Towards compiler-aided correctness checking of adjoint MPI applications

Alexander Hück*, Joachim Protze†, Jan-Patrick Lehr*,
Christian Terboven†, Christian Bischof* and Matthias S. Müller†
*Institute for Scientific Computing, Technische Universität Darmstadt, Darmstadt, Germany
{alexander.hueck, jan-patrick.lehr, christian.bischof}@tu-darmstadt.de
†IT Center and Chair for High-Performance Computing, RWTH Aachen University, Aachen, Germany
{protze, terboven, mueller}@itc.rwth-aachen.de

***Abstract*—Algorithmic Differentiation (AD) is a set of techniques to calculate derivatives of a computer program. In C++, AD typically requires (i) a type change of the built-in double, and (ii) a replacement of all MPI calls with AD-specific implementations. This poses challenges on MPI correctness tools, such as MUST, a dynamic checker, and TypeART, its memory sanitizer extension. In particular, AD impacts (i) memory layouts of the whole code, (ii) requires more memory allocations tracking by TypeART, and (iii) approximately doubles the MPI type checks of MUST due to an AD-specific communication reversal. To address these challenges, we propose a new callback interface for MUST to reduce the number of intercepted MPI calls, and, also, improve the filtering capabilities of TypeART to reduce tracking of temporary allocations for the derivative computation. We evaluate our approach on an AD-enhanced version of CORAL LULESH. In particular, we reduce stack variable tracking from 32 million to 13 thousand. MUST with TypeART and the callback interface reduces the runtime overhead to that of vanilla MUST.**

***Index Terms*—adjoint MPI, correctness, type mismatch, algorithmic differentiation**

I. Introduction

In previous work, we extended the dynamic MPI correctness checker MUST [1] with our tool TypeART [2] to detect type-related mismatches between the type-less MPI communication buffers and the declared datatype. In particular, the work aimed at detecting errors w.r.t. manually constructed MPI derived datatypes. Here, the developer is responsible for constructing a *memory overlay* by specifying offsets for the MPI library to extract and communicate the correct values of a data structure. Any mistake in the construction of the derived datatype by, e.g., specifying a wrong offset can lead to subtle bugs or code portability issues, as the standard states in [3, Sec. 4.1.12]:

> "It is not expected that MPI implementations will be able to detect erroneous, 'out of bound' displacements [...]"

TypeART tracks all memory allocations relevant to MPI calls and, thus, MUST with TypeART can detect such errors.

A. Challenges of domain-specific MPI correctness

In this work, we investigate the particular use-case of algorithmic differentiation (AD, [4], [5]). AD is a set of techniques to compute derivatives of a target program to, e.g., conduct sensitivity studies for model verification [6] or data assimilation [7]. To that end, in C++, all built-in floating-point types are changed to a user-defined AD type, which provides overloads for all operators to compute the original and derivative value, respectively, or record the operation performed, depending on the AD approach employed. Book-keeping for distributed derivative computations is handled with domain-specific MPI libraries [8]–[10].

While the application of AD is straightforward in theory, the AD changes usually require a careful revalidation of the target code and its numerics. For MPI type correctness, this pertains to the changes in data layouts of the whole program due to the AD type change. In particular, to apply MUST with the TypeART extension to an AD-enhanced code, further complications have to be tackled, e.g., (i) the high count of additional allocations due to the derivative calculation and (ii) the AD-specific MPI interface and communication.

1) The impact of AD on data layouts: In Fig. 1, a typical AD type change applied to a **struct** is shown. The AD type **adouble** replaces the previously used built-in **double**.

```
1
2 struct S {int i; double d[2];};
```

```
#include "adouble.h"
struct S {int i; adouble d[2];};
```

Fig. 1. Left: The original **struct**. Right: The **double** type is replaced by some AD type **adouble**. It encapsulates the original value and additionally the corresponding derivative value, or records the operation, in an implementation dependent manner.

In Fig. 2, the impact of AD on a potential memory layout of the **struct** is shown. The AD type is at least twice the data size of the built-in **double** if it encapsulates derivative information.

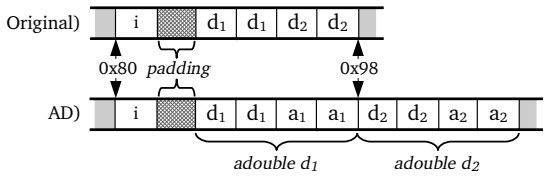

Fig. 2. Impact on the memory layout of the **struct** shown in Fig. 1 after the type change. Each cell represents 4 bytes, padding is added for an 8 byte aligned memory layout (a_i denotes the additional derivative value).

 DOI: 10.1109/Correctness51934.2020.00010

Hence, any memory layout specification can be impacted by these changing data sizes, and, thus, needs to be carefully reexamined in a target program after a type change, see Fig. 3.

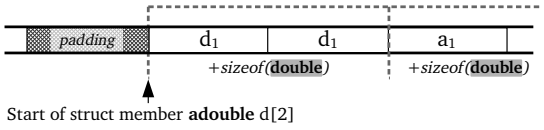

Fig. 3. To construct a MPI derived data type, the offsets to each data member have to be specified. Applying a type change, the offset calculation or pointer arithmetic with, e.g., the **sizeof** operator, needs to change. Otherwise, as shown here, if the user previously extracted the two double values with the **sizeof** operation, the wrong values are extracted due to the new datatype that includes derivative information.

2) Additional allocations: In previous work, we observed that the amount of required allocation tracking is one of the main culprits of performance regression induced by TypeART and MUST in a target code. For efficiency, modern AD tools use template metaprogramming and inlining of calls for the derivative computation, see [11]. As a consequence of the inlining, though, we measured an increase of more than a factor of $2000\times$ for total tracked stack allocations of the AD-enhanced Coral LULESH benchmark [12].

3) AD domain-specific MPI communication: A particular mode of AD, called *reverse mode* (RM), requires the reversal of program execution to calculate the derivative information called *adjoints* [13]. Thus, all MPI calls are replaced for the required program reversal. As a result, for instance, an adjoint MPI send executes internally (i) a vanilla send and, also, at a later time, (ii) a vanilla receive for the reversal. These forward and reverse communication patterns apply to all AD-related MPI communication calls.

