Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation

William S. Moses
MIT CSAIL
Cambridge, MA
wmoses@mit.edu

Sri Hari Krishna Narayanan
Argonne National Laboratory
Lemont, IL
snarayan@anl.gov

Ludger Paehler
Technical University of Munich
Munich, Germany
ludger.paehler@tum.de

Valentin Churavy
MIT CSAIL
Cambridge, MA
vchuravy@mit.edu

Michel Schanen
Argonne National Laboratory
Lemont, IL
mschanen@anl.gov

Jan Hückelheim
Argonne National Laboratory
Lemont, IL
jhuckelheim@anl.gov

Johannes Doerfert
Argonne National Laboratory
Lemont, IL
jdoerfert@anl.gov

Paul Hovland
Argonne National Laboratory
Lemont, IL
hovland@mcs.anl.gov

***Abstract*—Derivatives are key to numerous science, engineering, and machine learning applications. While existing tools generate derivatives of programs in a single language, modern parallel applications combine a set of frameworks and languages to leverage available performance and function in an evolving hardware landscape.**

We propose a scheme for differentiating arbitrary DAG-based parallelism that preserves scalability and efficiency, implemented into the LLVM-based Enzyme automatic differentiation framework. By integrating with a full-fledged compiler backend, Enzyme can differentiate numerous parallel frameworks and directly control code generation. Combined with its ability to differentiate any LLVM-based language, this flexibility permits Enzyme to leverage the compiler tool chain for parallel and differentiation-specific optimizations.

We differentiate nine distinct versions of the LULESH and miniBUDE applications, written in different programming languages (C++, Julia) and parallel frameworks (OpenMP, MPI, RAJA, Julia tasks, MPI.jl), demonstrating similar scalability to the original program. On benchmarks with 64 threads or nodes, we find a differentiation overhead of $3.4 - 6.8\times$ on C++ and $5.4 - 12.5\times$ on Julia.

***Index Terms*—automatic differentiation, MPI, OpenMP, Tasks, compiler, LLVM, hybrid parallelization, parallel, distributed, C++, RAJA, Julia, Enzyme**

I. Introduction

Derivatives are at the core of many modern applications in science and engineering, such as machine learning [1], gradient-based optimization [2], [3], inverse problems [4], and computer graphics [5]. Automatic differentiation (AD) is a method for the automatic generation of derivatives of mathematical functions implemented in computer programs. AD is able to compute derivatives accurately to machine precision, unlike finite difference approaches.

Parallel computation, using a variety of frameworks, has become the de facto standard for large-scale computing and machine learning applications. This commonly involves using parallel dialects and frameworks such as the Message Passing Interface (MPI) [6] to provide distributed parallelism, or OpenMP [7] and Julia tasks [8] for shared-memory parallelism, as well as higher-level frameworks such as RAJA [9].

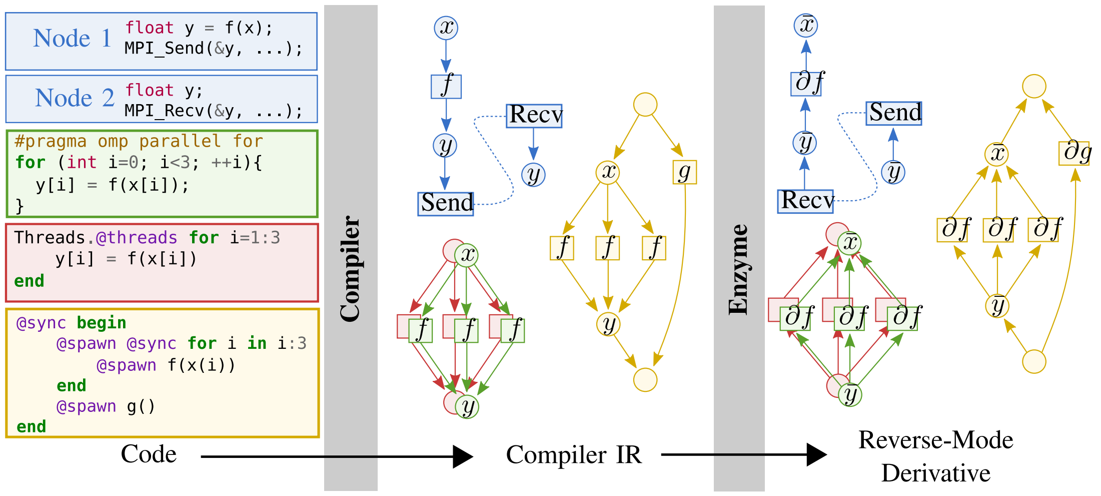
In addition to being difficult to create any derivatives of parallel programs, it is desirable to preserve the original program's parallelism for the accumulation of derivatives. This is not always straightforward, particularly in the so-called reverse-mode AD or the closely related back-propagation [10]–[15], which will be briefly explained in Section III.

This paper demonstrates how using a common low-level compiler infrastructure to synthesize adjoints of parallel codes enables differentiation across a wide variety of parallel models and source languages. To this end, we extend the Enzyme automatic differentiation framework [16], which already supports synthesizing adjoints of GPU kernels [17] to arbitrary parallel frameworks representable as a directed acyclic graph (DAG) of dependencies. To showcase the generality of our approach, we differentiate MPI (distributed parallelism), OpenMP (multicore parallelism), Julia Tasks (multicore parallelism within a JIT), and describe how additional frameworks can be supported by simply marking the parallelism.

By enabling support for the underlying programming models within the compiler, we are able to differentiate any parallel framework built on top of them such as RAJA (running atop OpenMP and MPI) and MPI.jl (Julia bindings for MPI). Moreover, we demonstrate that differentiating low-level parallelism concepts such as shared and thread-local memory automatically yields support for higher-level primitives such as reductions or `firstprivate` variables. Finally, we showcase how jointly supporting these parallelism models in one tool naturally enables differentiation of hybrid parallel programs, and that deep integration of AD into the compiler enables performance optimizations usually only available in domain-specific/functional programming languages. Overall, our paper makes the following contributions:

- An extension to the theory of reverse-mode differentiation of single-static-assignment (SSA) intermediate representations to handle parallel execution of instructions, and thus differentiation of parallel languages and constructs that lower to such a representation.
- A demonstration of how implementing this model within the

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE



```
float y = f(x);
MPI_Send(&y, ...);
```

```
float y;
MPI_Recv(&y, ...);
```

```
#pragma omp parallel for
for (int i=0; i<3; ++i){
  y[i] = f(x[i]);
}
```

```
Threads.@threads for i=1:3
    y[i] = f(x[i])
end
```

```
@sync begin
    @spawn @sync for i in i:3
        @spawn f(x(i))
    end
    @spawn g()
end
```

Fig. 1: The compiler lowers the various parallel programming languages (left) into a common representation (center). Some constructs such as Julia tasks and OpenMP worksharing loops may result in an almost identical representation. The automatic differentiation rules in Enzyme can be written for the intermediate representation, greatly simplifying the generation of reverse-mode derivatives (right) for the input languages and constructs, and further enabling compiler-optimizations.

Enzyme AD engine enables end-to-end, automated reverse-mode differentiation of parallel constructs (OpenMP, MPI, RAJA, Julia Tasks, etc) written in an LLVM-compatible language (C/C++, Julia, Fortran, Swift, Rust, Python, etc).

- Experimental results for codes from the LULESH [18] benchmark suite written in C++/OpenMP, C++/MPI, C++/MPI+OpenMP, C++/RAJA, and Julia/MPI.jl and parallel variants of the miniBUDE mini-app [19] written in C++/OpenMP and Julia/Tasks.

II. Related Work

AD tools including TAF [20] and Tapenade [21] have offered differing levels of support for OpenMP. Both tools perform source-to-source transformation, have to express the gradient of OpenMP/MPI programs using valid OpenMP/MPI, and have to hence implement rules for many different OpenMP clauses. By working on LLVM IR, we avoid having to explicitly handle e.g. firstprivate/lastprivate variables, which are expressed in LLVM IR using standard assignment instructions at appropriate locations. Moreover, embedding within the compiler enables Enzyme to differentiate after (parallel) optimization, including the ability to hoist parallel code out of loops and providing better aliasing information. Special treatment of MPI reversal was implemented in adjoint MPI libraries [22], integrated into various AD tools such as CoDiPack [23], [24], ADOL-C [25], Tapenade [26], TAF [27] and dco [28], and used in applications such as the computational fluid dynamics (CFD) solvers SU2 [29], OpenFOAM [30] and STAMPS [31], and the NASA Ice Sheet System Model [32]. The published solutions for MPI require modifications to the original code, including the use of special MPI function signatures. Adjoint MPI library extensions are developed separately from the AD tools used for the remaining program, and can interfere with certain program analyses like activity analysis [33]. Instead, Enzyme covers MPI in a transparent and seamless manner without manual intervention.

