Event-Based Automatic Differentiation of OpenMP with OpDiLib

JOHANNES BLÜHDORN, MAX SAGEBAUM, and NICOLAS GAUGER, Chair for Scientific Computing, Technische Universität Kaiserslautern

We present the new software OpDiLib, a universal add-on for classical operator overloading AD tools that enables the automatic differentiation (AD) of OpenMP parallelized code. With it, we establish support for OpenMP features in a reverse mode operator overloading AD tool to an extent that was previously only reported on in source transformation tools. We achieve this with an event-based implementation ansatz that is unprecedented in AD. Combined with modern OpenMP features around OMPT, we demonstrate how it can be used to achieve differentiation without any additional modifications of the source code; neither do we impose a priori restrictions on the data access patterns, which makes OpDiLib highly applicable. For further performance optimizations, restrictions like atomic updates on adjoint variables can be lifted in a fine-grained manner. OpDiLib can also be applied in a semi-automatic fashion via a macro interface, which supports compilers that do not implement OMPT. We demonstrate the applicability of OpDiLib for a pure operator overloading approach in a hybrid parallel environment. We quantify the cost of atomic updates on adjoint variables and showcase the speedup and scaling that can be achieved with the different configurations of OpDiLib in both the forward and the reverse pass.

CCS Concepts: • Mathematics of computing → Automatic differentiation; Mathematical software performance; • Computing methodologies → Shared memory algorithms; • Software and its engineering → Software design engineering;

Additional Key Words and Phrases: Algorithmic differentiation, OpenMP, OMPT, high performance computing, AD tool design, reusable software

ACM Reference format:
Johannes Blühdorn, Max Sagebaum, and Nicolas Gauger. 2023. Event-Based Automatic Differentiation of OpenMP with OpDiLib. ACM Trans. Math. Softw. 49, 1, Article 3 (March 2023), 31 pages. https://doi.org/10.1145/3570159

1 INTRODUCTION

Automatic differentiation (AD) deals with the evaluation of derivatives of computer codes, in a machine accurate manner and for a computational cost that is often only a small multiple of the cost of the codes themselves. AD has repeatedly proven its value in scientific computing, especially for large simulation codes where symbolic derivatives are not feasible and finite differences produce numerically unstable and unreliable results. Areas of application include discrete adjoint solvers.
We view a computer program with input variables $x$ and output variables $y$ as a mathematical function $F: \mathbb{R}^n \rightarrow \mathbb{R}^m: x \mapsto y$. While an execution of this program for a specific input $x$ might involve millions of code statements, it always amounts to a sequence of elementary operations like sums, products, or standard math library functions like sin or exp. For all these operations, the derivatives are known; hence, we can evaluate the derivative of the full code with respect to $x$ by a rigorous application of the chain rule. The reverse mode of AD is a specific strategy for the structured traversal of the chain rule. For each variable $v$ of the code, whether input, output, or intermediate, we introduce an adjoint variable $\bar{v}$ that fulfills the relation $\bar{v} = \frac{dF}{dv}(v)^T \bar{y}$. For the adjoints $\bar{x}$ associated with the inputs $x$, we obtain

$$\bar{x} = \frac{dF}{dx}(x)^T \bar{y},$$

so that suitable choices of the adjoints $\bar{y}$ associated with the outputs $y$ yield various derivatives of $F$, and hence the computer program at hand. Choosing $\bar{y}$ is known as seeding. In an implementation, it is practical to assemble the adjoint variables in an incremental fashion statement by statement. Let a code statement be given by

$$w = \phi(u),$$

(1)

where $w \in \mathbb{R}$ is an intermediate variable or an output, $u \in \mathbb{R}^k$ consists of inputs and intermediate variables from previous statements, and $\phi: \mathbb{R}^k \rightarrow \mathbb{R}$ is composed of one or multiple elementary operations. The adjoint updates for the statement read

$$\bar{u}_j := \frac{\partial \phi}{\partial u_j}(u)\bar{w} \quad \text{for } j = 1, \ldots, k,$$

(2)

then set $\bar{w} = 0$. Compared to the data flow implied by a sequence of statements (1), the data flow of the corresponding adjoint updates (2) is reversed. Derivative information is propagated from $\bar{y}$ via adjoint intermediate variables to $\bar{x}$. This data flow reversal challenge can, for example, be approached as follows. First, the code is executed in its original structure in a forward pass; throughout, it is recorded in a suitable representation on a stack-like structure called tape. The tape stores all information required for the reversal of all statements, that is, identification for the adjoint variables $\bar{u}$ and $\bar{w}$ [Sagebaum et al. 2020], and information for the computation of the derivatives $\frac{\partial \phi}{\partial u_j}(u)$ [Hogan 2014; Sagebaum et al. 2018]. The order in which all statements (1) were executed in the forward pass is preserved in their recording on the tape. In a subsequent reverse pass, the tape is evaluated, that is, it is unwound from end to start to provide the data for the corresponding adjoint updates (2) in reverse order.

A key objective of AD tools is the automatic augmentation of a given source code by extra logic required for the computation of derivatives. The two main approaches for this are source transformation and operator overloading. Source transformation tools like Tapenade [Hascoët and Pascual 2013] are essentially compilers that take the given source code and transform it directly to the augmented source code. Operator overloading tools rely on a replacement of the floating point data type by a custom AD type. All elementary operations are overloaded for the AD type. From the user’s point of view, the AD type behaves like a floating point type; internally, the overloads execute the extra logic that is required for differentiation. Operator overloading tools
usually implement the taping approach described above, examples include ADOL-C [Walther 2009], Adept [Hogan 2014], and CoDiPack\(^1\) [Sagebaum et al. 2019].

In order to achieve maximum performance on today’s multicore architectures, simulation codes typically employ shared memory parallelism, for example, by means of OpenMP [OpenMP ARB 2021]. Differentiating such codes requires AD tools that can handle OpenMP directives. Research on combining AD and OpenMP started in the realm of source transformation tools when Bücker et al. [2001, 2002] suggested the generation of OpenMP parallel code for the simultaneous computation of multiple derivatives of an otherwise serial code with the forward vector mode. Bücker et al. [2004] extended this to the case of an already parallel code by using nested parallelism. First reports on source-to-source reverse mode differentiation of an OpenMP parallel simulation code are given by Giering et al. [2006] and Heimbach et al. [2002, 2005]. Subsequently, support for reverse mode AD for OpenMP parallel codes was established in operator overloading tools as well, but faced the challenge that OpenMP parallelism is specified in large parts via \#pragma directives. Techniques like operator overloading do not apply, and unlike, e.g., an MPI function call, an OpenMP directive may influence the control flow beyond its immediate location in the code. For a parallel region, e.g., the compiler will introduce thread handling both before and after the contained code. Hence, it is much harder to change the directives’ behavior without rewriting the source code. ADOL-C relies on the side effects of constructors and destructors of privatized objects to establish OpenMP support for parallel regions and parallel loops [Bischof et al. 2008; Kowarz 2008]. The source code rewriting can be reduced to a manual annotation of the respective pragma directives for these OpenMP features, as opposed to, e.g., annotations both at the beginning and end of parallel regions. Recently, it was shown in conjunction with Adept how compiler instrumentation can be used to introduce extra logic for the differentiation of general recursive fork-join programs in a fully automatic fashion [Kaler et al. 2021]. Source transformation tools, on the other hand, have unlimited access to OpenMP directives. Transformation rules for many additional OpenMP directives were established in Förster [2014]. Recently, capabilities to differentiate OpenMP parallel worksharing loops have been added to Tapenade, with a particular focus on the appropriate treatment of variable scopes and execution schedules [Hückelheim and Hascoët 2022].

We summarize some key insights from previous research. Transitioning from the reverse mode of AD in a serial context as described above to parallel codes, we face the problem that taping is an inherently sequential process. Therefore, computations that are performed simultaneously in different threads have to be recorded on different tapes. Furthermore, it was observed that the parallelization of the forward pass can be used to deduce a corresponding parallelism for the reverse pass. However, this is—except for special cases—not as simple as evaluating the tapes in parallel because any additional synchronization that occurred in the forward pass has to be transformed to corresponding reverse synchronization. Furthermore, while simultaneous read access to the same variable from different threads is fine in the forward pass, the corresponding adjoint updates (2) result in simultaneous write access to the same adjoint variable, which introduces data races in the reverse pass. This can be addressed conservatively by performing all adjoint updates atomically [Förster 2014]. In general, it is challenging to automatically identify variables that need atomic updates. Recently, an efficient parallel reversal approach was presented that does not rely on atomic updates [Kaler et al. 2021]. Instead, incremental adjoint contributions are stored in dedicated memory locations, and their reduction is delayed until the reversal of the statement that wrote the corresponding primal variable, with the overhead of providing and managing these memory locations. We remark that reductions are a well-known alternative to atomics, see, e.g., [Bischof et al. 2008; Giering et al. 2006; Hückelheim and Hascoët 2022; Kowarz 2008].