B. MUST in the adjoint AD workflow

Integrating MUST into this domain-specific context benefits two groups, (i) the AD expert that develops these domain-specific tools, and (ii) the AD user that applies it or uses an existing AD-enhanced code to compute derivatives.

For the former group, MUST helps verify their AD implementation, especially in the context of large-scale, real-world codes. Here, the interception and analysis of the internal MPI communication of the library are of particular interest. On the other hand, AD users rely on the correctness of these adjoint MPI libraries and overloading tools. Their interest is on a higher semantic level, i.e., they use MUST and TypeART to facilitate development of bug-free MPI communication in their complex HPC codes.

For its analysis capabilities, MUST currently relies on the MPI Profiling interface (PMPI, [3, Sec. 14.2]) to track relevant information for its internal analysis modules. Unfortunately, the AD MPI libraries do not provide such a mechanism, as they (i) do not provide a standardized PMPI-like weak symbol interception and (ii) their API may consist of generic C++ template functions [10], which require MPI function overloading for each template instantiation. To that end, we implement an interface extension to MUST based on XMPT [14], which acts as a callback interface to feed the analysis modules with the required information. XMPT is a callback-based tools interface for XcalableMP, a PGAS approach of the Japanese Exascale initiative. The callback interface replaces the weak symbol interception provided by PMPI. This allows MUST to be configured for either the low- or high-level analysis of AD experts and users, respectively. In summary, we make the following contributions:

- An extension to MUST for analysis based on XMPT.
- An improved allocation filter for the TypeART extension, capable of filtering across object files.
- An AD-enhanced version of LULESH with a MUST-verified adjoint MPI implementation.

The remainder of this paper is structured as follows: Section II introduces AD and adjoint MPI in more detail. MUST, the TypeART extension, and its improved filtering capabilities are introduced in Section III. We also describe the newly developed callback approach of MUST. In Section IV, we evaluate this approach on an AD-enhanced version of the Coral LULESH benchmark. The impact on the type tracking of MUST with TypeART is highlighted. A discussion of our approach is given in Section V. Related work is highlighted in Section VI. Finally, we conclude this work in Section VII.

II. Algorithmic differentiation and MPI

We introduce the fundamentals of the AD RM in Section II-A. MPI and derived datatypes in the adjoint context are introduced in Section II-B. For an extended introduction into AD, see [15] and the AD community portal www.autodiff.org.

A. Fundamentals of adjoint AD

With AD, we assume that each computer program is a composite function $\mathbf{y} = f(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^m$ of elemental functions (e.g., $\sin$) and operations (e.g., $+$) with known derivatives. The original code and its values are called *primal*.

Given f, the RM is based on the adjoint formulation

$$\bar{\mathbf{x}} = \frac{df}{d\mathbf{x}}^T \bar{\mathbf{y}} \equiv J_f^T \bar{\mathbf{y}} \ . \tag{1}$$

$\bar{\mathbf{y}} \in \mathbb{R}^m$ is the vector for the adjoint direction, $\bar{\mathbf{x}} \in \mathbb{R}^n$ is the result of the adjoint formulation and $J_f \in \mathbb{R}^{n\times m}$ is the Jacobian. The adjoints are derivatives of the final result w.r.t. intermediate variables and are propagated in reverse order through the program flow. Applying (1) to each intermediate operation, e.g., a binary operation of scalar values, yields

$$c = \phi(a,b) \mapsto \bar{a} \stackrel{+}{=} \bar{c}\frac{\partial \phi}{\partial a} \ , \ \bar{b} \stackrel{+}{=} \bar{c}\frac{\partial \phi}{\partial b} \ .$$

For instance, using a single assignment notation, the statement $z = ab + \sin(c)$ of scalar values yields

$$\begin{array}{l} \text{Forward} \\ \text{Section} \end{array} \left\{ \begin{array}{llll} \bar{a},\bar{b},\bar{c} = 0 & & \bar{z} = 1 \\ t_1 & = ab & \bar{t}_1,\bar{t}_2 = \bar{z} \\ t_2 & = \sin(c) & \bar{c} \stackrel{+}{=} \bar{t}_2 \cos(c) \\ z & = t_1 + t_2 & \bar{b} \stackrel{+}{=} \bar{t}_1 a \\ & & \bar{a} \stackrel{+}{=} \bar{t}_1 b \end{array} \right\} \begin{array}{l} \text{Backward} \\ \text{Section} \end{array} .$$

As evident, with $\bar{z} \equiv 1$ the final adjoint values of the inputs a,b,c are the entries of J_f. Initializing $\bar{z}$ is called *seeding*.

1) Implementation: In complex codes, statements relevant to the derivative computation typically span many functions and translation units. Hence, for required values during the reversal, all (relevant) operations of the program execution are recorded on a global data structure called *tape* by the AD tool. The derivative of a RM overloading type is, therefore, typically implemented as a pointer to the adjoint on the tape.

B. Adjoint MPI

To compute adjoints in the context of distributed computations with MPI, three libraries [8]–[10] exist. They all provide (i) a subset of modified MPI functions that enable the adjoint of communication, and (ii) a back-end interface that an AD tool has to implement for the required derivative book-keeping. MPI derived datatypes are (partially) supported by the libraries MeDiPack [10] and Adjoinable MPI [8].

1) Adjoint Communication: A MPI communication call can be understood as an assignment operation in the adjoint formulation, see (1). The RM then requires a reversal of the dataflow between the MPI buffers. Hence, all calls to MPI functions in the original program are typically replaced, e.g., MPI_Send $\mapsto$ AMPI_Send. Internally, the AMPI_Send invokes vanilla MPI calls, see Fig. 4.

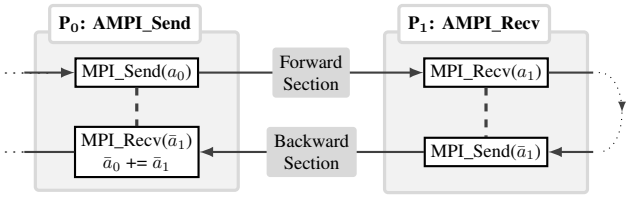

Fig. 4. Adjoint MPI send/recv operation of two processes. Only the primals are sent to P_1, which is equivalent to an assignment $a_1 = a_0$. During the reversal, the adjoints of P_0 need to be updated. Hence, P_1 sends these to P_0, which then updates its local adjoints $\bar{a}_0$. The adjoint MPI library orchestrates the required reversal of the MPI operations.