Reverse-mode differentiation of parallel read access to shared memory may result in concurrent increment access and thus require special treatment to avoid data races. Enzyme uses atomic updates whenever the analysis can not otherwise guarantee safe access. For the special case of stencil loops, PerforAD [34] instead provide a Python-based DSL that uses loop transformations to avoid concurrent increments during the reverse sweep. The functional programming language Futhark [35] differentiates high-level parallel routines and the authors discuss the use of generalized histograms, while other work considers generalized reductions [36] for the same task. Enzyme [16], [17] previously introduced support for race-free GPU-parallel programs (CUDA, ROCm) with support for different memory types, and block-level synchronization, as well as relevant AD and GPU-specific optimizations. This paper extends the work in [17] to differentiate any DAG-based parallel framework in a single tool, and alongside novel generic parallel optimizations.

III. AD Background

Differentiation of programs is performed by augmenting each individual instruction with auxiliary instructions to compute its partial derivative, and augmenting each individual variable with an auxiliary variable to hold derivative values. The derivatives are accumulated following the chain rule of calculus to obtain the derivatives of the overall program. The order in which individual derivatives are accumulated does not change the overall result, but does affect the run time and memory consumption.

Many tools implement AD capabilities using a variety of strategies and supporting input languages including C [37], [38], C++ [23], [39]–[41], Fortran [26], Julia [42], [43], or MATLAB [44], while machine learning frameworks such as TensorFlow [45], PyTorch [46], JAX [47], and DEX [48] support AD natively.

Two strategies are common: The *forward mode* accumulates derivatives in the order of the original computation, and is efficient for programs with few differentiable inputs and an arbitrary number of differentiable outputs. In contrast, the *reverse* or *adjoint mode* and the closely related *back propagation* are efficient for programs with an arbitrary number of differentiable inputs and few differentiable outputs. This is a common situation in machine learning, engineering and science, where functions with millions of input parameters are commonly optimized subject to a scalar loss function.

Reverse mode accumulates derivatives in the inverse order of the original computation, requiring data flow reversal and special handling for overwritten values that must be preserved or recomputed for the derivative computation of nonlinear instructions. One common approach is to trace the computation at run time using operator-overloading. Another approach is to use source-rewriting before compilation, which significantly reduces the performance overhead of differentiation, at the cost of more complex tool development. The Enzyme approach is closely related to source-rewriting, but has unique advantages due to its deep integration into the LLVM compiler. We refer to [16], [17] and [49], [50] for detailed discussions of Enzyme and AD, respectively.

IV. Differentiation model

Enzyme uses reverse mode AD by default, and the remaining discussion will be exclusively about this mode. We will refer to an instruction and its auxiliary derivative instruction as *primal* and *adjoint*, and we will refer to auxiliary variables as *shadow*. We will identify the shadow of the output of an instruction `I` with `shadow(I)`, which represents the derivative of `I`. Within Enzyme, taking the derivative of an instruction `I` involves four steps:

1) Load and zero `shadow(I)`.
2) Compute the partial derivative of `I` w.r.t. its inputs.
3) Multiply the result of (2) with the shadow retrieved in (1).
4) Increment the shadow of `I`'s input with the result of (3).

Evaluating the adjoint of all instructions in reverse order ensures that when evaluating the adjoint of `I`, `shadow(I)` contains the total derivative, or sum of all partial derivatives from its uses. This holds because the adjoints for all uses of `I`, which add the corresponding partial derivatives to `shadow(I)`, occur before the adjoint of `I` when run in reverse order since all uses of `I` must occur after `I` when run forward.

A. Differentiating parallel tasks

Instructions within fork-join parallel programs do not have a defined order in which they are run. Instead, instructions form a directed acyclic graph (DAG) of permissible orderings. A node in the DAG with multiple children represents a spawn, and a node with multiple successors represents a sync. Reverse mode AD of such a program requires reversal of that DAG. The above differentiation model has to be extended for this situation. Since `I` must still dominate all the uses of `I` in the original (primal) program, all adjoints of the uses are guaranteed to have been executed prior to the adjoint of `I`. The adjoint of those uses, however, may occur in parallel. When operating on a parallel program, Enzyme will perform an atomic addition or reduction when incrementing the shadow of an instruction that is not thread-local. This ensures that the total derivative is available and the computation is correct.

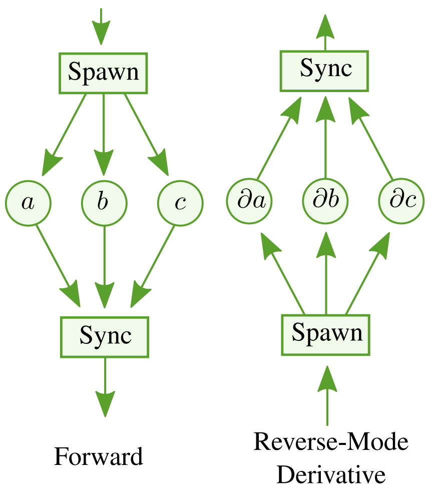


Fig. 2: Illustration of Correctness for Parallel AD: The control and data flow is reversed, hence inverting the data- and control flow. In the forward pass (left) the control flow goes from spawn to sync. In the reverse-mode derivative (right), the locations of spawn and sync are reversed.

Differentiation of a parallel for loop and `spawn/sync` pair can now be shown to correctly implement the reversal of the DAG. The `sync` in the primal is transformed into `spawn` in the adjoint, and a `spawn` in the primal is transformed into a `sync`. A parallel for loop spawns off several tasks in parallel that are subsequently synchronized. Differentiating a parallel for loop results in a parallel for loop of adjoints at the corresponding location in the reversed DAG. This is equivalent to the `sync` of the primal parallel for being transformed into a `spawn` of all tasks constituting the loop. See Figure 2 for an illustration. For a thorough example and proof of how to differentiate parallel control flow in the specific context of a GPU-style barrier in Enzyme (as opposed to any general parallelism, described here), see [17].

B. Differentiating message passing

MPI's model can be thought of as an implicit parallel for loop across the entire program, with distinct address spaces, focusing on explicit data management as opposed to execution management. MPI communication routines do not expose the implicit parallel for construct but instead expose only explicit data management using function calls with inputs and outputs. MPI's nonblocking communication (e.g., `MPI_Isend`, `MPI_Irecv`, `MPI_Wait`) can be treated as parallel task constructs, where `MPI_Isend` and `MPI_Irecv` dispatch a task synchronized at the corresponding MPI_Wait.[1] Differentiation of MPI in this fashion is quite efficient and results in twice the number of MPI calls, for both the primal and derivative values (which may be able to be fused during optimization), and at most thrice the amount of MPI-related memory (the original buffer to send/receive, the derivative buffer to send/receive, and potentially a temporary buffer for derivative accumulation).

C. *Caching of intermediate results*

The adjoints of instructions often require the arguments of the original value. For example, when computing the adjoint of x^2, one needs to preserve the original value of x to compute the adjoint $2x$. If the instruction that computed x cannot be rerun and the value is not otherwise available in the reverse pass, the value needs to be cached. Using a minimum-cut recompute vs cache analysis [17], Enzyme determines a minimal set of values that must be preserved in order to satisfy the dependencies of the reverse pass. Enzyme allocates caches in one of three ways:

1) Allocate a stack variable if that variable is guaranteed to be alive for the entire duration of the differentiation
2) Allocate an array prior to the loop and store the value in a slot indexed by the loop variable if the value is computed within a loop with known size
3) Dynamically reallocate an array within the loop if the loop does not have a known size

V. Compiler-Integrated Differentiation

In contrast to existing differentiation approaches, Enzyme performs differentiation inside of the compiler. This enables Enzyme to access and modify the program at a variety of stages during the compilation pipeline. Deeply integrating AD within the entire compiler stack provides Enzyme with a large amount of flexibility. For example, Enzyme can run additional optimizations both prior to and after differentiation, identify source-line information from metadata, rewrite and modify library calls, and even run a JIT compiler. In addition to enabling new optimization opportunities, these capabilities thereby allow Enzyme to handle a wide array of parallel constructs in a robust and concise manner by writing a few core routines that can be generally applied to parallel methods.

Enabling support for parallelism within our extension to Enzyme requires three steps:

1) Identifying a runtime call or construct which enables parallelism. This enables Enzyme to produce parallel-safe derivative accumulation.
2) Telling Enzyme how to call the parallel runtime call, which it will call with an automatically generated derivative of the body of that construct.
3) Marking any information which is required to compute the derivative of the parallel construct as needing to be preserved by Enzyme's caching infrastructure.

[1] Depending on MPI implementation, and parameters this may only specify that the task on the current node has completed (e.g. the MPI_Isend) and not necessarily that the corresponding task on the partner node has completed (e.g. the MPI_Irecv).

Given all this, Enzyme empowers the user to differentiate parallel constructs without rewriting their original code.