\(^1\)https://www.scicomp.uni-kl.de/software/codi/.
In this article, we present a novel event-based approach for the reverse mode automatic differentiation of OpenMP parallel codes with operator overloading tools, and its implementation in the new open source tool OpDiLib\textsuperscript{2}. Starting with version 5.0 of the OpenMP specification \cite{OpenMPARB2018}, internal parts of the OpenMP runtime are formalized by execution model events. Furthermore, OMPT becomes part of the specification, an API that allows tool developers to observe and react to these events \cite{Eichenberger et al. 2013}. OMPT was originally designed to provide a standardized interface for the performance monitoring of OpenMP applications; more generally, it allows for the coupling of the OpenMP runtime with tools that execute in the address space of the OpenMP parallel program \cite{OpenMPARB2021}, which is precisely the case for operator overloading AD tools. In this work, we identify and demonstrate the applicability of OMPT for automatic differentiation. Specifically, it allows us to implement augmentations required for AD handling as side effects of OpenMP constructs, without any additional modifications of the OpenMP parallel source code.

In order to support OpenMP in an operator overloading AD tool, two kinds of adaptions must be made. First, the AD tool itself must be thread-safe. Second, an appropriate handling of OpenMP mechanisms must be provided. While certain OpenMP support has been crafted into some operator overloading AD tools by now, we demonstrate with OpDiLib for the first time that the second part can be implemented independently of a specific operator overloading AD tool. Moreover, this implementation is based on a limited set of features that many AD tools already have, which allows us to design OpDiLib as a universal add-on for classical operator overloading AD tools. Support for OpenMP in such a classical operator overloading tool is then achieved by making it thread-safe in the first step, and coupling it with OpDiLib in the second step. This continues a philosophy of AD tool agnostic support for parallelism in operator overloading that was established for MPI\textsuperscript{3} in the tools AMPI \cite{Schanen et al. 2010} and MeDiPack\textsuperscript{4}. The coupling is achieved by implementing an AD tool wrapper, which we detail in Section 2. OpDiLib bindings are already implemented for CoDiPack. This article should serve as a useful starting point for coupling OpDiLib with other operator overloading AD tools in the future.

OpDiLib’s event-based differentiation logic is presented in Section 3. It is formulated in terms of AD events that mark the beginnings and ends of OpenMP constructs, and the AD handling is implemented as reactions to these events. This corresponds to an AD-centered point of view that allows for a unified handling of similar OpenMP mechanisms and hides many OpenMP and OMPT specific details that are not relevant for AD. Furthermore, the differentiation logic itself does not depend on OMPT at all. While OMPT execution model events can be used to generate AD events, the differentiation logic can also be reused together with other AD event generation mechanisms. With this approach, we establish in Section 3.1 operator overloading AD support for the constructs of the OpenMP 2.5 specification \cite{OpenMPARB2005}, with the exception of atomics and the flush directive, including several constructs that were only treated by source transformation tools previously.

From a user’s point of view, OpenMP is very easy to apply. Introducing parallelism to a code can be as straightforward as supplying a loop with a \texttt{#pragma omp parallel for} directive. To achieve maximum parallel performance, on the other hand, additional restructuring of the code and the application of more advanced OpenMP features might be required. OpDiLib follows the same philosophy. With OMPT, a first differentiated version of an OpenMP parallel code can be obtained without any additional code changes and without restrictions on the data access patterns. We

\footnotesize\textsuperscript{2}https://www.scicomp.uni-kl.de/software/opdi/.
\footnotesize\textsuperscript{3}https://www.mpi-forum.org/.
\footnotesize\textsuperscript{4}https://www.scicomp.uni-kl.de/software/medi/.
achieve this by defaulting to atomic updates on adjoint variables. Starting from there, a user may employ additional optimizations. An important part of improving the performance of the parallel reverse pass is the elimination of atomic updates, which might only be possible after a revision of the data access patterns. Then, OpDiLib’s adjoint access control tools which we present in Section 3.2 can be used to disable atomic updates for selected parts of the code.

Section 4 presents two different backends for the generation of AD events. Section 4.1 details the generation of AD events via OMPT execution model events. OpDiLib is designed to be used with these modern OMPT features. Nonetheless, compilers are still in the process of establishing support for OMPT. As many projects rely on older compiler versions or compilers without OMPT features, we provide a second backend that does not require OMPT. It is based on an alternative interface to OpenMP functionality that consists of replacement macros for OpenMP 

#pragma directives and clauses as well as replacements for OpenMP’s runtime functions. The only additional step required for its application is the replacement of OpenMP constructs. As the macro interface is fully compatible with the OMPT backend, these code changes do not have to be revisited for a transition from the macro backend to the OMPT backend. Section 4.2 provides an overview of the macro interface and highlights some implementational details of the macro backend.

Together, the AD tool wrapper, the logic layer, and the backend layer make up OpDiLib’s layered architecture. Figure 1 visualizes the essential relations between these layers, with further details given in the respective Sections 2, 3, and 4.

In Section 5, we conduct a performance study of OpDiLib applied to an OpenMP-MPI hybrid parallel code. We use CoDiPack as the underlying AD tool and MeDiPack for the differentiation of MPI. While a mixed approach between operator overloading and source transformation was established for hybrid parallel codes by Schanen et al. [2012], we demonstrate with OpDiLib that they can also be handled by pure operator overloading, which admits a simplified toolchain without additional source transformation compilers and does not involve interfaces between different AD paradigms. We compare OpDiLib’s backends, different compilers, quantify the cost of atomic updates on the adjoint vector and demonstrate the impact of OpDiLib on the parallel performance of the forward and reverse pass on two types of processors. We conclude our work in Section 6.
2 AD TOOL COUPLING

In this section, we present the assumptions on the AD workflow that the implementation of OpDiLib is based on and collect the properties and features that an operator overloading AD tool must have in order to be coupled with OpDiLib. The coupling is achieved by an implementation of OpDiLib’s AD tool wrapper interface which we showcase in this section. We close with a discussion of some advanced use cases.

To begin with, we assume that an AD workflow such as depicted in Listing 1 is used, and that the AD tool supports the classical operations involved.

```c
// within serial code
AD::masterTape->registerInputs(inputs);
AD::masterTape->setActive(); // start recording

// function to be differentiated
f(inputs, outputs);

// involves parallel code
AD::masterTape->setPassive(); // stop recording
AD::masterTape->registerOutputs(outputs);
AD::masterTape->seedOutputs(seeds);
AD::masterTape->evaluate(); // reverse pass
AD::masterTape->reset();
```

Listing 1. AD workflow.

In Listing 1, all parallel regions are assumed to be contained in \( f \). Particularly, we assume that the recording is controlled by the initial implicit task, that is, serial parts of the code. Hence, when transitioning to parallel code, the general AD workflow set up for the serial version can be kept. As before, the user controls AD via the tape of the initial implicit task, to which we refer as master tape. OpDiLib performs additional treatment for parallel code in \( f \) in a fully automatic fashion and ultimately embeds appropriate reverse handling into the master tape. In order to be coupled with OpDiLib, an AD tool must additionally

- allow for the creation and destruction of tapes at runtime,
- treat tapes as thread-local objects that can be exchanged at runtime, and on which the classical operations such as recording or evaluation can be performed individually,
- provide access to tape positions and support positional evaluation and positional reset of tapes,
- have capabilities to embed custom function calls into the tape evaluation, commonly referred to as external functions [Kowarz 2008].

Furthermore, the AD tool must be thread-safe for use in an OpenMP parallel environment.

- Recordings that are performed in parallel on different tapes must be thread-safe. Particularly, the simultaneous distribution of indices to active variables in multiple threads must be safe.
- The AD tool must support atomic operations on the adjoint vector. It is sufficient if each evaluation uses atomic access by default. However, to allow for additional performance optimizations, it is preferred that the adjoint access mode can be specified for each individual positional tape evaluation. The most obvious optimization is that serial parts of the code are not evaluated with atomic updates.

The full interface that has to be implemented for coupling OpDiLib with an operator overloading AD tool is shown in Listing 2. It reflects the AD tool requirements identified above. Note that any
specific types used by the AD tool for tapes and tape positions are abstracted away, hence the various `void*` pointers in the interface.

```cpp
struct ADToolWrapperInterface {
    public:
        virtual ~ADToolWrapperInterface() {}

        virtual void init() = 0;
        virtual void finalize() = 0;

        // tape creation and deletion
        virtual void* createTape() = 0;
        virtual void deleteTape(void* tape) = 0;

        // management of thread local tape of caller
        virtual void* getThreadLocalTape() = 0;
        virtual void setThreadLocalTape(void* tape) = 0;

        // position handling
        virtual void* allocPosition() = 0;
        virtual void freePosition(void* position) = 0;
        virtual size_t getPositionSize() = 0;
        virtual std::string positionToString(void* position) = 0;
        virtual void getTapePosition(void* tape, void* position) = 0;

        // tape handling
        virtual bool isActive(void* tape) = 0;
        virtual void setActive(void* tape, bool active) = 0;
        virtual void evaluate(void* tape, void* start, void* end,
            bool useAtomics = true) = 0;
        virtual void reset(void* tape, bool clearAdjoints = true) = 0;
        virtual void reset(void* tape, void* position,
            bool clearAdjoints = true) = 0;
        virtual void pushExternalFunction(void* tape, Handle const* handle) = 0;

        // tape editing
        virtual void erase(void* tape, void* start, void* end) = 0;
        virtual void append(void* dstTape, void* srcTape, void* start,
            void* end) = 0;
    };
```

Listing 2. Interface that guides the implementation of an AD tool wrapper.