Likewise, a broadcast, say, becomes a sum reduction during the program reversal. These adjoint communication pair-patterns exist for other MPI calls and remain deadlock-free when applied to a (correct) target code [8].

2) MPI Datatypes: Conceptually, all AD MPI libraries work similarly w.r.t. datatypes. Sending primal datatypes is unchanged, whereas the elementary AD-related datatype is treated as follows: (i) In the forward section, only the primal values are exchanged, while (ii) in the backward section, only the adjoint values are communicated in reverse order. The library handles the extraction of these values.

Derived datatypes are treated similarly. The developer, as before, (i) specifies and passes the typemap to the (overloaded) datatype constructor, and (ii) communicates data with the datatype. Constructed types that include the AD type are treated as follows: An additional datatype is constructed internally that mirrors the overall user-specified typemap. However, the new datatype replaces the AD type with a type field holding the primal. The original data alignments are remapped to account for this change. From a user perspective, the buffer is sent as the user-specified datatype but, internally, a modified buffer is created and communicated. For the program reversal, similar to the primitive datatypes, only the adjoints are communicated for the required updates.

3) Templates: MeDiPack, in particular, provides a generic wrapper class for holding the MPI_Datatype handles. The wrapper holds both the user-specified and adjoint specific datatypes. However, the adjoint MPI calls are template functions that are instantiated for the specific MPI datatype wrapper. Hence, a PMPI-like approach for MUST would require providing function overloads for each such type.

III. MUST and TypeART

MUST intercepts the MPI calls in a target program to feed its analysis modules with information for detecting, e.g., deadlocks or type mismatches of the specified MPI datatypes between the sender and receiver. The TypeART extension allows MUST to compare the type-less **void** buffer and the static MPI datatype for correctness.

TypeART is based on the Clang/LLVM compiler toolchain to (i) statically analyze a target code for type information, and (ii) instrument all allocations relevant to MPI calls. This enables its runtime library to track dynamic type allocation information for MUST to query, see Fig. 5.

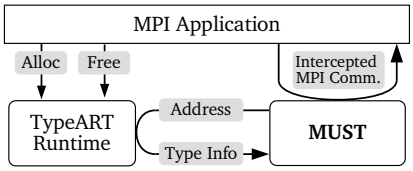

Fig. 5. TypeART as an extension to MUST. For every intercepted MPI call, MUST queries the address of the type-less buffer using TypeART's runtime. The resulting allocation information are compared to the MPI datatype passed to the MPI call. If an inconsistency is detected, MUST reports it.

MUST's callback extension to support the adjoint MPI libraries is discussed in Section III-A. In Section III-B, TypeART is briefly introduced, see also [2] for more implementation details. In particular, we highlight recent additions to TypeART enabling, e.g., better allocation filtering.

A. MUST Callback Interface Extension

MUST is based on GTI (generic tool infrastructure), which can be understood as a distributed multi-agent network. The network itself is implemented as a so-called *Tree-Based Overlay Network* with intra-layer communication [16]. In the classic MPI use case, the sensors of this network consist of MPI function wrappers. To interface with other programming languages, MUST has language-specific sensors like OMPT-based sensors for OpenMP applications or XMPT-based sensors for XcalableMP applications [14]. For the correctness analysis, MUST comes with agents to track state (e.g., creation of communicator handles) and agents for specific analyses.

```
1  int AMPI_Irecv( ... ) {
2    void* tool_data;
3    if (callbacks.MPIAD_Irecv)  // start callback
4      callbacks.MPIAD_Irecv( ... , MPIADT_begin, &tool_data,
5                                   __builtin_return_address(0));
6    ..............  original AMPI_Irecv code  ..............
7    if (callbacks.MPIAD_Irecv)  // end callback
8      callbacks.MPIAD_Irecv( ... , MPIADT_end, &tool_data,
9                                   __builtin_return_address(0));
10   return ampi_ret_val;
11 }
```

Fig. 6. Example of MPI-AD tools callbacks integrated into an AMPI function. The arguments for the callbacks are extracted from the function signature of the AMPI call. This allows for correctness checks on the user-passed buffers, and avoids (i) checking the internally executed MPI communication, see Fig. 4, and, also, (ii) the corresponding, modified buffers.

1) MPI-AD tool interface: We designed the MPI-AD tool interface to provide the information for the analysis by direct calls, disabling the MPI function wrapping. To that end, the necessary callbacks (function pointers) are registered with MUST at program startup, and all relevant AMPI functions are augmented with the appropriate callback. This augmentation needs to be performed once per adjoint MPI library.

The data of the underlying MPI call is passed, and we need additional arguments for book-keeping to emulate the original wrapping of MPI calls with MUST, see Fig. 6. Some of the arguments are valid and of interest only before and others only after the underlying MPI call. In particular, the MPI interface has IN, OUT and INOUT arguments. For this reason, we need two callbacks for each AMPI call and mark the begin and end with a flag. As an example, for MPI_Wait, we need the originally passed request handles in the analysis executed during the end callback. The tool_data argument allows MUST to transfer such information from the begin to the end callback. The code pointer argument set to __builtin_return_address(0)[1] allows MUST to provide source code information in error reports.

B. TypeART

The TypeART framework is shown in Fig. 7.

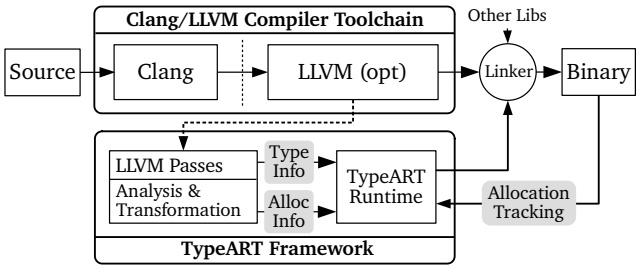

Fig. 7. TypeART framework, adapted from [2]. A target code is compiled with the Clang/LLVM compiler. TypeART extends LLVM with passes to (i) extract static type and allocation information (serialized for use with the runtime), and (ii) instrument all allocations relevant to MPI calls. The runtime library is linked with the target binary and accepts these instrumentation callbacks to provide MUST with the required metadata for type correctness checks.