A. *Identifying Parallel Constructs*

The easiest way for Enzyme to identify a parallel construct is by identifying a corresponding runtime call. Enzyme provides several utilities for recognizing function calls that match a certain pattern. For example, when compiling with Clang (the C/C++ compiler frontend for LLVM), Flang (the Fortran compiler for LLVM), or MLIR (a higher level intermediate representation), a program with OpenMP parallelism will call the `__kmpc_fork_call` function to run a given closure on all threads (see Figure 3). When writing programs with MPI for parallelism, the LLVM IR will contain calls to functions like `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` which asynchronously send data to another process, receive data from another process, and wait for a given operation to finish, respectively. Identifying the parallel constructs from Julia is somewhat more difficult because some of its parallel runtime calls do not have a unique ABI and instead create just-in-time compiled functions. Instead of identifying these calls from the (potentially inlined) assembly structure of the function, Enzyme can explicitly mark a source-level Julia method (such as `Base.threads_for`) as matching a parallel pattern (e.g. a call to task creation). In particular, the ability to leverage the compiler to mark arbitrary functions enables Enzyme to be invariant to the randomized names generated by Julia's JIT. This not only permits Enzyme to recognize the individual task creation mechanisms, but alternatively can be used to identify an entire parallel for-loop construct directly, instead of the underlying tasks which implement it.

B. *Differentiating Parallel Constructs*

Now that Enzyme can identify a program's parallelism, Enzyme must be taught how to call these parallel constructs in order to fill in the derivative information. Some parallel constructs, such as OpenMP, or Julia Tasks take a function closure. Differentiating functions which call a closure requires telling Enzyme to differentiate the closure body and potentially wrapping the auto-generated derivative of the closure to match the expected ABI and calling convention. See Figure 4 for an example of Enzyme-generated derivative closures for the OpenMP program in Figure 3. Finally, one needs to tell Enzyme the corresponding adjoint of the function, just like any other functions or LLVM instructions. As an example, differentiating `MPI_ISend` results in a `MPI_Wait` and differentiating a Julia spawn task (`Base.enq_work`) in a corresponding Julia task wait (`Base.wait`).

C. *Data caching*

The final step to enable differentiation is to inform Enzyme about any information which must be preserved to compute the adjoint of a parallel construct. As an example, consider an OpenMP worksharing loop construct (`#pragma omp parallel for` which divides an iteration space of arbitrary size to be run efficiently on the available system threads. The bounds

```
void square(double* data) {
  #pragma omp parallel
  {
    int tid = threadid();
    data[tid] = data[tid] * data[tid];
  }
}
```

```
void outlined(double*& data) {
    int tid = threadid();
    data[tid] = data[tid] * data[tid];
}
void square(double* out, int start, end end) {
  __kmpc_fork(outlined, out, start, end);
}
```

Fig. 3: ***Left:*** An OpenMP function which squares an element of an array on each thread. ***Right:*** The compiler lowers this construct to a closure `outlined` of the body and a call to the `__kmpc_fork` runtime call which runs the closure on each thread.

```
void ∇square(float* data, float* d_data) {
  // Allocate an array to cache all the values of data[i]
  // from the reverse pass, so the original value can be
  // used to compute the reverse pass.
 float* cache = new float[numthreads()];
  // Run the forward pass on every thread.
  __kmpc_fork(aug_outlined, data, d_data, cache);
  // Run the reverse pass on every thread.
  __kmpc_fork(rev_outlined, data, d_data, cache);
  // Free the cache
  delete[] cache;
}
```

```
// Bodies automatically generated by Enzyme when
// informed about the closure.
void aug_outlined(float*& data, float*& d_data, float*& cache) {
  int tid = threadid();
  cache[tid] = data[tid];
  data[tid] = data[tid] * data[tid];
}
void rev_outlined(float*& data, float*& d_data, float*& cache) {
   int tid = threadid();
   d_data[tid] *= 2 * cache[tid];
}
```

Fig. 4: ***Left:*** Gradient of square (ref. Figure 3). This calls the OpenMP parallel runtime call twice, once for the forward pass, and once for the reverse pass. ***Right:*** The forward and reverse passes of the outlined OpenMP parallel body.

```
int send(double* data, double* d_data, int dst) {
  // The original and shadow requests.
  MPI_Request req, d_req;

  // The forward pass of Isend, which also stores what
  // type of instruction (and its buffer) for use in a
  // reverse wait.
  MPI_Isend(data, MPI_REAL, dst, req);
  d_req = {ISend, d_data, ... };
  ... // Code for corresponding Irecv
  MPI_Wait(&req);

  // Derivative of MPI_Wait:
  //  If the origin task was an Isend, perform Irecv
  if (d_req.type == ISend)
    MPI_Irecv(...);
  ... // Derivate code for corresponding Irecv
  // Derivative of MPI_Isend
  MPI_Wait(...)
}
```

Fig. 5: An asynchronous MPI send request and its corresponding derivative. Since the derivative of the wait must know what type of instruction it synchronized in order to spawn of its corresponding adjoint in the reverse, the request type is stored in the shadow request in the forward pass. A full MPI program would also need to call an analagous `recv` and derivative on the destination node.

of the loop must be preserved for the adjoint construct to execute a corresponding worksharing loop across the same number of iterations. Caching data for a `MPI_Wait`, however is more difficult. The derivative of a wait on task `t` is to spawn the corresponding derivative task `shadow(t)`. However, MPI has multiple types of tasks (send, receive) which may be synchronized by the same `MPI_Wait`. This can be resolved through the use of shadow variables. We can define the shadow variable of the original request to store what task was being waited upon. Therefore when computing the adjoint of `MPI_Wait` in the reverse pass, Enzyme can look inside the shadow request to identify whether it should create an `MPI_Isend` or `MPI_Irecv`. This is demonstrated in Figure 5.

D. General Applicability

Existing tools to differentiate parallel programs must understand every parallel construct in the language they are designed to differentiate. In contrast, by operating in the compiler, we can choose to instead differentiate parallel programs with many constructs (e.g. private memory, reductions) after they have been lowered into simpler operations (e.g. load and store operations). This enables Enzyme to differentiate these constructs without any explicit support being required, as Enzyme already knows how to differentiate memory operations. Similarly, higher level parallel languages such as RAJA, which internally implement parallelism using lower level frameworks (e.g. OpenMP) do not need any explicit support to be handled, since we can choose to differentiate after it has been lowered to OpenMP. This flexibility extends across languages. Adding the corresponding handler for an MPI call in Enzyme differentiates MPI code regardless of whether it was written directly in C++, or using the MPI.jl [51] wrapper inside Julia. Of course, this does not mean that differentiation needs to be applied at the lowest level, but that differentiating a single lower level construct enables differentiation of several higher-level routines and languages. For example, even though Enzyme differentiates Julia tasks, which are used to implement a "parallel for" in Julia, we still also provide an explicit derivative for the Julia "parallel for" for performance. In contrast, differentiating certain memory constructs (like private memory), or any code able to be optimized may be faster when differentiated at a lower-level since reducing the work of the original code can make an outsized impact in the reverse program [16], [17].

E. Optimization and Differentiation

Running optimizations prior to differentiation was found to provide a significant speedup in the original Enzyme paper [16].

This effect occurs for two primary reasons. First, the additional optimizations result in simplified code which has improved analysis properties (e.g. aliasing, readonly, etc). Secondly, the additional optimizations reduce the work done by the function being differentiated, which in turn, enables the corresponding generated derivative to perform less work in both the forward and backwards passes. The need to optimize parallel programs has been well studied in a variety of works such as Tapir applying optimizations for Cilk [52] and OpenMPOpt [53]. This need is accentuated in the context of differentiation where improving aliasing properties can enable Enzyme to avoid unnecessarily caching variables for use in the reverse pass. For example, without optimization an OpenMP closure function captures all of the surrounding variables by value, and can potentially alias any memory. Moreover, applying parallel optimization after differentiation may also help. For example, such an optimization may be able to merge the two parallel fork calls made in Figure 4. We evaluate the impact of running OpenMPOpt in the context of differentiation in Section VII.

VI. Other Parallel Constructs

This section gives an overview of how supporting parallel control flow (parallel for, task create/wait) and memory can enable support for other common parallel constructs.

A. Memory

1) Local vs Shared Memory: Enzyme is designed to have common caching and adjoint increment routines that can be used to implement the adjoint of any instruction of a function call with relative ease. Introducing a new parallel model does not require a modification to the caching infrastructure besides informing Enzyme about what calls are parallel.