In the remaining parts of this section, we discuss some refinements of the AD workflow from Listing 1 for advanced use cases.

**Multiple recordings and evaluations.** The AD tool may support multiple recordings on the same tape and multiple evaluations of one recorded tape, but OpDiLib does not depend on these features. However, it differentiates them correctly if they are present.

**Positional evaluation of the master tape.** If the AD workflow in the serial part of the code involves positional evaluations of the master tape, additional small modifications of the workflow are required in order to get correct results with OpDiLib. As detailed in Section 3, OpDiLib maintains internal states for the differentiation. State changes are tracked alongside the recording on
the classical tapes, and reverted accordingly during evaluation. Hence, if parts of the master tape are evaluated starting from a position that does not match OpDiLib’s current state, the state corresponding to that position must be restored first. To that end, OpDiLib’s logic layer offers functions for the export and recovery of internal states. These have to be used alongside the export of tape positions and prior to the positional evaluation, respectively. The resulting workflow is depicted in Listing 3.

```
AD::Position startPos = AD::masterTape->getPosition();
AD::masterTape->startRecording();
// function to be differentiated, involves parallel code
f(inputs, outputs);
AD::masterTape->endRecording();

AD::Position endPos = AD::masterTape->getPosition();
OpDiLib::State endState = OpDiLib::logic->exportState();
...
// other code
OpDiLib::logic->recoverState(endState);
AD::masterTape->evaluate(endPos, startPos);
```

Listing 3. Modified workflow for positional evaluation of the master tape.

**Tape activity changes within parallel code.** There are some edge cases that—unlike we assumed in Listing 1—require a local interruption of the recording within an otherwise recorded parallel region, for example, external functions that are executed on the AD type instead of a scalar type. This is supported by OpDiLib. In general, the tape activity is used by OpDiLib as an indicator whether or not to handle AD events, which has multiple implications. First, recordings may only be performed inside a parallel region whose entry is recorded because otherwise no parallel recording environment is prepared. Second, the tape activity at any begin event should be identical to the tape activity at the matching end event. Third, any synchronization that takes place during interruptions of the recording is not handled by OpDiLib. Tape activity changes are therefore discouraged. An experienced user, on the other hand, can take advantage of this and eliminate synchronization from the tapes that is not required during the reverse pass. To give an example, the following code showcases a barrier that does not result in a barrier in the reverse pass.

```
AD::threadLocalTape->setPassive();
#pragma omp barrier
AD::threadLocalTape->setActive();
```

**External functions involving parallelism.** While OpDiLib uses external functions to embed state changes and reverse parallelization into classical tapes, they are originally designed as a tool for users to provide custom derivatives for code, say a function \( g \), where blackbox differentiation is inaccurate or costly in terms of memory or runtime [Kowarz 2008]. A typical example for this is the differentiation through a solver for a linear system of equations [Sagebaum et al. 2013]. Instead of the statement-level recording, user-provided code for the differentiation of \( g \) is embedded into the tape. Transitioning to the parallel case, the user-provided derivative code does not only replace the derivative computation but also the reverse parallelism that would otherwise be deduced automatically by OpDiLib. This gives experienced users another opportunity to fine-tune the performance.
OpDiLib is not aware whether the tapes it evaluates contain usual statement-level recordings or external functions. Hence, support for external functions involving parallelism is mostly an issue of the underlying AD tool. Particularly, its external function tools must be thread-safe, which involves declaration of inputs and outputs, tape activity changes, the registrations of the external function on the tapes, and where necessary, barrier synchronization between these steps. From OpDiLib’s point of view, each thread that registered the external function on its tape in the forward pass results in a thread that participates in the evaluation of the custom derivative code in the reverse pass.

3 EVENT-BASED DIFFERENTIATION LOGIC

We present the event-based differentiation logic that is implemented in OpDiLib’s logic layer. Throughout, we identify requirements for the event generation mechanism.

3.1 Handling of OpenMP Constructs

We propose to implement the AD handling of OpenMP as reactions to events that occur in pairs and indicate the beginning and end of OpenMP constructs alongside the OpenMP workflow. This is inspired by the execution model events introduced in the OpenMP 5.0 specification [OpenMP ARB 2018]. However, there is a multitude of execution model events, and a lot of internal information is passed to the callbacks that can be registered for these events. Not all of it is relevant for differentiation. For the purpose of AD, we derive from the execution model events a reduced set of events with correspondingly adapted data, to which we refer as AD events in the following. AD events admit a joint AD handling of similar OpenMP constructs. Furthermore, they do not depend on OMPT and can be used together with other event generation mechanisms, for example, the one presented in Section 4.2. An overview over the event-based framework is provided in Figure 2. For each AD event, the handling consists of a forward action and a reverse action that are accompanied
Fig. 3. Parallel region related AD events in the intuitive ordering with respect to each other and the OpenMP workflow, as well as corresponding AD data flow. Runtime information that the event generation mechanism is required to provide is tagged with (omp).

by appropriate AD data. The forward action contains additional code for the forward pass and the reverse action contains reverse OpenMP logic in the shape of an external function. Upon encountering the event, the forward action is executed immediately and the reverse action is pushed onto the thread’s tape. AD data encompasses any data required for the AD handling in the forward and reverse actions of matching begin and end events. It is usually created in the forward action of a begin event, passed to the matching end event, and provided to the reverse actions of either of both events as external function data. In addition, the AD data is provided to related AD events. Besides generating the AD events, the event generation mechanism should provide them with runtime information as required for the AD handling and link matching begin and end events as well as families of related events by providing them with the correct AD data.

Differentiation of OpenMP constructs within the framework presented in Figure 2 is enabled by the observation that code augmentations for AD are required solely at the beginning and end of OpenMP constructs, together with the respective positions in the reverse pass. In the remaining parts of this section, we show how known operator overloading approaches fit into this framework, how approaches known from source transformation can be transferred to an operator overloading setting, and how these mechanisms can be extended to cover additional OpenMP constructs, up to what is specified in OpenMP 2.5 [OpenMP ARB 2005]. In doing so, we identify also required runtime information.

Parallel regions and implicit tasks. Appropriate AD handling in an operator overloading setting was established by Kowarz [2008] and implemented in ADOL-C [Bischof et al. 2008]. Threads perform concurrent recordings on different tapes in the forward pass, and evaluate them in parallel in the reverse pass. We use this as a detailed example of AD handling within the event-based framework illustrated in Figure 2. The corresponding AD event family with associated AD data is displayed in Figure 3, with the respective forward and reverse actions detailed in Figure 4. The preparation of the parallel recording environment and the parallel tape evaluation are handled by the ParallelBegin and ParallelEnd events, respectively, whereas the management of individual tapes and tape position is the responsibility of ImplicitTaskBegin and ImplicitTaskEnd events. As an advancement to what is reported on by Kowarz [2008], and similar to what is achieved by Kaler et al. [2021], the implementation shown in Figure 4 can be extended to cover also nested parallelism. This is achieved by appropriate tape management. Without nesting, it suffices if the mechanism behind the OpDiLib::acquireTape() and OpDiLib::releaseTape() calls ensures that no tape is used for recording by two threads simultaneously. With nesting, we provide tapes according to the position of the implicit task in the nesting tree. Tapes can be reused in subsequent parallel regions.
Fig. 4. Forward and reverse actions of the parallel region AD event family. Actions without description are either empty (white) or contain management tasks like data allocation, data initialization, and the reverse action external function push (grey). The corresponding AD data is displayed in Figure 3.

Fig. 5. Nested parallel regions (right) with the corresponding nesting tree (left). Non-leaf nodes indicate forks and edges correspond to implicit tasks, numbered relative to the parent task. If workA is imbalanced so that the implicit tasks 0.0 and 0.1 are finished before 1.0 and 1.1 are started, a simple tape management might use the same tapes to record 1.0 and 1.1. If indices are reused, the same index pools would be used for temporaries throughout workB in the left and right half of the nesting tree. As all four nested tasks are likely evaluated in parallel in the reverse pass, this would create data races.

Sync regions. It is known from source transformation that the reverse counterpart of a sync region, like an explicit #pragma omp barrier directive, is also a barrier directive, with no additional treatment in the forward pass [Förster 2014]. We introduce a SyncRegionBegin event and a SyncRegionEnd event and place the reverse barrier in the reverse action of one of these events. This covers all types of barriers, including implicit barriers at the end of OpenMP worksharing directives, where we require the event generation mechanism to produce events only if there is no nowait clause.