[1]https://llvm.org/docs/LangRef.html#llvm-returnaddress-intrinsic

1) Compiler Passes: Based on the generated LLVM intermediate representation (IR), the analysis pass (i) collects all heap, stack and global allocations, and (ii) filters these by discarding all allocations not relevant to any MPI call. The instrumentation pass, subsequently, (i) serializes the type information of the allocations (see the following paragraph for details), and (ii) adds an instrumentation hook for our runtime.

In Fig. 8, an instrumentation hook for a heap allocation is shown. The hook passes the (i) memory pointer, (ii) a static type id to determine the allocated type, (iii) the extent of the allocation, and, optionally, (iv) a static allocation id that identifies the code location of the allocation. The latter argument can be used by MUST to, e.g., accurately determine the allocation location where a type mismatch originated from.

```
(double*) malloc(n * sizeof(double));
```

```
1 %1 = call i8* @malloc(i64 %0) // %0 = n * sizeof(double)
2 %2 = udiv i64 %0, 8              // %2 = %0 / sizeof(double)
3 call void @__typeart_alloc( i8* %1,  i32 6,  i64 %2, i32 1 )
4 %3 = bitcast i8* %1 to double*
```

Pointer, Type id, Extent, Alloc id (*opt.*)

Fig. 8. Instrumented LLVM IR of a malloc call. TypeART adds instructions to calculate the extent of the array dynamically (line 2). The callback is shown in line 3: In total, we pass four arguments to our runtime. The type id is determined statically, here 6 for the built-in **double**. Likewise, global and stack allocations are instrumented (not shown for brevity).

2) Type Representation: Type ids are used in the runtime library for identifying the effective type of a pointer at runtime. Built-in types have predetermined ids and layouts. A user-defined type, on the other hand, is handled during the compilation by (i) creating a unique type id, and (ii) serializing its type layout to a database (for runtime lookups).

3) Allocation Filtering: TypeART performs a conservative inter-procedural forward data-flow analysis to enable the filtering of allocations that are not part of an MPI call. For its effectiveness, all relevant function definitions need to be available in the current translation unit (TU), see Fig. 9.

```
1  extern foo_bar(int*);     // The definition is not available at this stage
2  void bar(int* x, int* y) {
3    *x = 2;                 // x is not used after
4    MPI_Isend(y, ...);      // y is passed to an MPI routine
5  }
6  void foo() {
7    int a = 1, b = 2, c = 3;
8    bar(&a, &b);
9    foo_bar(&c);
10 }
```

Fig. 9. Example of three relevant cases, from [2]: The filter follows the allocations of a and b along their data flow. It eventually reaches the definition of bar. (i) The analysis detects a filtering opportunity for a, as the aliasing pointer x is never part of an MPI call. (ii) In contrast, b is instrumented as the aliasing pointer y is part of an MPI call. (iii) Likewise, the allocation c must be instrumented as it is passed to an interface function call.

To remedy the problem of reduced effectiveness of our filter in the case of interface function calls, we implemented a

```
1 "foo_bar": { "callees": [ ], "parents": ["foo"] },
2 "bar": { "callees": ["MPI_Isend"], "parents": ["foo"] },
3 "foo": { "callees": ["bar", "foo_bar"], "parents": [] }
```

Fig. 10. Serialized CG excerpt for Fig. 9: The filter queries for foo_bar, and detects no other *callees*. Hence, the allocation of c can be filtered.

whole-program call-graph (CG) analysis. To that end, we use a Clang-based tool that is part of our performance analysis framework PIRA [17], which builds the CG using the Clang abstract syntax tree in a preprocessing step by (i) first constructing TU-local CG's and, subsequently, (ii) merging them into a single whole-program CG.

The Clang tool works as follows: For the TU-local CG, all function definitions or function calls in a translation unit are visited to construct the relevant call relationships. Functions without definition in this TU referred to as non-local functions, are added as nodes, to be resolved during the subsequent merge step. Special care has to be taken for (i) C++ method calls in an inheritance hierarchy, and (ii) calls based on function pointers. For both cases, the tool tries to build finite sets of potential call targets. In the former case, the tool iterates over the method's inheritance hierarchy and adds all overridden methods to the set of call targets. For the latter case, the function pointer value is queried for its definition to determine the call target (a context-insensitive, TU-local points-to analysis). The merging of these local CG's is straightforward: (i) The actual definition replaces the non-local call target nodes. (ii) All other nodes are simply merged in the final whole-program CG file.

Using the CG, the filter performs a reachability analysis to determine if a path exists from a function call to any MPI call, see Fig. 10. Three distinct cases need to be handled for a call path analysis in TypeART:

i) *reaches*: The call chain contains a MPI call. The allocation is not filtered.
ii) *never reaches*: In contrast, the call chain never reaches such a call. The allocation is filtered.
iii) *maybe reaches*: A call exists in the call chain that, in the CG representation, has no definition but is not an MPI call. This applies to, e.g., C language system library functions such as printf. We consider these functions benign for the filtering analysis as they likely never call any MPI function. The allocation is also filtered.

However, the CG is limited to the call-path information, and does not provide data flow information at the granularity of function arguments. This leads to a conservative cross-TU filter strategy, keeping more allocations than strictly necessary.

IV. Evaluation

We apply the AD tool CoDiPack [11] and the AD MPI library MeDiPack [10] to the Coral LULESH benchmark. We chose these candidates as (i) they provide modern C++ implementations of the adjoint concept, including template meta-programming for efficiency, (ii) MeDiPack is the most feature-complete adjoint MPI library, and, also, (iii) due to our past experience with CoDiPack [18].

In Section IV-A, we describe the main code changes w.r.t. the AD-enhancement of LULESH. Subsequently, in Section IV-B, MUST is applied to the LULESH benchmark variants. We compare the overall impact of our AD-related changes to the original benchmark (henceforth called *primal*), and highlight the significant impact of the heavy use of templates and the corresponding code inlining on our tooling approach. We also contrast the original allocation filtering strategy STD and the newly implemented CG-based filter CG, see Section III-B3. Vanilla refers to either the AD or primal benchmark without a TypeART instrumentation.