The common adjoint increment routine begins by performing analysis to detect whether the shadow memory location being modified is thread- (or node-) local. This analysis builds of alias analysis to deduce if any allocation, or more specifically, offset into memory, could be used on another thread. As an example, an allocation defined within a thread which is not captured must be thread-local. If the memory location is thread-local, Enzyme performs an efficient serial load, add, and store. If this cannot be proven, Enzyme will next attempt to prove that the given memory location is the same for all threads within a parallel for loop (for all containing parallel loops). If this is the case, Enzyme will look in its catalog of reductions to see whether a reduction implementation for that style of thread exists; if so, Enzyme then will use it to sum the contribution for all threads. If none of these situations apply, Enzyme will perform an atomic add. Besides optionally registering a new reduction, a parallel framework designer adding Enzyme support can inform Enzyme that a given location is thread-local. It is legal to fall back and mark every location as being shared among threads (resulting in many atomics/reductions), but doing so may not be desirable for performance. Enzyme provides several helper methods for marking thread-local properties. The shadow of function-local registers and allocations can be marked as thread-local (this is the case for OpenMP, MPI, and CUDA but not for pthreads, Julia tasks, or Cilk tasks). Enzyme supports a differentiation configuration which assumes that the generated derivative function will itself be called in parallel and that any derivative memory location passed as an argument may be accumulated in parallel. While we found these options sufficient for OpenMP and MPI, additional thread-local settings can be implemented (and thus made available to any parallel model that chooses to apply them).

2) Private Memory: OpenMP and other parallel frameworks have a variety of different memory clauses. For example, OpenMP private (and its cousins firstprivate / lastprivate) specifies that a variable has a separate copy per thread (with first private initializing the thread-local value to the value outside the loop and the lastprivate specifying that after the loop completes, the final iteration's thread-local value should be copied to the variable outside the loop). These constructs are already lowered to allocations and stores (as required) at the semantically correct location. Therefore, no additional work is required to handle these.

Consider the program at the top left of Figure 6. Since the variable `in` is marked firstprivate, a thread-local copy of `in` will be created, initialized to the argument, as is made explicit on the bottom left of Figure 6 with `in_local`. When executed, the first iteration handled by each thread will set `out[i]` to `in`, whereas all other iterations will set `out[i]` to zero.

Differentiating this with Enzyme will produce the code to the right in Figure 6. In the reverse pass, the reverse for loop will set the derivative of `in_local` to zero at the start of an iteration (adjoint of `in_local = 0`), then increment the derivative of `in_local` by the derivative of `out[i]` (adjoint of `out[i] = in_local`). This approach simplifies to merely setting the derivative of `in_local` to the derivative of the last iteration when run in reverse, or equivalently the first iteration when run in the original program. Since the primal code set the first iteration of each thread equal to `in`, the correct adjoint is indeed the sum of the derivatives of all the indices that were set to `in`. This case would be especially challenging for any source-to-source AD transformation tool since OpenMP has no "for" construct that will subdivide the loop and then reverse the order of each per-thread chunk. In contrast, not only is this possible to do on the LLVM level with Enzyme, but is automatically handled by handling the parallel and memory primitives alone.

3) Reductions: Proper handling of private and shared memory allows Enzyme to handle higher-level parallel constructs built on top of memory, regardless of implementation. For example, the C++ version of LULESH implements a custom reduction, shown in Figure 7[2]. In contrast, RAJA provides a custom reduction operation/template for later use. Both reduction styles are automatically handled by Enzyme.

[2]Depending on the size and parallel overhead, it may be more efficient to implement parallel min as a divide-and-conquer. The example in Figure 7 is used by LULESH, and the divide-and-conquer style version is also to be handled by Enzyme.

```
void fp(double* out, double in) {
  #pragma omp parallel for firstprivate(in)
  for (int i=0; i<N; i++) {
    out[i] = in;
    in = 0;
  }
}

void fp(double* out, double in) {
  #pragma omp parallel
  {
    double in_local = in;
    #pragma omp for
    for (int i=0; i<N; i++) {
      out[i] = in_local;
      in_local = 0;
    }
  }
}
```

```
double ∇fp(double* out, double* d_out, double in) {
    double d_in = 0;
    #pragma omp parallel for firstprivate(in)
    for (int i = 0; i < N; i++) {
        out[i] = in;
        in = 0;
    }
    // Run the reverse pass
    __omp_parallel(rev_outlined, out, d_out, in, d_in);
    return d_in;
}
void rev_outlined(int tid, double*& out, double*& d_out,
                           double& in, double& d_in) {
    double d_in_local = 0;
    int lb = 0, ub = N;
    __omp_for_loop(tid, &lb, &ub);
    for (int i=ub-1; i>=0; i--) {
        d_in_local = 0; // adjoint of in_local = 0
        d_in_local += d_out[i]; // adjoint of out[i] = in_local
        d_out[i] = 0;
    }
    // Enzyme will fall back to atomic if not proven thread local.
    atomic { d_in += d_in_local; }
}
```

Fig. 6: ***Top Left:*** An OpenMP function that uses firstprivate memory to set the first iteration handled by each thread to `in`, the remainder to 0. ***Bottom Left:*** An explicit version of the code on the top left, with firstprivate being replaced with an equivalent thread-local `in_local`. ***Right:*** C code representing the gradient generated by Enzyme.

```
double min_per_thread[num_threads()];
#pragma omp parallel
{
  double min_value = 0;
  #pragma omp for
  for(int i = 0; i < N; i++)
    min_value = min(data[i], min_value);
  min_per_thread[omp_get_thread_num()] = min_value;
}
double final_val = 0;
for(int i = 1; i < omp_get_num_threads(); i++)
  final_val = min(final_val, min_per_thread[i]);
```

Fig. 7: A manual user-written min reduction, as simplified from its use from the `CalcCourantConstraintForElems` and `CalcHydroConstraintForElems` functions in LULESH. While this could be rewritten to use higher-level reduction routines which can be handled by both Enzyme and other tools, differentiating it "as-is" requires correct handling of a variety of OpenMP constructs.

B. Concurrent Caching

Instructions and allocations computed within a parallel region create thread-local values or registers. Special care must be taken to ensure that the same values created within each thread are available and mapped to the corresponding thread in the reverse pass. Enzyme caches such values by preallocating memory for each thread and storing each value at an index corresponding to the current thread ID. If the same threads used for the forward-pass are also available at the corresponding time in the reverse pass (this includes Julia threads, the LLVM OpenMP runtime[3]), the corresponding reverse computation will access the corresponding caches indexed by their thread ID. If a different number or set of threads may be available, the parallel framework must inform Enzyme how to remap the threads.

Caches of values computed within a worksharing parallel for loop which does not specify how the loop's iterations map to threads, however, can be stored in a location indexed by the iteration of the for loop. This approach provides flexibility in how iterations are distributed among threads and even permits a different mapping of threads to iterations in the reverse pass.

C. High-Level Language Constructs

1) Foreign Library Calls: Unlike statically compiled languages (e.g., Fortran, C++, Rust, Swift) that call into libraries such as MPI by linking to the appropriate symbol, just-in-time compiled languages must dynamically load the symbol at runtime for use. This requirement presents additional challenges for Enzyme because the compiler will not be able to recognize a call to `MPI_Send`, since all it will see is a call to a specific integer address. We remedy this within Enzyme.jl (Julia's bindings for Enzyme) by performing an additional processing step on the LLVM IR of the function to be differentiated by Enzyme. In that pass, Enzyme.jl will look for calls to an integer address and identify the name of the function being called by looking through Julia's symbol table. This then allows Enzyme to identify the function being called and generate the corresponding derivative code.[4]

2) Garbage Collection: Support for special garbage collection (GC) intrinsics must also be handled within Enzyme in order to differentiate parallel code that uses a library such as MPI.jl. Allocation of garbage-collected variables is straightforward and is handled by registering the garbage collection allocation function to Enzyme's allocation handler.

[3]This is stronger than the current OpenMP specification. However, as Enzyme exists within LLVM, this can be assumed. If the LLVM OpenMP runtime is changed to no longer have this property, Enzyme can check the LLVM version it was built against and select a different cache mapping.

[4]This process is done for all foreign library calls and thus remedies similar issues that may occur when calling other foreign libraries.

Julia contains special macros `GC.preserve` which specify that a given variable must be preserved within the given scope, even if there are no uses known by Julia. This macro is necessary when making foreign function calls, which may not appear to Julia as a use of the memory. The macro is lowered into the function call `gc_preserve_begin` and `gc_preserve_end` runtime calls, which takes a list of variables to be preserved.In addition to preserving the original variables as specified, Enzyme must modify the call to also preserve the shadow of any variable being preserved, since they may also be modified in a way not known to Julia. Enzyme will also add a corresponding GC preservation in the reverse pass. This informs Julia's garbage collector to similarly preserve variables when computing the adjoint of that region. For example, if a `memcpy` had its arguments marked for preservation in the original code, the derivative of the `memcpy` (containing stores and loads to the shadow) would now also have its arguments marked for preservation. There may be instances when preservation is not needed, but this overly conservative approach is still correct and may be further optimized later.

D. Non-determinism

Non-determinism in parallel programs can arise from undefined behavior in the program. As an example in a parallel program one can create a *write-race* where several threads write distinct values to a single memory location at the same time. In Enzyme, like in other serial and parallel AD tools, differentiating a program with undefined behavior may result in a gradient calculation with undefined behavior. Take a primal function which reads from undefined memory, this will result in a gradient function which reads from a corresponding location of undefined shadow (derivative) memory.