Worksharing. Worksharing directives like #pragma omp for, #pragma omp single, and #pragma omp sections define independent units of work that are scheduled for execution by specific threads at runtime. Since each thread is equipped with its own tape already, the specific worksharing is
referred to in the recordings performed on the different tapes; the same worksharing is automatically achieved in the reverse pass simply by evaluating the tapes in parallel. This means that neither #pragma omp section declarations of units of work nor schedule clauses require additional treatment, whereas appropriate schedule reversals have received particular attention in source transformation [Hückelheim and Hascoët 2022]. Additional augmentations for worksharing directives in [Forster 2014] are either specific to source transformation or part of a joint reversal approach, where the memory footprint is reduced by checkpointing techniques as part of the transformation. As is common in operator overloading tools, a user has to employ such techniques manually. WorkBegin and WorkEnd events are reserved for future augmentations of worksharing directives.

Mutexes. To differentiate mutex-type synchronization correctly, we have to reverse the specific execution order produced in the forward pass. This is reflected in a counter-based AD approach developed for critical regions in source transformation [Forster 2014], where entering the critical region corresponds to acquiring the mutex and leaving the critical region corresponds to releasing it. Figure 6 shows how it can be implemented with the help of MutexAcquired and MutexReleased AD events in our framework. We make the observation that this concept can be used to differentiate any synchronization of mutex type and apply it for critical regions, locks, nested locks, ordered clauses, and reduction clauses. Counters are associated with mutexes via the mutex kind and a unique mutex id that have to be provided by the event generation mechanism. The current values of all counters are part of OpDiLib’s state that we mentioned before in Section 2 in connection with Listing 3.

Reductions. Since the OpenMP 4.0 specification [OpenMP ARB 2013], user-defined reductions can be declared, which can be leveraged for reductions involving data types that are used in operator overloading AD tools. However, a declaration of a custom reduction alone is not sufficient to obtain correct reverse mode derivatives. Reductions are performed in OpenMP in two stages [OpenMP ARB 2021]. First, each thread that participates in the reduction accumulates its results in a private variable. At the end of the OpenMP construct with the reduction clause, the threads’ individual results are then accumulated into the shared reduction target, usually involving several threads and shared intermediate variables, for example, in a tree-like reduction. From an AD point of view, the operations in the course of the reduction are recorded just as any other operations on the tapes of the threads that execute them. However, access to shared intermediate variables and
the final reduction target requires synchronization, which must be turned into reverse synchronization again. We require that a thread receives a `MutexAcquired` event before contributing to a shared variable, and a `MutexReleased` event afterward; just as if access to the shared variable was safeguarded by a mutex. To ensure that no thread begins with the reversal of the structured block of the OpenMP construct before the reversal of the shared reduction stage has progressed to a point where all contributions have been propagated to the private reduction variable of that thread, we require that additional `SyncRegionBegin` and `SyncRegionEnd` events are emitted between the two reduction phases. This is important for threads that did not participate in the shared reduction phase and did not receive mutex AD events.

Further clauses and directives that do not require treatment. The `#pragma omp master` directive does not involve any synchronization; it only declares that the code will be executed by a specific thread, and hence, recorded on a specific tape. Thus, it will also be evaluated by only one thread. This is in line with the results on source transformation from Förster [2014]. OpenMP allows combined constructs, usually a parallel directive combined with a worksharing construct, for example, `#pragma omp parallel for`. Such combined constructs are supported automatically by the logic presented above as long as all involved mechanisms generate their corresponding AD events. Furthermore, we do not have to add special treatment for data-sharing attribute clauses such as `private` or `firstprivate` or data copying clauses such as `copyprivate`. The resulting copy operations are recorded in the same fashion as any other operations. Source transformation tools, on the other hand, have to implement treatment for such clauses [Förster 2014; Hückelheim and Hascoët 2022; Hückelheim et al. 2019].

Atomics. The OpenMP specification allows `#pragma omp atomic` directives only for scalar types, which excludes the custom types that are used for active variables in an operator overloading code. Many of the remaining use cases for atomics, such as computations on non-AD floating point types or increments of counters, include synchronization that does not impact on the correctness of the derivative. Therefore, we do not add AD handling of atomic directives. In theory, synchronization due to atomics could also be handled by mutex AD events; in fact, atomics generate `mutex-acquired` and `mutex-release` execution model events. In practice, patterns like busy-waits would quickly blow up the mutex counters and are more efficiently handled by custom mutex events that reflect the synchronization mechanism. If a pattern such as the busy-wait synchronization in Figure 6 was used in the forward pass, one `MutexAcquired` event could be emitted upon a completed busy-wait, with the corresponding `MutexReleased` event placed after decrementing the counter. Note that missing AD handling of atomics does not prevent us from using them in the reverse pass for atomic updates on adjoint variables or for synchronization as displayed in Figure 6.

Flush directive. The `#pragma omp flush` directive enforces a consistent view of the threads on the memory, either in general or with respect to specific variables. As can be seen, e.g., in the examples in the OpenMP 2.5 specification [OpenMP ARB 2005], it can be used to implement quite irregular synchronization patterns. We see no way to handle the flush directive without implicitly turning it into a stronger kind of synchronization like atomics, mutexes, or barriers, effectively eliminating its advantages. Regarding the reverse pass, no issues with inconsistent views of memory should occur if atomic updates are used on adjoint variables. Otherwise, we offer a reverse flush that the user may push to the tapes manually, in the same spirit as the reverse barrier described in Section 3.2.

Higher order derivatives. In the adjoints of tangents approach, the forward mode of AD is used for all derivative orders except the outermost one. This strategy is not only favored for its efficiency properties [Griewank and Walther 2008] but also supported by the differentiation logic presented above. The adjoints of adjoints approach, on the other hand, applies the reverse mode to the reverse
mode, which involves recording of an evaluation that—in a parallel context—likely involves atomic updates on adjoint variables. This conflicts with missing AD handling of atomics. It also gives rise to further implementational issues in the event generation mechanism that are discussed in Section 4. Therefore, adjoints of adjoints are not supported by OpDiLib.

3.2 Shared Reading and Atomic Adjoints

OpDiLib itself cannot automatically identify whether or not atomic updates of adjoint variables are required. In particular, unlike source transformation tools [Hückelheim and Hascoët 2022], it cannot distinguish between private and shared variables specified on OpenMP constructs via the corresponding data-sharing attribute clauses. The reason is that there are no execution model events associated with these clauses, and hence, there is no solution for this issue in the scope of this work. In order to make OpDiLib as applicable as possible, we do not want to impose any restrictions on data access patterns and we do not want to require users to restructure their codes to get a first working version. Therefore, we default to atomic updates on adjoint variables. On the other hand, a user should have as much flexibility as possible to disable atomic adjoints manually where they are not needed. To that end, OpDiLib tracks an adjoint access mode per thread alongside the recording as part of the parallel region’s associated AD data, which the event generation mechanism is required to make available. The adjoint access options are Atomic, which is the default for OpDiLib, or Classical, which is the non-atomic counterpart. The user may control the adjoint access mode by calls to the setAdjointAccessMode routine of the logic layer at arbitrary points in the code. The recording that is performed after the call until the next change of the adjoint access mode in that thread is then evaluated in the reverse pass according to the chosen mode. This can be used to disable atomics globally, for example, if the reversal strategy from Kaler et al. [2021] is implemented in the underlying AD tool. It can also be applied to mark individual parallel regions as safe for evaluation without atomics, for example, if the user decides to make thread-private copies of all inputs to the parallel region [Bischof et al. 2008; Kowarz 2008]; or if a parallel loop has self-adjoint memory access [Hückelheim et al. 2019]. In addition, one may isolate any parts of large parallel regions as safe for evaluation without atomics in a fine-grained manner, even across the boundaries of parallel regions. The following example demonstrates that additional synchronization might be required during the reverse pass.

```cpp
ADType shared_data[2] = ...;
ADType result[2] = ...;
#pragma omp parallel num_threads(2)
{
  int i = omp_get_thread_num();
  OpDiLib::logic->setAdjointAccessMode(OpDiLib::Classical);

  // part without shared reading
  result[i] *= shared_data[i];

  OpDiLib::logic->addReverseBarrier();
  OpDiLib::logic->setAdjointAccessMode(OpDiLib::Atomic);

  // part with shared reading
  result[i] += shared_data[0] * shared_data[1];
}
```

The change of the adjoint access mode in line 12 separates adjacent parts of the code with and without shared reading. During the forward pass, shared reading is not a problem, and no
synchronization is required. During the reverse pass, however, one thread might finish first with the atomic evaluation of the recording for line 15 and begin with unsafe updates on the adjoint of \texttt{shared_data[i]} while the other thread is still busy with the safe evaluation of its recording for line 15, resulting in simultaneous safe and unsafe writes to the same variable. This is prevented by introducing a barrier exclusively in the reverse pass by the additional call in line 11. It pushes an external function containing a \texttt{#pragma omp barrier} statement to the tape, just as in the handling of the \texttt{SyncRegion} events in Section 3.1.