A. AD-enhancement of LULESH

The AD-enhancement of LULESH is mostly straightforward. It required (i) the redeclaration of the global basic scalar alias **Real_t**, (ii) the replacement of the MPI routines with corresponding MeDiPack calls, and, also, (iii) seeding and extraction routines calling the CoDiPack API for the adjoint computation.

1) Type change: In Fig. 11, the type change of the global scalar alias is shown. The introduction of (generic) wrapper functions for several C-library IO-related functions used in the LULESH code base was required, as they are incompatible with user-defined types.

```
1  #include "codi.hpp"
2  using AD_real = codi::RealReverse; // RM AD overloading type
3  using Real_t = AD_real;  // AD_real replaces built−in double
4  template<typename ... Args>
5  void printf_oo(const char *fmt, Args &&... args) {
6    printf(fmt, detail::value(std::forward<Args>(args))...);
7  }
8  template<typename ... Args>
9  void fprintf_oo(FILE* f, const char* fmt, Args&&... args) {...}
10 template<typename ... Args>
11 void sprintf_oo(char* buf, const char* fmt, Args&&... args) {...}
```

Fig. 11. Required type definition for CoDiPack. The three math helper function LULESH defines (i.e., SQRT, FABS, CBRT) had to be defined for the AD_real type. They are, however, simple passthrough implementations to the equivalent CoDiPack overloads (not shown for brevity). The calls to C language variadic functions have been replaced by generic wrappers (line 4–11) using template parameter pack extension[3]. For the AD type, the helper function detail::value extracts the primal value before passing it to the C function (not shown for brevity). For built-in types, the function simply forwards the value.

2) Adjoint MPI-related changes: In Fig. 12, an excerpt of the changes to the MPI communication calls is shown. The source-level impact is mostly limited to changing the prefix.

3) Main time-stepping compute loop: The main compute loop is augmented with API calls to CoDiPack for seeding and extracting the derivative values. For each time step, in a black-box fashion, the derivative of the energy (e) at the origin of the domain w.r.t. the pressure (p) is computed. The tape is reset after each time step. The values computed by

[3]https://en.cppreference.com/w/cpp/language/parameter_pack

```
1  auto baseType = ampi_datatype<Real_t>();
2  AMPI_Comm_rank(AMPI_COMM_WORLD, &myRank);
3  AMPI_Irecv(&domain.commDataRecv[pmsg * maxPlaneComm],
4             recvCount, baseType, fromRank, msgType,
5             AMPI_COMM_WORLD, &domain.recvRequest[pmsg]);
```

Fig. 12. Modified MPI communication routines in LULESH (gray marker). In line 1, a function to select the MPI datatype using template specializations was introduced. All other communication routines were modified similarly.

AD LULESH regarding, e.g., the calculated error, agree up to round-off with the values of the primal.

4) AD-enhancement process: For a self-contained code base like LULESH, overall only few code changes are required. For more complex code bases, the change of the built-in floating-point type to a user-defined AD type can lead to several complications that have to be fixed by the AD expert. Complications typically arise due to (i) the different treatment of built-ins compared to user-defined types by the C++ language causing compilation errors [19], and (ii) the usage of external libraries [18], [20] and C-language function calls [21] that are not compatible with the AD type (as seen in LULESH). While tools exist to (partially) automate the process of the type change [22], these may not be able to handle, e.g., external solver libraries, which require special treatment in the adjoint context [23].

B. Evaluating LULESH

The benchmarks were run on compute nodes of the Lichtenberg high-performance computer of TU Darmstadt, with two Intel Xeon 2680v3 processors at a fixed frequency of 2.5 GHz and 64 GB RAM. To compile the benchmarks, we use the Clang compiler 10.0.0 with Open MPI 4.0.3. The results are the median over five runs.

The default optimization flag is used (-O3). Debug information (-g) was additionally included for the MUST tool to generate useful diagnostics including source code references. The benchmark is executed with 8 processes equally distributed on two compute nodes. We cap the iteration count of the compute loop at 200 to keep the AD-induced overhead manageable. All other LULESH parameters are the default values. An additional process is reserved for MUST, as it uses a separate MPI process for correctness analysis.

1) MPI communication impact: In Fig. 13, the additional MPI communication impact of AD for our particular LULESH configuration on rank 0 is shown. The other 7 processes have slightly different send and receive operation counts in the compute loop. Overall, however, they have the same total number of point-to-point communication calls.

2) Static coverage: In TABLE I, the static instrumentation statistics of TypeART are shown. The increased count of instrumented memory-related operations of the AD version can be explained by the header-based template class design of the adjoint libraries, which causes the inlining of their codes. Hence, a high number of internal operations and instantiated templates (each parsed as a unique type in the LLVM IR) are

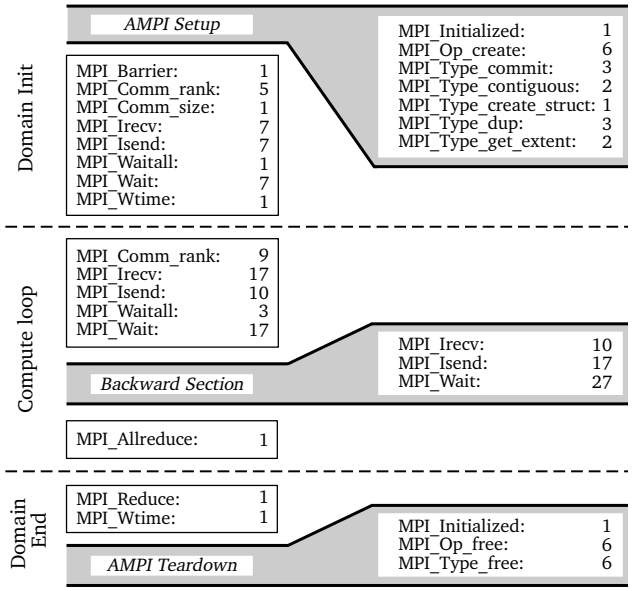

Fig. 13. MPI communication call count trace summary of P_0 (between init and finalize). Right: Additional, AD-induced MPI calls of the MeDiPack library. For each compute loop iteration, a reversal of the communication is executed for the adjoints. In contrast, the first and last phases are executed once, respectively. Note: MeDiPack implements the MPI_Allreduce as a MPI_Allgather and subsequent MPI_Reduce_local calls (not shown).