A further source of non-determinism is that it is possible to have a program with an undefined parallel execution order that yields a deterministic result (through synchronization, reductions, atomics, or simply distinct memory locations per thread). In the case of synchronization, the reversal of the dependency DAG described in Section IV-A and derivative accumulation described in Section VI-A enable correct handling deterministic, but racy programs. In the context of a barrier or locked/atomic region in the forward pass, this will result in a program semantically equivalent to another barrier or atomic region in the reverse pass. For the locked/atomic region, the now serialized parallel tasks must be executed in the reverse order, which is performed by caching the actual execution order. For atomic instructions this is simpler as atomic instructions within LLVM return the previous value of memory – which is precisely what would require caching. Some simpler atomic instructions like add do neither require caching the execution order, nor an additional value.

VII. Evaluation

To illustrate the composability of Enzyme's differentiation of parallel frameworks, we apply it to several distinct parallel variations of LULESH [54], [55], and miniBUDE [19]. In these results, *forward* denotes the time it takes to run the original program and *gradient* denotes the time it takes to both run the original program and compute the derivative of all the inputs, and *overhead* denotes the ratio of the gradient runtime to the forward runtime.

LULESH is a 5000-line hydrodynamics proxy application developed by Lawrence Livermore National Laboratory[5]. As an unstructured explicit shock hydrodynamics solver, it emulates the behavior of complex solvers by splitting the computational domain into volumetric elements on an unstructured mesh, hence mimicking the complex data movement characteristics of unstructured data structures. We designed our evaluation to test how effectively a single low-level implementation of parallelism within an automatic differentiation tool can enable a diverse set of parallelism models. We evaluate LULESH variations that use MPI, OpenMP, hybrid MPI+OpenMP, MPI.jl, and the RAJA portable parallel programming framework, written in C++ and Julia. To compare our performance against the automatic differentiation performance to the CoDiPack-differentiated LULESH of Hück et al. [56].

Mini-BUDE is a 200-line mini-app. developed by the University of Bristol emulating the main computational kernels of the heavily compute-bound molecular docking engine BUDE [57]. BUDE predicts the binding energy of two molecules using molecular mechanics, in order to evaluate the ability of test molecules to bind with a target molecule. Each potential pose of the molecules needs to be evaluated for its free energy, hence resulting in hundreds of thousands pose-evaluations for each molecule. Our evaluation on miniBUDE was designed to validate our automatic differentiation performance claims on LULESH on a second, distinct application, as well as testing Enzyme's ability to automatically differentiate Julia's shared-memory parallelism. We evaluate an OpenMP version in C++, and a Julia-version utilizing tasks.

A. Benchmark Implementation Details

a) ***C++****:* The C++ code is based on the official 2.0 release of LULESH[6]. We modified the code by creating a second shadow domain to store the derivative result and added an option to switch between primal and derivative computation. The only other change made was to pass member functions to the communication subroutines at compile time rather than as an array. This is not required for differentiation or correctness, and member functions are passed as an array in the RAJA version. For the OpenMP-version of miniBUDE[7] we created a shadow domain for the computational kernel and added an option to switch betweeen primal and derivative computation.

b) ***RAJA****:* The RAJA code is based off LULESH 2.0 from the official RAJA benchmark repository[8]. We added a second domain to store derivatives and added a flag to enable differentiation. The only changes we made were to use a standard allocator, rather than the custom allocator in the repository, and to free memory at the end of each

[5]https://asc.llnl.gov/codes/proxy-apps/lulesh
[6]`https://github.com/LLNL/LULESH`
[7]`https://github.com/UoB-HPC/miniBUDE`
[8]`https://github.com/LLNL/RAJAProxies`

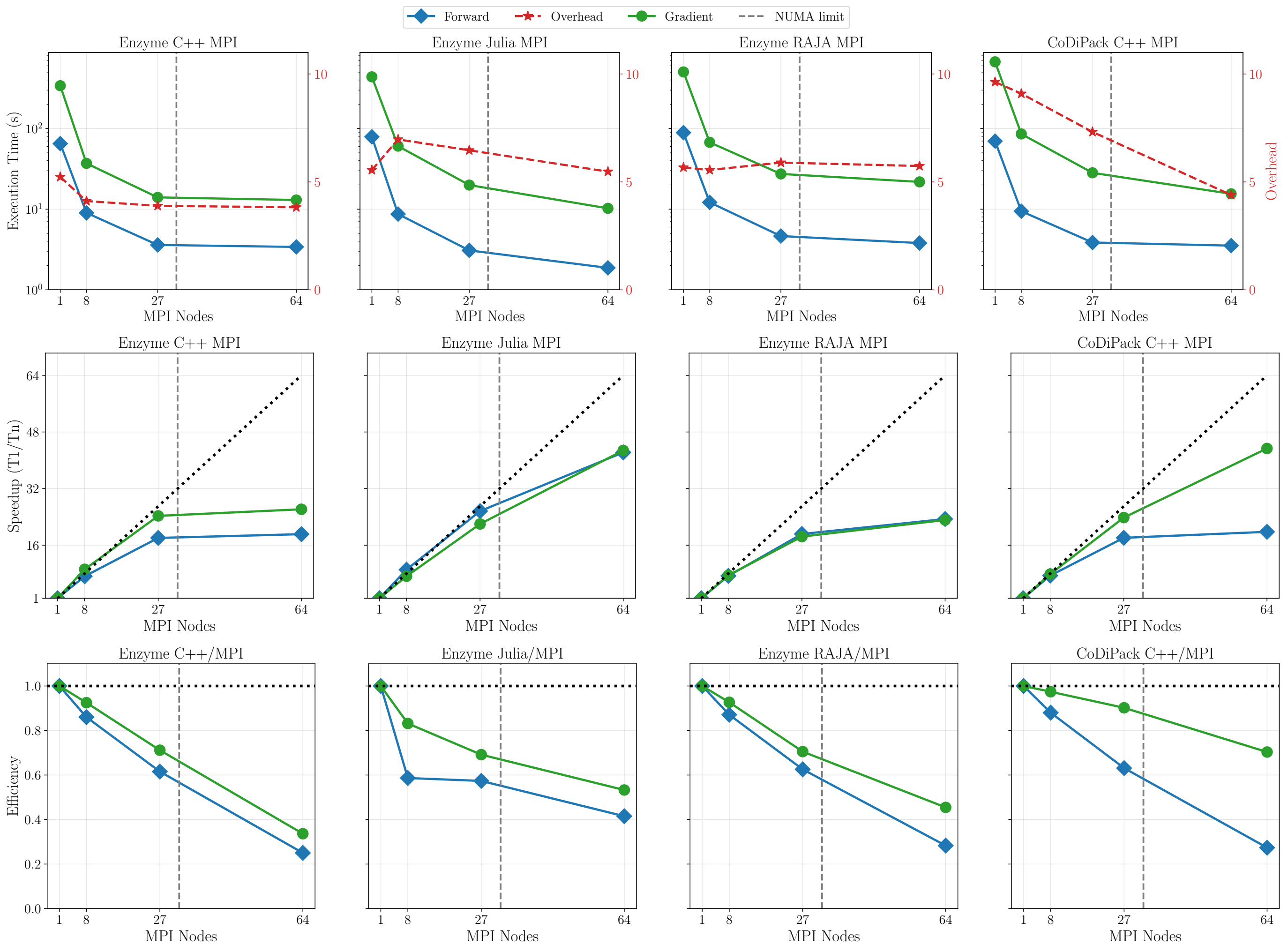


Fig. 8: **Top Row:** Runtime for 10 iterations of the LULESH proxy-benchmark for the different implementations. The number of processors is increased, while the overall problem size stays fixed. We used the following task count-block size combinations: 1:192, 8:96, 27:64, and 64:48. **Middle Row:** Strong scaling behavior. **Bottom Row:** Weak scaling behavior. The number of processors is increased, while the per-processor problem size stays fixed. The block size used was 48.

iteration (calling `std::vector::shrink_to_fit` in addition to the `std::vector::clear`). The RAJA version was seemingly identical to the vanilla C++ version with the exception of using C++ `std::vector` instead of bare pointers, RAJA looping constructs instead of regular serial or OpenMP parallel for loops, and the runtime member function passing.

*c) **Julia**:* Since no official or unofficial version of LULESH exists in Julia, we built a new version from scratch based on the official C++ version, and LLNL's unverified FORTRAN version of LULESH 1.0. We elected to port LULESH's MPI communication to Julia through the use of MPI.jl [51]. While the code attempts to remain faithful to the official C++ code, some differences include the use of garbage-collected arrays and minor changes to better match standard Julia design paradigms. The code's correctness was verified against LULESH's correctness checks of [18]. For the Julia-version of miniBUDE we created a shadow domain for the computational kernel, no-inlined the core kernel, and added the option to switch between the primal and derivative computation for evaluation purposes.