### 4 EVENT GENERATION MECHANISMS

The AD events introduced in Section 3.1 have to be generated alongside the OpenMP workflow. In principle, the required augmentation of the source code by AD events could be performed by hand. The objective of OpDiLib’s backend layer is to automatize this tedious task, up to the point where no changes to the source code are required. This is achieved with the OMPT backend that is presented in Section 4.1. It makes use of modern OpenMP features around OMPT that allow for an observation and augmentation of the OpenMP workflow. This way, we solve the key challenge of augmenting \texttt{#pragma} directives without manual annotations of the source code.

As a fallback for compilers without OMPT support, we provide a second backend that hides the augmentations of the source code inside replacement macros for OpenMP \texttt{#pragma} directives and clauses as well as replacements for OpenMP’s runtime functions and additional end macros. They form an alternative interface to OpenMP functionality which we present in Section 4.2. It provides access to the same set of AD features as the OMPT backend. Clearly, the macro backend requires the rewriting of all OpenMP directives and runtime function calls. In principle, given the structural similarities of the original constructs and their replacements, this task could be performed by a source transformation preprocessing script in an automatic fashion. The code that is written with the replacements is also fully compatible with the OMPT backend. This means that no additional rewriting is required for a transition to the modern features. As long as the OMPT backend is used, original OpenMP constructs and their respective replacements can be used in the same code.

Full code examples that showcase the use of the two backends can be found in the OpDiLib repository.\(^5\)

#### 4.1 OMPT Backend

The OpenMP specification [OpenMP ARB 2021] formalizes the OpenMP workflow by defining execution model events. For each event, a custom callback function can be registered that is invoked by the OpenMP runtime at each occurrence of the corresponding event in the respective thread. The callback function receives detailed information about the OpenMP mechanisms and objects involved. In many cases, it also offers possibilities to associate custom data with events. This allows developers to build their own tools on top of OpenMP and integrate them seamlessly into the workflow of the OpenMP runtime. We use these features for the generation of AD events without any modifications of the OpenMP parallel source code. Since the AD events introduced in Section 3.1 are derived from the execution model events, a key part of a callback implementation is always the transformation to the corresponding AD event. Additionally, OMPT is powerful enough to fulfill further requirements identified throughout Section 3, for example, AD data transport, mutex identifier generation, or general access to parallel AD data.

We illustrate how a backend can be implemented with functionality provided by OMPT using the example of a parallel region and its implicit tasks. The OpenMP specification [OpenMP ARB 2021] defines the following order of events for a parallel region.

\(^5\)https://github.com/SciCompKL/OpDiLib.
— A thread that encounters a parallel construct receives a parallel-begin event before any implicit task associated with the parallel region is created.
— Guided by the requested degree of parallelism, implicit tasks are generated, all of which execute the structured block of the parallel construct. Each implicit task is executed by a dedicated thread. The thread that encountered the parallel region executes the implicit task with index zero.
— Each thread that participates in the parallel region receives an implicit-task-begin event before it starts executing the structured block. After it has finished executing the structured block and has received events associated with the implicit barrier at the end of the parallel region, it receives an implicit-task-end event.
— After the thread that encountered the parallel region has received its implicit-task-end event, it receives a parallel-end event. After that, it resumes execution of the task in which it encountered the parallel region.

This order is inherited by the corresponding AD events. We remark that no ordering is specified for the parallel-end event and implicit-task-end events of implicit tasks other than the one with index zero. The extreme case is that they occur no sooner than immediately prior to the next implicit-task-begin in that thread, in particular after the next parallel-begin event. We observed this in the LLVM OpenMP runtime. The implementation of the differentiation logic has to be correct also for these extreme cases. Listing 4 displays implementations of the callbacks for parallel-begin and parallel-end events that match the ompt_callback_parallel_begin_t and ompt_callback_parallel_end_t signatures as defined by the OpenMP specification [OpenMP ARB 2021].

```c
void on_parallel_begin(void* ptr, ompt_data_t* parallelData, unsigned int requested, int, void const*) {
    parallelData->ptr = OpDLib::logic->onParallelBegin(requested);
}

void on_parallel_end(void* ptr, ompt_data_t* parallelData, ompt_data_t*, int, void const*) {
    OpDLib::logic->onParallelEnd(parallelData->ptr);
}
```

Listing 4. Callbacks for parallel-begin and parallel-end events.

We omitted the names of arguments that we do not use. The callback for the parallel-begin event receives information about the requested parallelism, that is, the number of threads that was specified with the num_threads clause if present, or the default number of threads instead, as required in Section 3.1. Furthermore, the OpenMP runtime allocates a parallel data object of structured type ompt_data_t that is associated with the parallel region. It has a member void* ptr that can be used to add custom data. We use it to store the parallel AD data object returned by the logic layer. The callback for the corresponding parallel-end event receives the same OMPT data object with the ptr member as it was assigned in the parallel-begin callback. Hence, we can use it to transport the AD data between the matching begin and end events as displayed in Figures 1 and 2 and required in Section 3.1. OMPT’s data object associated with the current parallel region can also be queried at arbitrary points in the code via the ompt_get_parallel_info runtime entry point. Thus, we implement access to the parallel AD data object of the current parallel region as is needed in Section 3.2 by querying OMPT’s parallel data object, and extracting the parallel AD data object from it in the same fashion as in Listing 4. Listing 5 displays the implementation of the callback for implicit-task-begin and implicit-task-end events. It has the ompt_callback_implicit_task_t signature as required by the OpenMP specification [OpenMP ARB 2021].
### Table 1. Mapping between OMPT Callbacks and AD Events

| OMPT callback type                       | AD events                          |
|------------------------------------------|------------------------------------|
| ompt_callback_parallel_begin            | ParallelBegin                      |
| ompt_callback_parallel_end               | ParallelEnd                        |
| ompt_callback_implicit_task              | ImplicitTaskBegin, ImplicitTaskEnd |
| ompt_callback_mutex_acquired             | MutexAcquired                      |
| ompt_callback_mutex_released             | MutexReleased                      |
| ompt_callback_sync_region                | SyncRegionBegin, SyncRegionEnd      |
| ompt_callback_work                       | WorkBegin, WorkEnd                 |
| ompt_callback_reduction                  | MutexAcquired, MutexReleased        |

```c
1 void on_implicit_task(ompt_scope_endpoint_t endpoint, ompt_data_t* parallelData,
2     ompt_data_t* taskData, unsigned int actual, unsigned int index, int flags) {
3
4     if (flags & ompt_task_initial) {
5         return;
6     }
7
8     if (ompt_scope_begin == endpoint) {
9         taskData->ptr = OpDiLib::logic->onImplicitTaskBegin(
10             actual, index, parallelData->ptr);
11     }
12     else {
13         OpDiLib::logic->onImplicitTaskEnd(taskData->ptr);
14     }
15 }
```

Listing 5. Callback for *implicit-task-begin* and *implicit-task-end* events.

It filters out the special case of the initial implicit task, that is, the task that executes the serial parts of the program. It does not require additional AD treatment. The callback receives all parameters from the OpenMP runtime that are needed for the implicit task AD events, that is, the actual degree of parallelism (as opposed to the requested one), the index in the team of threads, and the parallel AD data object. The task data object provided by OMPT is again used to exchange AD data with the corresponding end event.

Table 1 provides an overview overall callbacks that we use to generate the AD events with OpDiLib’s OMPT backend. Not each execution model event has a dedicated callback function. In contrary, a single callback often handles a large number of events. For example, the *critical-acquired* event due to a `#pragma omp critical` directive and the *lock-acquired* event caused by the runtime function `omp_set_lock` both result in invocations of the `ompt_callback_mutex_acquired` callback. Likewise, an `ompt_callback_sync_region` callback handles (amongst others) explicit barrier directives and implicit barriers at the end of, e.g., worksharing directives that do not specify a *nowait* clause. It is beneficial for AD that similar types of events are collected in one callback for two reasons. First, this often coincides with the fact that all execution model events collected by the same callback also require the same treatment from an AD point of view and can be forwarded to the same module of the logic layer. Second, we do not have to distinguish whether the execution model event is caused by a `#pragma` directive, the presence or absence of clauses, a runtime function call, or other mechanisms. This allows us to conveniently establish AD support for a large range of OpenMP features.

In the view of further requirements identified in Section 3.1, the OpenMP runtime passes unique mutex identifiers to mutex callbacks, which we can immediately reuse to identify and track
mutexes in the logic layer. Pending mutex-released events in one thread might occur only after mutex-acquired events for the same mutex in another thread. Unlike the handling of the mutex-acquired event, the handling of the mutex-released event is not protected by the corresponding mutex, which is regarded in the implementation of the differentiation logic. The reduction callback is special among the cases in Table 1 in the sense that it produces mutex AD events. According to the OpenMP specification [OpenMP ARB 2021], the reduction-begin event is received before the task begins to perform load and stores on shared reduction variables, and the reduction-end event is received after these are completed. We use these to generate the mutex AD events required in Section 3.1. Thus we record the reduction operations as if they were performed in an equivalent serial manner, that is, we choose a total order for all reduction events that preserves the partial order indicated by the reduction tree. Reversal is then performed according to this total order. Conveniently, the LLVM OpenMP runtime produces sync region events for a so-called implementation specific barrier that separates the execution of the structured block and the accumulation in private reduction variables from the part of the reduction that actually involves synchronization. These events are detected and handled by OpDiLib like any other sync region. Hence, when using this OpenMP runtime, it is ensured that no thread begins with the reversal of the structured block before all reduction operations are reverted. This is important for threads that did not participate in the reduction and did not receive mutex events. This implementation specific barrier is not guaranteed by the OpenMP specification. Besides its usefulness for AD, it provides insights in the reduction process that should also be of interest to performance monitoring tools. Therefore, we think that it is a desirable feature in general and would like to see it in future versions of the OpenMP specification. If OpDiLib is used together with other OMPT implementations, it might be necessary to add a reverse barrier by hand, or to use semi-automatic techniques similar to those in the macro backend. We close this section with a discussion of additional issues due to the use of OMPT.