TABLE I
INSTRUMENTATION STATISTICS. NOTE: STACK AND GLOBAL REPRESENT THE FILTERED COUNT. FILTER PERCENTAGE IN BRACKETS [%].

LULESH	Heap	Free	Filter	Stack [%]	Global [%]	Types
Primal	14	6	CG	19 [64.8]	0 [100]	10
			STD	32 [40.7]	0 [100]	
AD	289	434	CG	72 [96.4]	2 [99.8]	147
			STD	615 [68.9]	5 [99.0]	182
AD [CB]	289	434	CG	114 [94.4]	2 [99.8]	145

detected by the TypeART pass and subsequently instrumented. These reasons also explain the high number of extracted type information. The type layout is serialized on-demand, whenever an allocation is instrumented. Thus, resulting in the difference between the filter implementations. In particular, AD with the MUST callbacks (AD [CB]), has a different behavior due to the integrated callbacks based on dynamically set function pointers. The filter, thus, performs differently, keeping more stack allocations.

3) Dynamic coverage: In TABLE II, the tracked allocation and MPI type check counts at runtime are shown.

a) Allocation filtering: The juxtaposition of the two filter implementations shows the effectiveness of the CG version for the AD-enhanced LULESH benchmark. With the original STD filter, over 32 million stack variables are tracked overall during the execution, even though the maximum stack depth is only 32. This can be explained by the expression templates (and the inlining), which introduces more stack variables that are subsequently tracked but also regularly discarded. The total number is due to missing crucial temporaries in a hot

TABLE II
RUNTIME STATISTICS FOR (I) TRACED MEMORY OPERATIONS, AND (II) MPI-RELATED TYPE CHECKS. THE MEDIAN OF ALL PROCESS VALUES IS SHOWN.

	Traced Memory Operations						MPI Type Checks	
LULESH	Tot. Heap	Filter	Tot. Stack	Tot. Global	Max. Stack	Max. Heap	Total	Unique
Primal	40,063	CG	1,816	0	17	79	5,813	30
		STD	2,624	0	21			
AD	71,344	CG	13,816	2	18	223	11,213	524
		STD	32,429,228	8	32			544
AD [CB]	71,344	CG	28,246	2	31	223	5,813	30

kernel function of the code. The new filter, on the other hand, eliminates these and, thus, reduces the total tracked stack variables by a factor of $2{,}347\times$. The AD [CB] version approximately doubles the number of tracked stack operations compared to the AD version due to the additional tracking caused by the callback interface. In contrast, for the primal, the CG filter reduces the tracked stack allocations by a factor of $1.45\times$. The internal allocation of buffers by the adjoint MPI library explains the AD-related heap operations.

b) MPI type checks: The number of MPI type checks for AD approximately doubles compared to the primal. This is due to the reversal of send and receive operations in the compute loop. In particular, for the primal, we observe $5{,}414$ and, hence, doubling to $10{,}828$ point-to-point communication calls for the primal and AD, respectively. The rest consists of intercepted collective operations where MUST analyzes send and receive semantics separately. With AD [CB], as expected, we reduce the checks to that of the primal.

For AD, the higher number of unique address checks, i.e., the distinct memory addresses MUST passes to TypeART to query type information, is explained by the internal buffers of MeDiPack for sending the datatypes, see Section II-B2. Only for the AD benchmark, based on the filter implementation, we observe a difference between these counts. TypeART instruments a different number of allocations based on the filter, which likely affects the compiler to apply code transformations and optimizations for code generation.

4) Runtime: The relative runtime overhead is shown in Fig. 14. Per loop iteration, the overhead factor of AD for a time step is approximately $9\times$. The subsequent tape evaluation for the adjoints results in a combined factor total of about $11\times$ compared to the vanilla benchmark. In contrast, TypeART itself induces only little overhead. Comparing filter strategies: The absolute runtime savings are 5 s for TypeART with the new filter (not shown), and for MUST with TypeART we measure about 8 s time saving for the AD benchmark. For AD [CB], although more stack allocations are tracked, the reduced type checks bring the performance to the level of MUST without TypeART's tracking. In contrast, the primal runtime configuration differences are negligible.

5) Memory: The relative memory overheads are shown in Fig. 15. The memory overhead factor of AD vanilla compared to the primal vanilla is approximately $1.4\times$. TypeART's induced overhead is less than 4 MB for both variants compared to vanilla. MUST combined with TypeART adds approximately $9 \sim 12$ MB compared to vanilla. The filter strategy has almost no impact on memory savings for MUST.

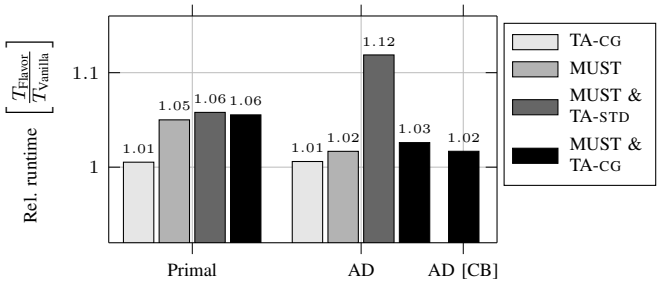

Fig. 14. The relative runtime overhead w.r.t. vanilla. Vanilla primal runtime: 7.59 s. Vanilla AD runtime: 83.52 s.

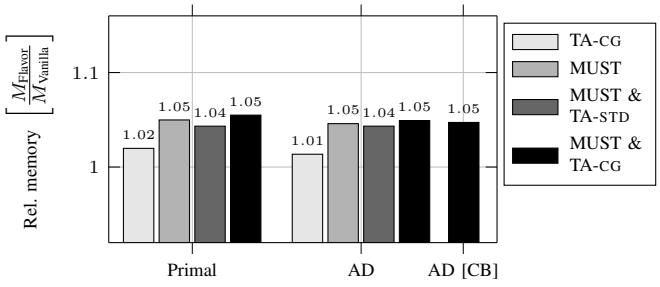

Fig. 15. The relative median memory overhead of a single MPI process w.r.t. vanilla. Vanilla primal RSS: 171 MB. Vanilla AD RSS: 242 MB.