*d) **CoDiPack**:* CoDiPack [23] is an existing operator overloading automatic differentiation tool for C++ with an extension to differentiate through MPI code. We use a version of LULESH modified by the CoDiPack authors [56] which rewrites variables and communication within the application to use CoDiPack-specific variants. We use CoDiPack LULESH as a performance baseline for existing state-of-the art tools. Like Enzyme, we use CoDiPack in reverse-mode for the tests.

e) Setup: Experiments were run on an AWS c6i.metal instance with hyper-threading and Turbo Boost disabled, running Ubuntu 20.04 running on a dual-socket Intel Xeon Platinum 8375C CPU at 2.9 GHz with 32 cores each and 256 GB RAM.

All C++ codes are compiled using LLVM 14, and the Julia codes use Julia version 1.7.1. C++ MPI-codes are run with OpenMPI 4.0.3, and Julia's MPI code is run with MPICH 4.0.1. Experiments with OpenMP were benchmarked with LLVM 14's OpenMP implementation. We measured the time

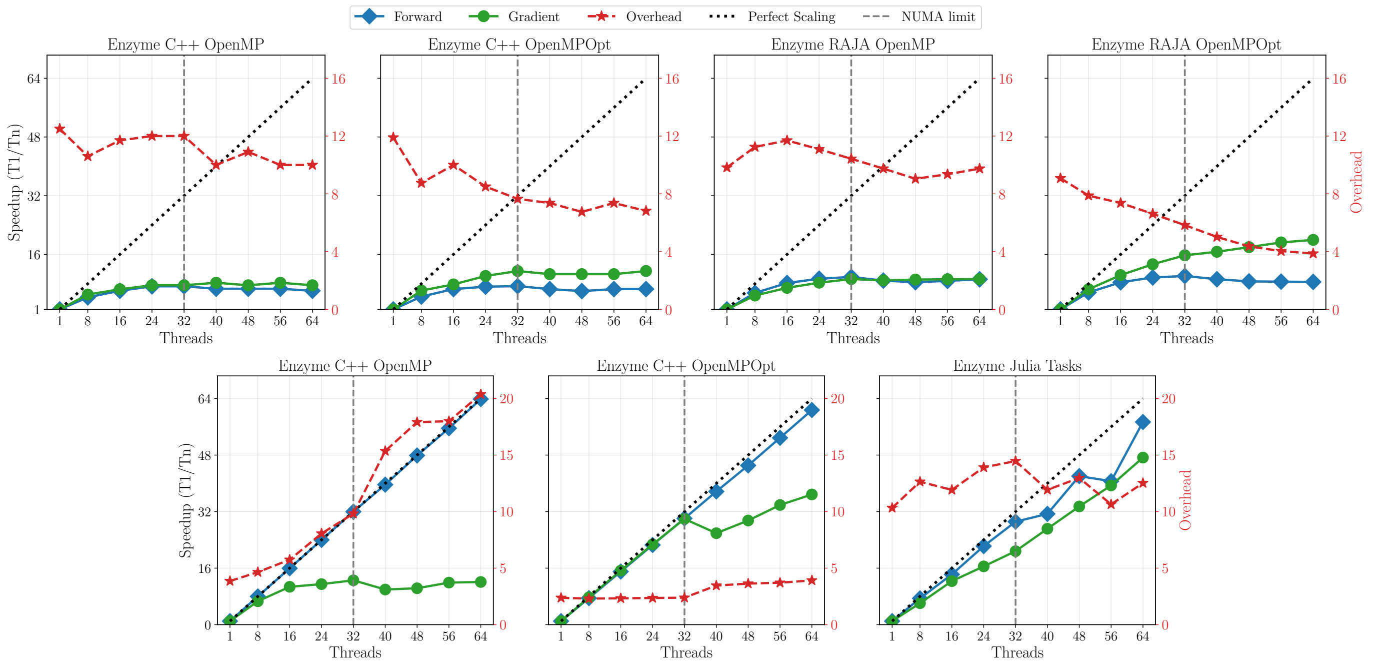


Fig. 9: Thread parallelism strong scaling on the LULESH **(Top Row)** and BUDE **(Bottom Row)** proxy-benchmarks for the different evaluated implementations. The number of available processors is increased, while the overall problem size stays fixed. The block size used for LULESH was 96 and the default number of poses was used for BUDE.

taken to execute the forward and differentiated versions of LULESH, and miniBUDE using different types of parallelism. For LULESH MPI strong scaling we report runtimes from 10 consecutive iterations from one run. For LULESH C++ and RAJA, MPI weak scaling, thread scaling, and MPI task and thread strong scaling we report times for 100 iterations. All remaining runs of LULESH experiments use 10 iterations. For the C++ and Julia versions of miniBUDE, we report times for the default number of iterations (100 and 8 iterations respectively). We studied the parallel scaling of the forward and differentiated code with increasing MPI rank and OpenMP thread counts.

B. Gradient verification

For realistic applications it is rarely feasible to perform tests for all relevant input values, nor is it generally feasible to compute the entire gradient or Jacobian matrix for applications with many active inputs or outputs using both the forward and reverse mode. It is therefore common practice to limit tests to certain inputs, and in the case of AD, to further limit tests to certain projections of the Jacobian matrix that can be efficiently computed with multiple approaches for comparison.

In order to verify the gradients computed by Enzyme, we selected a projection that can be efficiently computed using the reverse mode, while also being easy to approximate using finite differences. Using reverse mode, this projection can be computed by initializing all shadow variables to 1, and summing the computed shadow variables. When using finite differences, the same projection can be computed by perturbing all input variables at once by the same small value and summing the resulting derivatives of all output variables as approximated by the corresponding finite difference formula (we use central differences for the perturbations and derivative approximations). Both projections should yield the same scalar value, up to round-off and truncation errors. We note that this is similar to the "fast mode" gradient check [58] implemented in PyTorch.

VIII. Results

For each of the benchmarks (LULESH, miniBUDE), and parallel frameworks (OpenMP, MPI, OpenMP+MPI, RAJA, Julia Threads, MPI.jl) we evaluated the scalability of the original code, Enzyme-generated derivatives, and baseline CoDiPack-generated derivatives, if available.

LULESH requires the number of MPI ranks to be a perfect cube, MPI scaling tests hence ran on 1, 8, 27, and 64 ranks. The strong scaling of the benchmarks is shown in Figure 8 **(middle row)**. Here, we plot the speedup (time to run on one rank divided by time to run on N ranks) as the total amount of work is fixed while the number of ranks is increased. We find the scaling behavior of the derivative computation to be better than that of the primal in the C++, RAJA and CoDiPack cases. It matches the primal for the Julia implementation. It can be observed that the speedup of all cases reduces after 27 ranks, because of non-uniform memory access (NUMA). Each socket on the AWS instance can support 32 threads, beyond which threads must access non-local memory resulting in increased memory latency and consequently reduced speedup.

In both strong and weak scaling experiments, all versions of MPI-based LULESH differentiated with Enzyme (C++, Julia, RAJA), the differentiated code scales similarly to that of the original function. For the C++ and RAJA tests the decreased weak scaling of both the original LULESH benchmark and

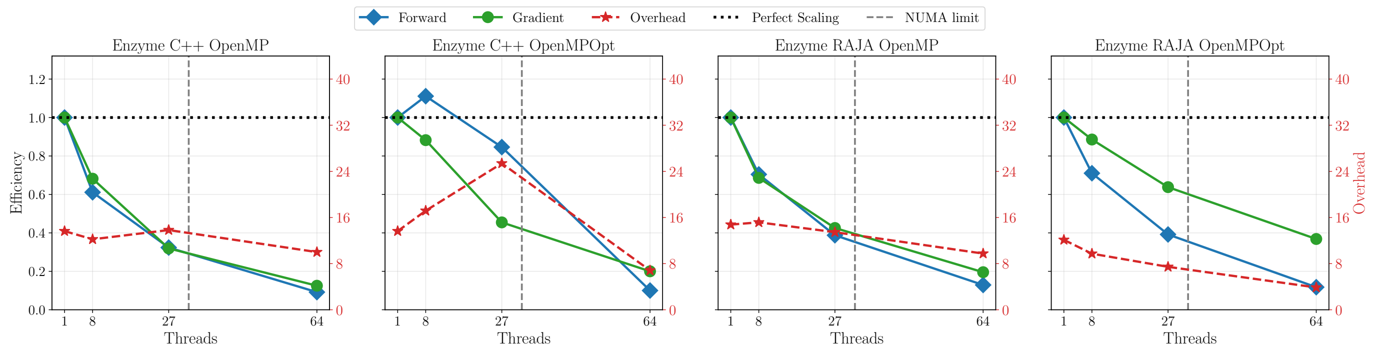


Fig. 10: Thread parallelism weak scaling on the LULESH proxy-benchmarks for the different evaluated implementations. The number of available processors is increased, while the problem size per processor stays fixed. We used the following thread count-block size combinations: 1:24, 8:48, 27:72, and 64:96.

