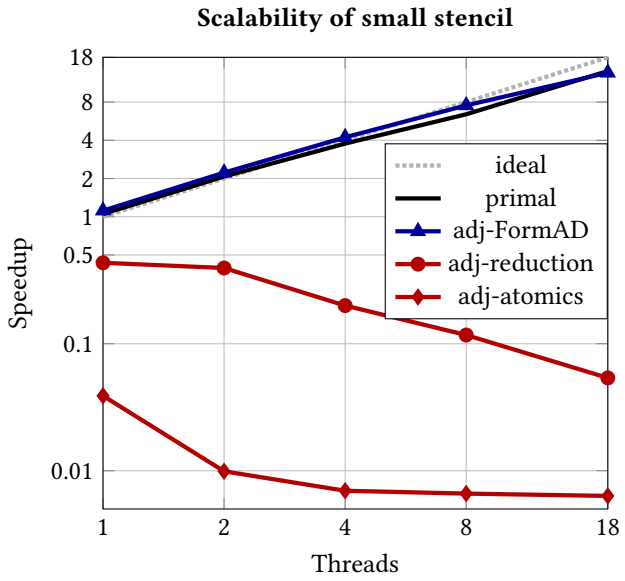

Figure 5. Parallel speedup for small stencil test case. The primal and FormAD programs scale well, with a maximum speedup on 18 threads of 13.4× and 13.6× respectively. The adjoint programs with atomics and reductions never exceed the serial performance, and actually slow down as more threads are added.

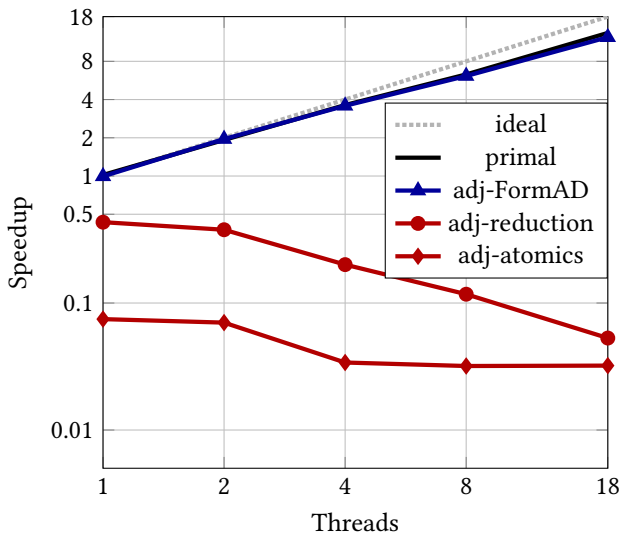

Figure 6. Parallel speedup for large stencil test case. The primal and FormAD programs scale well, with a maximum speedup on 18 threads of 13.12× and 12.4× respectively. The adjoint programs with atomics and reductions never exceed the serial performance, and actually slow down as more threads are added.

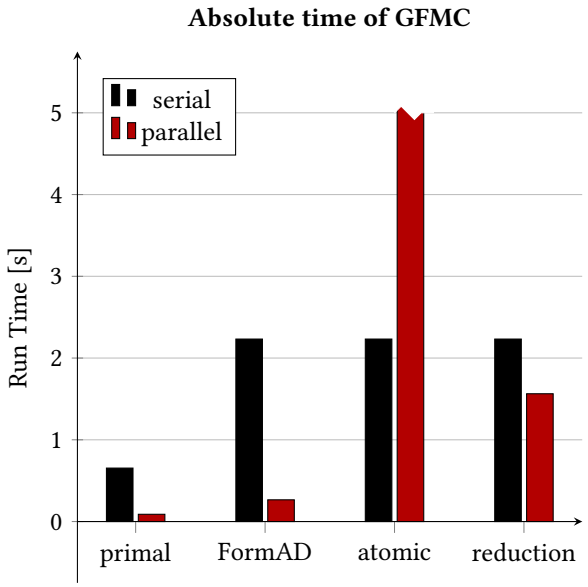

Figure 7. Absolute run times for GFMC. The sequential primal and adjoint take 0.655s and 2.23s. The FormAD adjoint performs best on 18 threads with a run time of 0.266s, whereas the adjoint with reductions performs best on 4 threads with 1.56s, which is 5.88× slower. The atomic adjoint version requires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

```
    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.

6.2 GFMC

The Green's function Monte Carlo kernel (GFMC)[1] is part of the CORAL performance benchmark suite [19]. While the

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

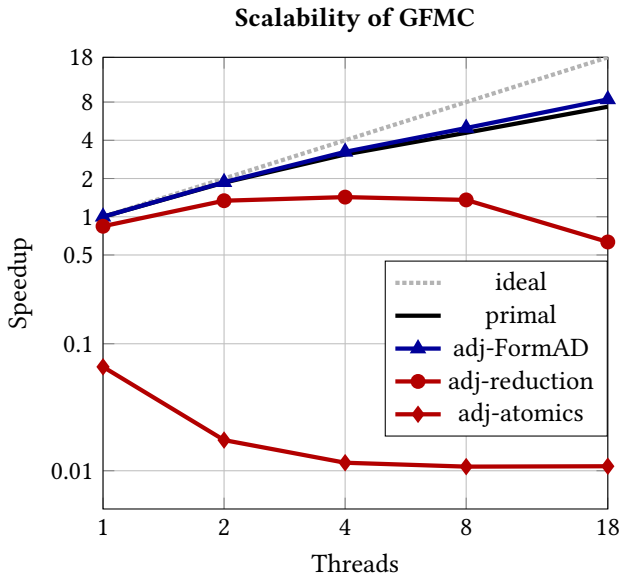

Figure 8. Parallel speedup for GFMC. The primal and FormAD adjoint achieve up to 7.35× and 8.39× respectively. The adjoint with reductions peaks at 1.43× on 4 threads. The version with atomics is between 10× and 100× slower than the serial version.

original benchmark contains just one parallel, 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 prevents FormAD from performing any beneficial optimizations.

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 safeguards.

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 the adjoints using atomics or reductions by more than 5×.

6.3 LBM

This test case uses a Fortran translation of the Lattice-Boltzmann Method [3] (LBM) solver from the Parboil benchmark suite [18]. 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 local values on each point.

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)
```

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

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.

```
(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)
```

This set does not contain at least one of the index expressions that is used to increment an adjoint variable:

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

For this reason, FormAD reports the adjoint for `srcgrid` as unsafe to parallelize without atomics, and the generated adjoint code contains the same safeguards as without the use of FormAD.

6.4 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. FormAD can return with a response indicating a potential conflict as soon as an unsafe index expression is found in the adjoint, whereas a response indicating safe (disjoint) accesses generally requires exploring the full set of index expressions. This is explains why the GFMC test case (which is safe) takes more time and more queries than the GFMC* and LBM test cases (which are both unsafe).

7 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, as shown in our test cases.

Our method and implementation could be improved in various ways in the future, including support for input languages other than Fortran, or tighter coupling between the analysis and AD tool. One could even integrate both tools into the same compiler framework, which would allow leveraging even more information sources or code transformations and to make more input programs amenable to our method. 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 left to 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] Shiyi Chen and Gary D Doolen. 1998. Lattice Boltzmann method for fluid flows. *Annual review of fluid mechanics* 30, 1 (1998), 329–364.

[4] 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.

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

[6] 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

[7] 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.

[8] 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

[9] 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

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

[11] 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

[12] 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).

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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.

[18] 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).

[19] 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.