V. DISCUSSION

A recent study of about 100 MPI applications [24] has shown, that most codes require MPI-2 or lower and point-to-point communication and collectives are the majority of used MPI features. However, derived MPI datatypes are the fourth most used feature in this study. Ensuring type correctness of such advanced features with MUST and TypeART will help the overall adoption thereof. This also applies to AD-enhanced MPI codes, as any previous memory layout assumption may no longer be valid with the new AD data layouts, especially when low-level pointer arithmetic is used. Hence, a tool like TypeART will only gain importance going forward.

A. Allocation filtering

The evaluation has shown that the application characteristic w.r.t. memory operations significantly changes with AD. Without any filtering, for our AD configuration, TypeART tracks

about 600 million stack operations per process (compared to about 6,800 for the primal), and the runtime is about 1,060 s. The original TypeART filtering mechanism, in contrast, reduced (i) the tracking of stack allocations by a factor 18×, and (ii) the runtime to about 83 s.

However, original filter missed many additional stack memory operations that are not part of the MPI communication. This is partly due to the data-flow tracking not accurately handling the additional nested template call hierarchies of the AD tool. Therefore, improving the filtering mechanism was worthwhile and reduced runtime by about 10% without loss of type-tracking precision.

The improved filter relies on all user-code function bodies being correctly identified with the CG-generator tool, as functions without a body are interpreted as system library functions that never call any MPI routine, see Section III-B3 (*maybe reaches*). If this assumption does not hold, we may filter allocations erroneously. Hence, if a call to a system library, e.g., a pre-installed parallel solver, uses MPI internally, the CG-generator tool needs to explicitly handle these library calls. To that end, the CG-generator tool can be extended by a plugin, or explicit system header annotations to treat such CG nodes as call targets where filtering is not allowed. In addition, changes to the target code w.r.t. adding or removing functions require a re-generation of the call graph, otherwise filtering may become ineffective or erroneous.

B. MPI type checks

The callback interface (AD [CB]) reduces the required type checks to that of the primal version of LULESH. AD users are mostly interested in the correctness of their (adjoint) MPI usage w.r.t. deadlocks or usage of derived datatypes. Therefore, it is sufficient to keep the analysis limited to, e.g., the datatype buffer passed to the adjoint MPI call: If the buffer of the datatype is incorrect, likely the internal handling of this buffer will also be incorrect and vice versa.

The difference of unique address checks for the filter implementations of the AD LULESH benchmark did not cause any complications. Likely this is code generation related, as we also observed stack variable address reuse during execution for AD with the STD filter implementation. This behavior can, e.g., be controlled with compiler flags and is typically activated for all stack variables.[4] However, a detailed analysis of the produced assembly and the impact of the filtering on the code generation is out of scope for this work.

C. Defects

We have not detected any type error. Initially, MUST revealed a datatype that was not correctly freed before finalize was called, originating from the MeDiPack library. This has since been fixed.

[4] `-fstack-reuse=reuse-level`, see https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html

VI. Related Work

Several MPI correctness checkers exist, e.g., [25]–[28]. However, we are not aware of any MPI correctness checker being applied to adjoint MPI codes. Especially static MPI checkers likely require adaptation for integrating the different communication routine signatures of these adjoint libraries. In addition, finding defects in the implementation of adjoint libraries, e.g., deadlocks in the reversal patterns, is not straightforward with a static analyzer compared to the dynamic MPI checker MUST.

A. MPI correctness checker

A discussion of previous related work to MUST and TypeART can be found in [2]. More recently, in [28], the authors use static analysis and symbolic code execution to find MPI defects. Regarding buffer type matching, for each creation of MPI derived types the primitive type components are tracked. This approach fails, if the analysis can not detect the effective type of a void buffer to compare with the tracked datatypes.

B. Adjoint MPI applications

The computation of adjoints in a distributed context has been done for large scale software packages in the past for, e.g., sensitivity studies or model optimizations [18], [29]–[31].

To highlight the relevancy of our approach, we briefly discuss the adjoint implementation of the CFD solver OpenFOAM [29]. The distributed computations of OpenFOAM are based on a communication wrapper around MPI, which does not pass type information to the low-level MPI communication routines. Instead, before passing a buffer to the MPI call, the wrapper serializes the data to a **char** array. Hence, data is communicated as a MPI_BYTE datatype. For correct reverse propagation of adjoints, the AD experts had to, therefore, implement a manual type detection scheme: (i) If primal values are exchanged, the communication is left unchanged, however, (ii) if an AD-related type is sent, the adjoint MPI library [9] is called instead. We believe the correctness of this approach can be (further) verified with MUST and TypeART.

VII. Conclusion

We presented extensions to MUST and TypeART that allows for analysis of applications with domain-specific MPI communication. In particular, for MUST, we developed a callback interface that can replace the previous approach of relying on weak-symbol forwarding using the PMPI specification to feed MUST's analysis modules. This enables a domain-specific view of the adjoint MPI libraries where a standardized PMPI-like interface is absent, and where templatized adjoint MPI functions would require the individual provision of function overloads for each instantiated template function.

The additional stack operations induced by the AD tool to compute the derivatives significantly affects the amount of required allocation tracking. Applying the MUST tool with the TypeART extension using the original filtering strategy, thus, showed an overhead of factor 1.12× due to tracking a total of about 32 million stack operations. As a consequence, we

implemented a whole-program call-graph filter which reduces the overhead to $1.03\times$, lowering the overhead close to the level of vanilla MUST without TypeART. The new MUST callback extension combined with the new filter, on the other hand, brings performance to the level of vanilla MUST at $1.02\times$.

For future work, a CG-based filter with per function argument data-flow tracking seems worthwhile. We will also apply the MUST callback extension to the other adjoint MPI libraries. Automating this process is the next step.

The TypeART library is available in the source code repository of the institute for Scientific Computing at TU Darmstadt, see https://github.com/tudasc/typeart. The MUST extension and AD LULESH are available upon request.

Acknowledgment

This work was funded by the Hessian LOEWE initiative within the Software-Factory 4.0 project, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — Project-ID 265191195 — SFB 1194, and by the European Union's Horizon 2020 research and innovation program under grant agreement 824080. Calculations were performed on the Lichtenberg cluster at TU Darmstadt. We thank Max Sagebaum for his advice w.r.t. AD LULESH and MeDiPack.

References

[1] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Müller, "MPI runtime error detection with MUST: Advances in deadlock detection," *Scientific Programming*, vol. 21, no. 3-4, pp. 109–121, 2013.