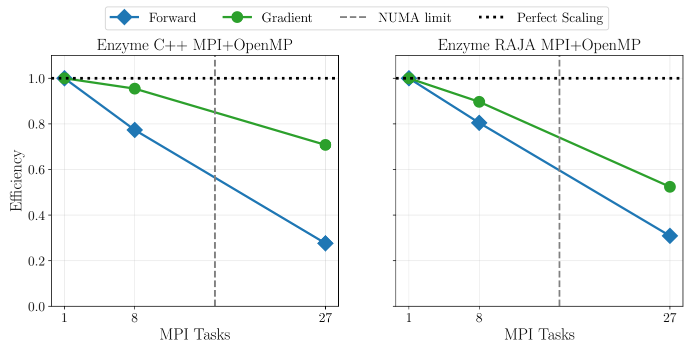


Fig. 11: Efficiency of LULESH when running with 1 8, and 27 MPI ranks, 2 OpenMP threads, and a block size of 48.

its derivatives can be explained by NUMA effects that occur when one needs to access data on more than one socket. We attribute the performance difference between LULESH.jl, and other LULESH implementations to the used MPI versions.

As the CoDiPack code is a modification of the C++ LULESH codebase with CoDiPack primitives, we can roughly compare the run times of Enzyme on the C++ LULESH against CoDiPack LULESH. While the CoDiPack gradient appears to scale better than Enzyme, this is because CoDiPack has a large gradient overhead (additional instructions required to compute the derivative of a single instruction in the reverse pass) for serial instructions unrelated to MPI. This causes the overall gradient overhead (considering all MPI and serial instructions) for CoDiPack to be quite high at 1 rank (see Figure 8 **(top row)**). The scaling tests perform fewer serial instructions per rank at higher node, causing the total overhead to be composed of fewer serial instructions in proportion to an MPI call. As a result, CoDiPack's apparently improved scalability is an artifact of the higher serial differentiation overhead being called proportionally fewer times, rather than scalability of its MPI differentiation.

We also evaluated strong scaling performance of the OpenMP C++ and RAJA versions of LULESH in Figure 9 (**top row**) (CoDiPack cannot differentiate OpenMP LULESH, and LULESH.jl does not use threads). To evaluate the effectiveness of parallel optimization we ran versions of the OpenMP LULESH with and without OpenMPOpt enabled. We extended the OpenMPOpt in LLVM 14 to also handle hoisting loads out of parallel regions. We again find that the scaling behavior of the derivative matches that of the original function.

We find that LULESH OpenMP has a relatively flat gradient overhead. The overhead drops when OpenMPOpt is enabled due largely to the fact that OpenMPOpt moves a pointer indirection out of a loop, improving alias analysis and allowing Enzyme to avoid caching as much data.

We evaluated the strong scaling performance of the OpenMP C++, OpenMPOpt C++, and Julia Task versions of miniBUDE (OpenMPOpt does not apply to Julia tasks). With regular OpenMP, the gradient overhead worsens as threads increase but does not grow with OpenMPOpt. This is again due to parallel load hoisting moving data outside a loop, which in this test enables Enzyme to avoid having to cache any data at all, electing instead to recompute temporaries. There is a slight decrease in scalability for the gradient at 32 threads due to using both sockets at that point and needing to update data from both CPU's. Notably, the gradient continues to scale on multiple sockets after the initial performance loss. miniBUDE.jl's overhead is higher, but again scales well. The higher overhead is because Julia arrays have an extra level of pointer indirection that causes alias analysis to conclude that several values need be cached. The amount of data being cached is still moderate due to Enzyme's ability to rematerialize temporary allocations.

The weak scaling performance of the OpenMP C++ and RAJA versions of LULESH can be found in Figure 10. We again find that scaling of the LULESH OpenMP and OpenMPOpt gradient matches that of the primal. The C++ OpenMPOpt displays anomalous behavior because OpenMPOpt did not optimize for a single thread as effectively. Finally, Figure 11 shows the scaling behavior of LULESH using both MPI task and OpenMP thread parallelism.

Overall, for all types of parallelism, the differentiated code scales similarly to the forward code. Since Enzyme can cache data without contention or a shared data structure, only gradient accumulation may add contention to the program. The use of analyses which detect what pointers and registers are thread local and thus can be accumulated serially helps preserve the parallel scaling properties.

IX. Conclusion

We have introduced a composable and generic LLVM-based mechanism to differentiate a variety of parallel programming models. In addition to simplifying the ability to handle high-level parallelism constructs, this marks the first time that an automatic differentiation tool can handle multiple parallelism models and multiple languages with a single implementation. We showcase the potential of this approach on the proxy apps LULESH, and miniBUDE, demonstrating Enzyme's practical use in real-world scientific simulation codes with nontrivial parallelization patterns. The overhead of the differentiated code is well inside the expected runtime of overheads of other state-of-the-art differentiation tools, is even comparable to the overhead of sequential programs [59], and observes the same scaling behavior of the differentiated code when compared with the original code. At the same time Enzyme does not require its users to utilize custom Adjoint-MPI libraries, and rewrite their application, making its AD of parallel programs much more accessible for users.

Acknowledgment

The authors would like to thank to Alex Chernyakovsky for thoughtful comments on the work. The authors would also like to thank Alexander Hück and Max Sagebaum for assistance in setting up the CoDiPack LULESH benchmark.

William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DE-SC0019323. Valentin Churavy gratefully acknowledges funding from NSF (grants OAC-1835443, AGS-1835860, and AGS-1835881), DARPA under agreement number HR0011-20-9-0016 (PaPPa). This research was made possible by the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program, by the Paul G. Allen Family Foundation, Charles Trimble, Audi Environmental Foundation. This material is based upon work supported by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003965. This research was supported in part by Los Alamos National Laboratories grant 531711, and in part by NSF Cyberinfrastructure for Sustained Scientific Innovation (CSSI) award numbers: 2104068 and 2103804. Research was sponsored in part by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

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.

Ludger Paehler was supported in part by the German Research Council (DFG) under grant agreement No. 326472365.

References

[1] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," 2018.

[2] M. B. Giles and N. A. Pierce, "An introduction to the adjoint approach to design," *Flow, turbulence and combustion*, vol. 65, no. 3, pp. 393–415, 2000.

[3] J. R. Martins and A. B. Lambe, "Multidisciplinary design optimization: a survey of architectures," *AIAA journal*, vol. 51, no. 9, pp. 2049–2075, 2013.

[4] O. Ghattas and K. Willcox, "Learning physics-based models from data: perspectives from inverse problems and model reduction," *Acta Numerica*, vol. 30, pp. 445–554, 2021.

[5] T.-M. Li, M. Lukáč, M. Gharbi, and J. Ragan-Kelley, "Differentiable vector graphics rasterization for editing and learning," *ACM Transactions on Graphics (TOG)*, vol. 39, no. 6, pp. 1–15, 2020.

[6] W. Gropp, W. D. Gropp, E. Lusk, A. Skjellum, and A. D. F. E. E. Lusk, *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, 1999, vol. 1.

[7] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[8] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017.

[9] D. Beckingsale, R. Hornung, T. Scogland, and A. Vargas, "Performance portable C++ programming with RAJA," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 455–456.

[10] H. M. Bücker, B. Lang, D. an Mey, and C. H. Bischof, "Bringing together automatic differentiation and OpenMP," in *Proceedings of the 15th ACM International Conference on Supercomputing, Sorrento, Italy, June 17–21, 2001*. New York: ACM Press, 2001, pp. 246–251. [Online]. Available: http://doi.acm.org/10.1145/377792.377842

[11] H. M. Bücker, B. Lang, A. Rasch, C. H. Bischof, and D. an Mey, "Explicit loop scheduling in OpenMP for parallel automatic differentiation," in *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications, Moncton, NB, Canada, June 16–19, 2002*, J. N. Almhana and V. C. Bhavsar, Eds. Los Alamitos, CA: IEEE Computer Society Press, 2002, pp. 121–126. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/HPCSA.2002.1019144

[12] C. Bischof, N. Guertler, A. Kowarz, and A. Walther, "Parallel reverse mode automatic differentiation for OpenMP programs with ADOL-C," in *Advances in Automatic Differentiation*, C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, and J. Utke, Eds. Springer, 2008, pp. 163–173.

[13] M. Förster, "Algorithmic differentiation of pragma-defined parallel regions: Differentiating computer programs containing OpenMP," Ph.D. dissertation, RWTH Aachen, 2014.

[14] J. Hückelheim, P. D. Hovland, M. M. Strout, and J.-D. Müller, "Reverse-mode algorithmic differentiation of an OpenMP-parallel compressible flow solver," *International Journal for High Performance Computing Applications*, 2017.

[15] J. Hückelheim, P. Hovland, M. Strout, and J.-D. Müller, "Parallelizable adjoint stencil computations using transposed forward-mode algorithmic differentiation," *Optimization Methods and Software*, vol. 33, no. 4-6, pp. 672–693, 2018. [Online]. Available: https://doi.org/10.1080/10556788.2018.1435654

[16] W. S. Moses and V. Churavy, “Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients,” in *Advances in Neural Information Processing Systems*, 2020, vol. 33, pp. 12 472–12 485.