Execution model events due to internal synchronization. OpDiLib uses OpenMP’s lock facilities for internal synchronization purposes. From the OpenMP runtime’s point of view, these are not any different from locks in user code and hence produce the same execution model events and cause invocations of the same callbacks. Besides the unnecessary performance cost of differentiating the logic layer’s internals, there is also a more imminent issue. If the user successfully sets a lock, an ompt_callback_mutex_acquired callback is dispatched and generates a MutexAcquired AD event. However, the handling of this event itself requires the acquisition of an internal lock that invokes the same callback, resulting in an infinite recursion. To fix this, the logic layer keeps track of inactive mutexes. These are then filtered out in the AD event handling. An experienced user can also use these mechanisms to mark mutexes from user code as inactive by a call OpDiLib::logic->registerInactiveMutex(mutexKind, mutexId). These are then disregarded for the differentiation. This is also recommended for mutexes introduced in the underlying AD tool for the purpose of thread-safety.

Execution model events during reversal. Since OpDiLib embeds OpenMP directives into the tape, execution model events are also produced during the reverse pass. However, the tapes are not in recording mode at this point. This is detected in the logic layer and the AD handling is skipped. We remark—despite the limited practical relevance—that execution model events during reversal might have an application for higher order derivatives that are computed in an adjoints of adjoints fashion, see the previous discussion at the end of Section 3.1. In an application of OpDiLib to itself, the execution model events during an evaluation for the higher order type could be used to perform the recording for the lower order type. This relates also to the next topic of discussion.
Multiple OMPT tools. Suppose we want to apply OpDiLib to an OpenMP parallelized code that uses an OMPT tool already, for example, for performance monitoring purposes. The simultaneous use of multiple OMPT tools is beyond the scope of the OpenMP specification [OpenMP ARB 2021]. To overcome this limitation, there was work in the direction of multiplexing of OMPT tools [Protze et al. 2019] that is now part of clang. We will conduct further tests to see whether it can be used for our purposes. Otherwise, the macro backend may serve as a fallback solution.

4.2 Macro Backend

Without OMPT, we are not aware of any way to change the behavior of the original OpenMP constructs in the scope of the program. Instead, we provide an alternative interface to OpenMP functionality. Code that is written according to this interface can then also be differentiated if a compiler without OMPT support is used. Specifically, we introduce macros that have to be used instead of the usual OpenMP directives and clauses, as well as functions that have to be used instead of the usual OpenMP runtime functions. This provides a semi-automatic way of differentiating OpenMP parallel code with operator overloading. The replacements always contain the original OpenMP construct but embed it into additional code that generates the required AD events, and the replacements follow mostly the same structure as the original OpenMP constructs.

Runtime functions. For each runtime function omp_* , we introduce a replacement opdi_* . For example, omp_set_lock is replaced by opdi_set_lock. The replacement has the same signature as the original function. Its implementation always forwards the call to the original runtime function, but surrounds it with an appropriate generation of AD events, as showcased in Listing 6.

Listing 6. omp_set_lock runtime function replacement.

```
void opdi_set_lock(omp_lock_t* lock) {
  omp_set_lock(lock);
  OpDiLib::logic->onMutexAcquired(OpDiLib::Lock,
    OpDiLib::backend->getLockIdentifier(lock));
}
```

As can be seen in Listing 6, the generation of mutex identifiers, which was previously performed by the OMPT runtime, now requires a dedicated custom implementation in the backend layer. For reasons of consistency, we offer replacements also for runtime functions that do not require AD treatment. A user code that should compile both with and without OpDiLib can then use a global macro to decide whether the omp_ or opdi_ prefix is used for the runtime function calls, without keeping track of the necessity of this switch for each individual runtime function.

Directives. We replace each OpenMP directive #pragma omp directive ... by a custom macro OPDI_DIRECTIVE(...). The idea is that the same clauses and hints that were passed to the original directive can also be passed to the macro, possibly again in the shape of their replacements. The macro expands to the original OpenMP directive surrounded by corresponding AD treatment. This is showcased for the barrier directive in Listing 7.

Listing 7. Barrier replacement macro.

```
#define OPDI_BARRIER(...) \
  OpDiLib::logic->onSyncRegionBegin(OpDiLib::Barrier); \
  _Pragma(omp barrier __VA_ARGS__) \
  OpDiLib::logic->onSyncRegionEnd(OpDiLib::Barrier);
```

The situation is more involved for OpenMP directives that are followed by a structured block, for example, #pragma omp for. This has two reasons. First, it might be necessary to generate AD events...
within the structured block. Second, it might be necessary to spawn AD events after the structured block. We make heavy use of a privatization technique that was already applied to address these issues for the handling of parallel regions in ADOL-C [Bischof et al. 2008; Kowarz 2008]. Augmentations at the beginning and end of the structured block can be generated as side effects of constructors and destructors of objects that are privatized via data-sharing clauses. These privatizations can be generated as part of both replacement macros and replacement clauses. However, this technique is not applicable or practical in all situations. First, it covers only OpenMP directives that support data-sharing clauses. Furthermore, the OpenMP specification leaves the respective order of constructor and destructor calls unspecified, does not support multiple privatizations of the same variable on one directive, and restricts subsequent privatization in nested directives to some extent [OpenMP ARB 2021]. Privatization is the only approach we are aware of to design replacement clauses with side effects on the recording environment. Thus, their limitations give rise to some special cases which we discuss below. To find a remedy for directives, we complement each `OPDI_DIRECTIVE(...)` replacement that is followed by a structured block by a corresponding `OPDI_END_DIRECTIVE` macro that has to be placed after the structured block. This is showcased in Listing 8 and resembles the Fortran way of using OpenMP directives rather than the C++ one.

### Table 2. Replacement Macros and Corresponding End Macros

| OpenMP directive | replacement macro | end macro |
|------------------|-------------------|-----------|
| #pragma omp barrier ... | OPDI_BARRIER(...) | OPDI_END_BARRIER |
| #pragma omp parallel ... | OPDI_PARALLEL(...) | OPDI_END_PARALLEL |
| #pragma omp for ... | OPDI_FOR(...) | OPDI_END_FOR |
| #pragma omp sections ... | OPDI_SECTIONS(...) | OPDI_END_SECTIONS |
| #pragma omp single ... | OPDI_SINGLE(...) | OPDI_END_SINGLE |
| #pragma omp single nowait ... | OPDI_SINGLE_NOWAIT(...) | OPDI_END_SINGLE |
| #pragma omp critical ... | OPDI_CRITICAL(...) | OPDI_END_CRITICAL |
| #pragma omp critical (name) ... | OPDI_CRITICAL_NAME(name, ...) | OPDI_END_CRITICAL |
| #pragma omp ordered ... | OPDI_ORDERED(...) | OPDI_END_ORDERED |
| #pragma omp master ... | OPDI_MASTER(...) | OPDI_END_MASTER |
| #pragma omp section ... | OPDI_SECTION(...) | OPDI_END_SECTION |

Listing 8. Example code on the left and equivalent code with replacement and end macros on the right.

```
1 #pragma omp parallel num_threads(4)
2 {
3   #pragma omp for schedule(guided, 1)
4     for (int i = 0; i < N; ++i) {
5       ...
6     }
7   #pragma omp master
8 }
9 #pragma omp section
```

```
1 OPDI_PARALLEL(num_threads(4))
2 {
3   OPDI_FOR(schedule(guided, 1))
4   for (int i = 0; i < N; ++i) {
5     ...
6   }
7   OPDI_END_MASTER
8 }
9 OPDI_END_PARALLEL
```

End macros also provide important degrees of flexibility in our implementation with respect to future adaptions. Table 2 provides an overview overall replacement directives and the corresponding end macros. Among the replacement directives there are two special cases. The first are named critical regions. The `critical` directive does not support data-sharing clauses. However, the supplied name is essential for the generation of a unique mutex identifier so that is has to be provided via a dedicated macro. Second, while the `single` directive supports data-sharing clauses, these apply only to the single thread that executes the structured block, which is not sufficient for
the treatment of the implicit barrier that regards all threads. Hence, a dedicated macro for single directives with a \texttt{nowait} clause is introduced. Note that the special replacements still use the same respective end macro.