[2] A. Hück, J.-P. Lehr, S. Kreutzer, J. Protze, C. Terboven, C. Bischof, and M. S. Müller, "Compiler-aided type tracking for correctness checking of MPI applications," in *2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness)*, 2018, pp. 51–58.

[3] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard, Version 3.1," 2015, *last visited* Oct 2020. [Online]. Available: www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[4] A. Griewank and A. Walther, *Evaluating Derivatives*, 2nd ed. SIAM, 2008.

[5] U. Naumann, *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. SIAM, 2012, vol. 1.

[6] G. R. Carmichael, A. Sandu *et al.*, "Sensitivity analysis for atmospheric chemistry models via automatic differentiation," *Atmospheric Environment*, vol. 31, no. 3, pp. 475–489, 1997.

[7] M. Asch, M. Bocquet, and M. Nodet, *Data Assimilation: Methods, Algorithms, and Applications*. SIAM, 2016.

[8] J. Utke, L. Hascoet, P. Heimbach, C. Hill, P. Hovland, and U. Naumann, "Toward adjoinable MPI," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–8.

[9] M. Schanen, U. Naumann, L. Hascoët, and J. Utke, "Interpretative adjoints for numerical simulation codes using MPI," *Procedia Comput. Sci.*, vol. 1, no. 1, pp. 1825–1833, 2010.

[10] M. Sagebaum and N. R. Gauger, "MeDiPack — Message Differentiation Package," 2020, *last visited* Oct 2020. [Online]. Available: www.github.com/scicompkl/medipack

[11] M. Sagebaum, T. Albring, and N. R. Gauger, "High-performance derivative computations using CoDiPack," *ACM Trans. Math. Softw.*, vol. 45, no. 4, 2019.

[12] "CORAL benchmark codes," *last visited* Oct 2020. [Online]. Available: https://asc.llnl.gov/CORAL-benchmarks/

[13] R. Giering and T. Kaminski, "Recipes for adjoint code construction," *ACM Trans. Math. Softw.*, vol. 24, no. 4, pp. 437–474, 1998.

[14] J. Protze, C. Terboven, M. S. Müller, S. G. Petiton, N. Emad, H. Murai, and T. Boku, "Runtime correctness checking for emerging programming paradigms," in *Proceedings of the First International Workshop on Software Correctness for HPC Applications*, ser. Correctness'17. ACM, 2017, pp. 21–27.

[15] A. H. Gebremedhin and A. Walther, "An introduction to algorithmic differentiation," *WIREs Data Mining and Knowledge Discovery*, vol. 10, no. 1, p. 21, 2020.

[16] T. Hilbrich, J. Protze, B. R. de Supinski, M. Schulz, M. S. Müller, and W. E. Nagel, "Intralayer communication for tree-based overlay networks," in *42nd International Conference on Parallel Processing, ICPP*, 2013, pp. 995–1003.

[17] J.-P. Lehr, A. Hück, and C. Bischof, "PIRA: Performance instrumentation refinement automation," in *Proceedings of the 5th ACM SIGPLAN International Workshop on Artificial Intelligence and Empirical Methods for Software Engineering and Parallel Computing Systems*, ser. AI-SEPS 2018. ACM, 2018, pp. 1–10.

[18] A. Hück, C. Bischof, M. Sagebaum, N. R. Gauger, B. Jurgelucks, E. Larour, and G. Perez, "A usability case study of algorithmic differentiation tools on the ISSM ice sheet model," *Optim. Method. Softw.*, vol. 33, no. 4–6, pp. 844–867, 2018.

[19] A. Hück, J. Utke, and C. Bischof, "Source transformation of C++ codes for compatibility with operator overloading," *Procedia Comput. Sci.*, vol. 80, pp. 1485–1496, 2016.

[20] M. Sagebaum, N. R. Gauger, U. Naumann, J. Lotz, and K. Leppkes, "Algorithmic differentiation of a complex C++ code with underlying libraries," *Procedia Comput. Sci.*, vol. 18, pp. 208–217, 2013.

[21] A. Hück, S. Kreutzer, D. Messig, A. Scholtissek, C. Bischof, and C. Hasse, "Application of algorithmic differentiation for exact jacobians to the Universal Laminar Flame solver," in *Computational Science - ICCS 2018*, ser. Lecture Notes in Computer Science. Springer, 2018, vol. 10862, pp. 480–486.

[22] M. O. Lam, T. Vanderbruggen, H. Menon, and M. Schordan, "Tool integration for source-level mixed precision," in *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, 2019, pp. 27–35.

[23] U. Naumann, J. Lotz, K. Leppkes, and M. Towara, "Algorithmic differentiation of numerical methods: Tangent and adjoint solvers for parameterized systems of nonlinear equations," *ACM Trans. Math. Softw.*, vol. 41, no. 4, Oct. 2015.

[24] I. Laguna, R. Marshall, K. Mohror, M. Ruefenacht, A. Skjellum, and N. Sultana, "A large-scale study of MPI usage in open-source HPC applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC'19. ACM, 2019, pp. 31:1–31:14.

[25] L. Glenn, C. Hua, C. James, H. Jim, K. Marina, and Z. Yan, "MPI-CHECK: a tool for checking Fortran 90 MPI programs," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 2, pp. 93–100, 2003.

[26] A. Droste, M. Kuhn, and T. Ludwig, "MPI-Checker: Static Analysis for MPI," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. ACM, 2015, pp. 3:1–3:10.

[27] C. C. Douglas and K. Krishnamoorthy, "Static analysis and symbolic execution for deadlock detection in MPI programs," in *Computational Science – ICCS 2018*. Springer, 2018, pp. 783–796.

[28] F. Ye, J. Zhao, and V. Sarkar, "Detecting MPI usage anomalies via partial program symbolic execution," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 794–806.

[29] M. Towara and U. Naumann, "A discrete adjoint model for OpenFOAM," *Procedia Comput. Sci.*, vol. 18, pp. 429–438, 2013.

[30] 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," *Geosci. Model Dev.*, vol. 9, no. 11, pp. 3907–3918, 2016.

[31] T. A. Albring, M. Sagebaum, and N. R. Gauger, "Development of a consistent discrete adjoint solver in an evolving aerodynamic design framework," in *16th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference*. AIAA, 2015.