[17] W. S. Moses, V. Churavy, L. Paehler, J. Hückelheim, S. Hari Krishna Narayanan, M. Schanen, and J. Doerfert, “Reverse-mode automatic differentiation and optimization of GPU kernels via Enzyme,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’21. New York, NY, USA: Association for Computing Machinery, 2021.

[18] I. Karlin, J. Keasler, and R. Neely, “LULESH 2.0 Updates and Changes, number = LLNL-TR-641973,” Tech. Rep., August 2013.

[19] A. Poenaru, S. Mcintosh-Smith, and T. Lin, “A performance analysis of modern parallel programming models using a compute-bound application,” in *High Performance Computing - 36th International Conference, ISC High Performance 2021, Proceedings*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer, Jun. 2021, pp. 332–350. [Online]. Available: https://www.isc-hpc.com/

[20] U. Naumann, L. Hascoët, C. Hill, P. Hovland, J. Riehme, and J. Utke, “A framework for proving correctness of adjoint message-passing programs,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 316–321.

[21] J. Hückelheim and L. Hascoët, “Source-to-source automatic differentiation of openmp parallel loops,” *ACM Trans. Math. Softw.*, vol. 48, no. 1, feb 2022. [Online]. Available: https://doi.org/10.1145/3472796

[22] M. Schanen, U. Naumann, L. Hascoët, and J. Utke, “Interpretative adjoints for numerical simulation codes using MPI,” *Procedia Computer Science*, vol. 1, no. 1, pp. 1825–1833, 2010, iCCS 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S187705091000205X

[23] M. Sagebaum, T. Albring, and N. R. Gauger, “High-performance derivative computations using codipack,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 4, pp. 1–26, 2019.

[24] M. Sagebaum and N. R. Gauger, “Medipack–message differentiation package,” 2020.

[25] A. Walther and A. Griewank, “Getting started with ADOL-C,” in *Combinatorial Scientific Computing*, U. Naumann and O. Schenk, Eds. Chapman-Hall CRC Computational Science, 2012, ch. 7, pp. 181–202.

[26] L. Hascoët and V. Pascual, “The Tapenade automatic differentiation tool: Principles, model, and specification,” *ACM Transactions On Mathematical Software*, vol. 39, no. 3, 2013. [Online]. Available: http://dx.doi.org/10.1145/2450153.2450158

[27] R. Giering, T. Kaminski, R. Todling, R. Errico, R. Gelaro, and N. Winslow, “Tangent linear and adjoint versions of NASA/GMAO’s Fortran 90 global weather forecast model,” in *Automatic Differentiation: Applications, Theory, and Implementations*. Springer, 2006, pp. 275–284.

[28] J. Lotz, U. Naumann, M. Sagebaum, and M. Schanen, “Discrete adjoints of PETSc through dco/c++ and adjoint MPI,” in *Euro-Par 2013 Parallel Processing*, F. Wolf, B. Mohr, and D. an Mey, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 497–507.

[29] T. A. Albring, M. Sagebaum, and N. R. Gauger, “Efficient aerodynamic design using the discrete adjoint method in SU2,” in *17th AIAA/ISSMO multidisciplinary analysis and optimization conference*, 2016, p. 3518.

[30] M. Towara, M. Schanen, and U. Naumann, “MPI-parallel discrete adjoint OpenFOAM,” *Procedia Computer Science*, vol. 51, pp. 19–28, 2015, international Conference On Computational Science, ICCS 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050915009898

[31] J.-D. Müller, J. Hueckelheim, and O. Mykhaskiv, “STAMPS: a finite-volume solver framework for adjoint codes derived with source-transformation AD,” in *2018 Multidisciplinary Analysis and Optimization Conference*, 2018, p. 2928.

[32] E. Larour, J. Utke, A. Bovin, M. Morlighem, and G. Perez, “An approach to computing discrete adjoints for MPI-parallelized models applied to ice sheet system model 4.11,” *Geoscientific Model Development*, vol. 9, no. 11, pp. 3907–3918, 2016. [Online]. Available: https://gmd.copernicus.org/articles/9/3907/2016/

[33] M. M. Strout, B. Kreaseck, and P. D. Hovland, “Data-flow analysis for MPI programs,” in *2006 International Conference on Parallel Processing (ICPP’06)*, 2006, pp. 175–184.

[34] J. Hückelheim, N. Kukreja, S. H. K. Narayanan, F. Luporini, G. Gorman, and P. Hovland, “Automatic differentiation for adjoint stencil loops,” in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337821.3337906

[35] R. Schenck, O. Rønning, T. Henriksen, and C. E. Oancea, “Ad for an array language with nested parallelism,” *arXiv preprint arXiv:2202.10297*, 2022.

[36] J. Hückelheim and J. Doerfert, “Spray: Sparse reductions of arrays in openmp,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 475–484.

[37] C. H. Bischof, L. Roh, and A. J. Mauer-Oats, “ADIC: an extensible automatic differentiation tool for ANSI-C,” *Software: Practice and Experience*, vol. 27, no. 12, pp. 1427–1456, 1997.

[38] L. Hascoet and V. Pascual, “The Tapenade automatic differentiation tool: principles, model, and specification,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 3, pp. 1–43, 2013.

[39] A. Griewank, D. Juedes, and J. Utke, “Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 22, no. 2, pp. 131–167, 1996.

[40] R. J. Hogan, “Fast reverse-mode automatic differentiation using expression templates in C++,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 4, pp. 1–16, 2014.

[41] V. Vassilev, M. Vassilev, A. Penev, L. Moneta, and V. Ilieva, “Clad—automatic differentiation using clang and llvm,” in *Journal of Physics: Conference Series*, vol. 608, no. 1. IOP Publishing, 2015, p. 012055.

[42] J. Revels, M. Lubin, and T. Papamarkou, “Forward-mode automatic differentiation in Julia,” *arXiv preprint arXiv:1607.07892*, 2016.

[43] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt, “A differentiable programming system to bridge machine learning and scientific computing,” 2019. [Online]. Available: https://arxiv.org/abs/1907.07587

[44] C. Bischof, B. Lang, and A. Vehreschild, “Automatic differentiation for MATLAB programs,” in *PAMM: Proceedings in Applied Mathematics and Mechanics*, vol. 2, no. 1. Wiley Online Library, 2003, pp. 50–53.

[45] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “TensorFlow: A system for large-scale machine learning,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[46] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, “Automatic differentiation in PyTorch,” in *NIPS 2017 Workshop Autodiff*, 2017.

[47] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, and S. Wanderman-Milne, “JAX: composable transformations of Python+NumPy programs, 2018,” *URL http://github. com/google/jax*, vol. 4, p. 16, 2020.

[48] A. Paszke, D. Johnson, D. Duvenaud, D. Vytiniotis, A. Radul, M. Johnson, J. Ragan-Kelley, and D. Maclaurin, “Getting to the point. index sets and parallelism-preserving autodiff for pointful array programming,” *arXiv preprint arXiv:2104.05372*, 2021.

[49] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[50] U. Naumann, *The art of differentiating computer programs: an introduction to algorithmic differentiation*. SIAM, 2011.

[51] S. Byrne, L. C. Wilcox, and V. Churavy, “MPI.jl: Julia bindings for the Message Passing Interface,” in *Proceedings of the JuliaCon Conferences*, vol. 1, no. 1, 2021, p. 68.

[52] T. B. Schardl, W. S. Moses, and C. E. Leiserson, “Tapir: Embedding fork-join parallelism into LLVM’s intermediate representation,” in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 249–265.

[53] J. Doerfert and H. Finkel, “Compiler optimizations for openmp,” in *International Workshop on OpenMP*. Springer, 2018, pp. 113–127.

[54] I. Karlin, “Lulesh programming model and performance ports overview,” Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2012.

[55] I. Karlin, J. Keasler, and R. Neely, “LULESH 2.0 Updates and Changes,number = LLNL-TR-641973,” Tech. Rep., August 2013.

[56] A. Hück, J. Protze, J.-P. Lehr, C. Terboven, C. Bischof, and M. S. Müller, “Towards compiler-aided correctness checking of adjoint mpi applications,” in *2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness)*. IEEE, 2020, pp. 40–48.

[57] S. McIntosh-Smith, J. Price, R. B. Sessions, and A. A. Ibarra, “High performance in silico virtual drug screening on many-core processors,”

The International Journal of High Performance Computing Applications, vol. 29, no. 2, pp. 119–134, 2015, pMID: 25972727. [Online]. Available: https://doi.org/10.1177/1094342014528252

[58] PyTorch, "Fast backward mode gradcheck," https://pytorch.org/docs/stable/notes/gradcheck.html#fast-backward-mode-gradcheck.

[59] J.-D. Müller and P. Cusdin, "On the performance of discrete adjoint cfd codes using automatic differentiation," *International journal for numerical methods in fluids*, vol. 47, no. 8-9, pp. 939–945, 2005.