\textit{Clauses.} Most clauses and hints do not require replacements and are passed to the replacement directives in their original form. Besides the special replacement directives discussed above, we provide an \texttt{OPDI\_NOWAIT} replacement clause that has to be used instead of \texttt{nowait} for correct treatment of the corresponding implicit barrier. Furthermore, any directive that specifies \texttt{reduction} clauses has to specify the \texttt{OPDI\_REDUCTION} macro once alongside the otherwise unmodified reduction clauses. It provides the reverse barrier that separates the threadprivate reduction phase from the shared, tree-like reduction phase. We remark that this macro has to be provided as a supplement instead of a replacement because multiple reduction clauses on one directive are allowed but multiple privatizations of the same variable on one directive are forbidden [OpenMP ARB 2021].

\textit{Reductions.} While the \texttt{OPDI\_REDUCTION} macro handles the required reduction barrier, it does not handle the mapping of reductions to mutex AD events. In the macro backend, we associate reductions with actual nested locks so that the required mutex AD events are produced by setting and unsetting these locks. This logic is hidden inside a replacement macro for the declaration of custom reductions. Since any reduction involving AD types is of custom type, this covers all the relevant cases. The replacement macro reads \texttt{OPDI\_DECLARE\_REDUCTION(OP\_NAME, TYPE, OP, INIT)} and expands to code as displayed in Listing 9.

\begin{verbatim}
1 #pragma omp declare \
2   reduction(OP\_NAME : TYPE : \ 
3   OpDiLib::Reducer(omp\_out) \ 
4   = OpDiLib::Reducer(omp\_out) OP OpDiLib::Reducer(omp\_in)) \ 
5   initializer(omp\_priv = INIT)
\end{verbatim}

Listing 9. Schematic reduction declaration.

It involves casts of \texttt{omp\_in} and \texttt{omp\_out} to a \texttt{Reducer} object that is associated with a global nested lock. Note that a nested lock has the distinguished property that it can be set multiple times from within the same thread without causing deadlocks. We implement the \texttt{Reducer} such that the nested lock is set in constructor calls, which happens three times in the course of one reduction statement. The three set operations are complemented by three unset operations as part of the overload of \texttt{operator =} for the \texttt{Reducer}. This ensures that the whole reduction statement is exclusively performed by one thread at a time. This produces a serialization of the reduction tree in the forward pass that is preserved by the mutex AD handling for the reverse pass. In addition, we ensure via templating techniques that different custom reductions use different \texttt{Reducer} objects and hence, different nested locks.

5 PERFORMANCE STUDY

To study the performance of OpDiLib, we apply it to a PDE solver for the coupled Burgers’ equations

\[
\begin{align*}
  u_t + uu_x + vu_y &= \frac{1}{R}(u_{xx} + u_{yy}), \\
  v_t + uv_x + vv_y &= \frac{1}{R}(v_{xx} + v_{yy}).
\end{align*}
\]

[Bahadır 2003; Biazar and Aminikhah 2009; Zhu et al. 2010] that was previously used for the performance evaluation of various aspects of automatic differentiation by Sagebaum et al. [2018, 2019,
We solve Equation (3) on \((x, y, t) \in D \times \mathbb{R}_{\geq 0}\) where \(D\) is a scaling of the unit square \([0, 1]^2\), together with boundary conditions and initial conditions at \(t = 0\) which are taken from the exact solution

\[
\begin{align*}
    u(x, y, t) &= \frac{x + y - 2tx}{1 - 2t^2}, \\
    v(x, y, t) &= \frac{x - y - 2yt}{1 - 2t^2},
\end{align*}
\]

that is given by Biazar and Aminikhah [2009]. We solve the equations with an upwind finite difference scheme, for which we discretize \(D\) by an equidistant grid and advance in time steps of fixed size \(\Delta t\). The simulations in this article were carried out with \(R = 1, D = [0, 50]^2\), on a grid with 2,000 \(\times\) 2,000 cells and for 20 time steps of width \(\Delta t = 0.0001\). We use the reverse mode of AD to differentiate the \(L^2\) norm of the solution after the final time step with respect to the initial data. CoDiPack is used as the underlying AD tool, configured for Jacobian taping [Hogan 2014] with a reuse index handler [Sagebaum et al. 2020]. For the differentiation of OpenMP, we couple it with OpDiLib as explained in Section 2. CoDiPack is also coupled with MeDiPack for the differentiation of MPI. We use hwloc\(^6\) for appropriate binding of MPI processes, and tie OpenMP threads to cores using OpenMP environment variables [OpenMP ARB 2021]. Dynamic frequency scaling is disabled to improve the consistency of the performance results. We display results obtained with the compilers clang++ 10.0.1\(^7\) and g++ 10.1\(^8\). The latter does not support OMPT yet and is hence an important use case for the macro backend presented in Section 4.2.

We begin with a detailed performance study on a dual-socket node of the Elwetritsch cluster at TU Kaiserslautern with two AMD EPYC 7351 processors at 2.4 GHz. Each processor consists of four NUMA domains with four cores each, giving a total of 32 cores per node. We use distributed memory parallelization between NUMA domains and employ shared memory parallelization within them. Solution fields are allocated per MPI process. Since adjoint variables are also allocated per the MPI process, allocation pattern and memory granularity with respect to NUMA domains are identical in the forward and reverse pass. Figure 7 visualizes the corresponding hybrid parallel splitting strategy. The spatial domain is first decomposed into columns of equal width, one for each MPI process. Each MPI process stores its part of the solution field in row-major order, with padding so that the beginnings of rows are aligned with cache lines. Halo exchange via MPI communication is required at the column boundaries. Second, each MPI process works on its column with multiple OpenMP threads. To that end, the column is horizontally subdivided into blocks of equal height, resulting in one block for each OpenMP thread on which it performs row-wise sweeps with the five-point finite difference stencil. As can be seen in Figure 7, shared reading might occur at the boundary between adjacent blocks assigned to different OpenMP threads. The issue of atomic updates discussed in Section 3.2 is reflected in the performance tests by three different adjoint configurations. While atoms are only used in the reverse pass, the respective code annotations and data access pattern adaptions also affect the forward pass.

Atomic adjoints All updates on all adjoint variables are performed atomically. There are no restrictions on data access patterns in the forward pass. No code modifications are needed.

Adjoint access control The code is annotated with the adjoint access control tools presented in Section 3.2. We default to the Classical adjoint access mode. Whenever we process the two rows adjacent to the neighboring block, we switch to the Atomic adjoint access mode.

\(^6\)https://www.open-mpi.org/projects/hwloc/.
\(^7\)https://releases.llvm.org/.
\(^8\)https://gcc.gnu.org/gcc-10/.
Fig. 7. Domain decomposition strategy with one MPI process per NUMA domain (right). We refer to this as NUMA splitting. Memory is allocated per MPI process. The detailed view of the boundaries between MPI processes and OpenMP threads (left) also indicates non-exclusive read access at the OpenMP block boundaries as well as close and distant locations in the row-major memory layout.

Classical adjoints. No atomics are used in the reverse pass. To guarantee exclusive read access in the forward pass, we double the number of blocks per MPI column and process them with the OpenMP threads in two sweeps in an interleaved manner. A reverse barrier is required between the two sweeps.

We start with a comparison of absolute performance values of the different adjoint configurations within one NUMA domain of the AMD EPYC 7351 processor, in which we execute the simulation with either one or four OpenMP threads. We also compare the performance to primal, undifferentiated and classical, serially differentiated versions of the simulation. The results are displayed in Figure 8. The different configurations are denoted on the vertical axis whereas the timings obtained with different compilers and AD setups are denoted on the horizontal axis. We begin with some general observations. If we compare for each configuration the impact of the different compilers (squares versus circles) and the impact of the backend choice (blue versus green), we see that neither of both has a significant influence on the performance. Furthermore, the adjoint configurations do not have an impact on the performance of the forward pass. The OpDiLib results obtained without OpenMP still make use of the thread-safe version of CoDiPack and hence, allow us to evaluate the impact of the CoDiPack modifications on the performance of the standard CoDiPack type (black versus blue). We see that it is mostly preserved, especially in the reverse pass. Using one OpenMP thread, we see a certain performance overhead in the forward pass that is not visible in the primal performance values. The major part of this is due to using a thread-safe CoDiPack type, with minor parts due to using OpenMP and OpDiLib. In the reverse pass, there are significant differences between the three adjoint configurations. While the adjoint access control approach and the classical adjoints strategy fully reproduce the performance of the classical, serially differentiated runs, using atomic adjoints incurs a significant performance overhead. With four OpenMP threads, the speedup of the AD forward pass is similar to the speedup of the primal, undifferentiated simulation, even slightly larger, regardless of the adjoint configuration. In terms of AD, this is also an improvement of more than a factor of two compared to the classical, serially differentiated versions. A parallel reverse pass with atomic adjoints is not only notably slower on average than the classical, serial reverse run but also exhibits huge variation in the timings, resulting in the most unreliable performance among all adjoint configurations. With the adjoint...
Fig. 8. Forward pass (a) and reverse pass (b) performance comparison of different parallel configurations within one NUMA domain of the AMD EPYC 7351 processor, consisting of 4 cores. Includes setups with classical AD, with the macro backend and with the OMPT backend, compiled with g++ □ and clang++ ○. In (a), we also display primal performance values. In the primal versions, atomic adjoints, and adjoint access control configurations coincide. Numbers denote averages of 20 runs after five discarded warm-up runs. Arrows indicate average cost and speedup relations.
access control approach, the parallel reverse pass overhead is reduced significantly. This shows that if there are only a few variables without exclusive read access, the adjoint access control tools provide a way to achieve acceptable parallel reverse performance without revising the data access patterns. The most reliable reverse pass speedup that is consistently at par with the forward pass speedup is obtained with the classical adjoints strategy, which does not use atomic updates at all. Among the atomic adjoints and adjoint access control configurations, the displayed test results seem to suggest differences between compilers and OpDiLib backends both with respect to the extent of noise and the average performance, but this cannot be reproduced in a reliable manner and admits no conclusion. We remark that there is almost no difference between the memory high water marks of the tested AD setups. Compared to the classical, serially differentiated simulation, any additional memory consumption due to OpDiLib’s differentiation logic and backend is negligible.

Next, we investigate the strong scaling of the adjoint configurations. In the forward pass, we also include the scaling of the primal, undifferentiated code. The scaling strategy is implied by the NUMA splitting displayed in Figure 7 together with the hardware characteristics of the AMD EPYC 7351 processor. For each configuration, the timings obtained with one MPI process and one OpenMP thread serve as the baseline. We first increase the number of OpenMP threads per MPI process to four, and then increase the number of MPI processes until the full node is used. The results are displayed in Figure 9. We see that all AD setups scale better than the primal, undifferentiated simulation. This is often observed in AD and is attributed to the fact that AD improves the computation to memory ratio in the forward pass [Nemili et al. 2016]. Neither the OpDiLib backend choice nor the compiler nor the adjoint configuration have a significant influence on the scaling in the forward pass. In the reverse pass, the OpDiLib backend choice does not have a significant impact either. clang++ achieves better scaling in the reverse pass, and less atomics correlate with better scaling. Overall, the reverse scaling results are very good, some configurations exhibit even superlinear behavior, which is fostered by multiple factors. In the splitting strategy displayed in Figure 7, shared reading—and hence serialization in the reverse pass—can only occur at the cells at block boundaries between adjacent OpenMP threads. The number of these cells is furthermore
Fig. 10. Domain decomposition strategy with one MPI process per socket (right). We refer to this as *socket splitting*. As indicated in the detailed view of the boundaries between MPI processes and OpenMP threads (left), OpenMP communication uses halo exchange, with which exclusive read access is guaranteed.

invariant with respect to the number of MPI processes so that, in the worst case, there is a constant small fraction of the computational tasks that does not scale. Furthermore, the splitting strategy itself supports scaling. Increasing the number of MPI processes corresponds to decreasing the width of the column for each process. This, in turn, implies that the distant memory locations that are accessed during one application of the finite difference stencil move closer to each other, see Figure 7. This applies analogously to the corresponding pattern of adjoint variable access in the reverse pass. Hence, memory locality improves with an increasing number of MPI processes.

We complement the performance results obtained so far by scaling tests with a more OpenMP centered approach. To that end, we use one MPI process per socket with up to 16 OpenMP threads on the AMD EPYC 7351 processor. To better distribute the memory load across the NUMA domains, we move from memory allocations per MPI process to memory allocations per thread. This results in a new splitting strategy that is visualized in Figure 10. Neighboring threads no longer access each other’s memory directly. Instead, we use halo exchange analogous to MPI communication, but implemented with OpenMP. This involves communication buffers and suitable synchronization; as a side effect, exclusive read access is guaranteed in the forward pass for all variables, which in turn impacts the adjoint configurations. Even though no atomic updates are needed, we can use them nonetheless to investigate their impact on the performance. Adjoint access control, however, is not meaningful in the scope of this splitting strategy. The results are displayed in Figure 11. They demonstrate that very good scaling can be achieved with OpDiLib and also with larger numbers of OpenMP threads per MPI process.

The reverse pass in tape-based AD is memory-bound, also if OpDiLib is used. We can make this visible on a different type of testing hardware, a dual-socket node of the Elwetritsch cluster with two Intel Xeon Gold 6126 processors at 2.6 GHz. Each processor consists of 12 cores that form a single NUMA domain. As we repeat the previous tests with the two splitting approaches adapted to the different NUMA layout and core number, the trends observed in Figure 8 are reproduced, but configurations with atomics perform less noisy and the individual cost and speedup factors are different. Particularly, the overall speedup factors are smaller relative to the number of used cores, even though this can be improved by moving to the socket splitting strategy with memory allocations per thread from Figure 10. An overview over all results with the OMPT backend and clang++ with both splitting approaches and both types of processors is given in Table 3. Figure 12 shows the scaling observed with socket splitting on the Intel Xeon Gold 6126 node. Clearly, configurations without atomic updates plateau whereas configurations with atomics keep scaling up to the full node. The reason is that the first operate closer to the memory bandwidth, and become
Fig. 11. Scaling results of OpDiLib on a dual-socket AMD EPYC 7351 node for the forward pass (left) and the reverse pass (right). Includes primal scaling results in the forward pass and the ideal scaling curves. Beyond the baseline, the number of MPI processes is fixed to two (one per socket) and the number of OpenMP threads per MPI process is increased up to 16.

Fig. 12. Scaling results of OpDiLib on a dual-socket Intel Xeon Gold 6126 node for the forward pass (left) and the reverse pass (right). Includes primal scaling results in the forward pass and the ideal scaling curves. Beyond the baseline, the number of MPI processes is fixed to two (one per socket) and the number of OpenMP threads per MPI process is increased up to 12.

limited by it. The relation to the memory bandwidth utilization in the reverse pass is investigated for both the AMD EPYC 7351 node and the Intel Xeon Gold 6126 node in Figure 13. As expected from the previous scaling results, the bandwidth limit is not exceeded on the AMD node. On the Intel node, speedup beyond the hardware limit is only possible due to caching. Consequently, both the bandwidth and the scaling curves flatten.

Overall, the test results show that very good parallel AD performance can be achieved with OpDiLib. High parallel efficiencies show that OpDiLib succeeds at scaling serial tape-based AD to
Table 3. Forward Pass Results (Top) and Reverse Pass Results (Bottom) with the OMPT Backend and clang++ on the AMD EPYC 7351 Node as well as the Intel Xeon Gold 6126 Node, with the NUMA Splitting Approach from Figure 7 and the Socket Splitting Approach from Figure 10

|                      | forward pass | reverse pass |
|----------------------|--------------|--------------|
|                      | one NUMA domain, NUMA splitting | one socket, socket splitting |
|                      | one thread, time in s | 4 threads, time in s | parallel efficiency | one thread, time in s | 16 threads, time in s | parallel efficiency |
| AMD EPYC 7351        |              |              |                    |                    |              |                   |
| primal               | 0.46         | 0.14         | 0.85               | 0.46               | 0.04          | 0.73              |
| atomic adjoints      | 12.29        | 3.32         | 0.93               | 14.96              | 0.83          | 1.12              |
| adjoint access control | 12.46       | 3.53         | 0.88               | –                  | –             | –                 |
| classical adjoints   | 12.30        | 3.33         | 0.92               | 14.17              | 0.85          | 1.04              |
| Intel Xeon Gold 6126 |              |              |                    |                    |              |                   |
| primal               | 0.68         | 0.08         | 0.74               | 0.68               | 0.06          | 0.94              |
| atomic adjoints      | 13.03        | 1.34         | 0.81               | 13.16              | 1.17          | 0.94              |
| adjoint access control | 13.10       | 1.38         | 0.79               | –                  | –             | –                 |
| classical adjoints   | 13.08        | 1.34         | 0.81               | 13.15              | 1.17          | 0.94              |
|                      |              |              |                    |                    |              |                   |
| AMD EPYC 7351        |              |              |                    |                    |              |                   |
| atomic adjoints      | 23.26        | 18.33        | 0.32               | 24.26              | 1.50          | 1.01              |
| adjoint access control | 8.39        | 4.02         | 0.52               | –                  | –             | –                 |
| classical adjoints   | 8.29         | 2.23         | 0.93               | 8.65               | 0.55          | 0.98              |
| Intel Xeon Gold 6126 |              |              |                    |                    |              |                   |
| atomic adjoints      | 22.62        | 4.45         | 0.42               | 22.43              | 1.94          | 0.96              |
| adjoint access control | 4.24        | 0.78         | 0.45               | –                  | –             | –                 |
| classical adjoints   | 4.28         | 0.54         | 0.66               | 4.39               | 0.53          | 0.70              |

For better comparability, we provide the parallel efficiency, that is, speedup normalized by the number of cores. Includes primal results in the forward pass.

multiple OpenMP threads. The overall performance is strongly influenced by hardware characteristics, which can be addressed by adapting data decomposition and parallelization accordingly. The reverse pass timings are primarily influenced by atomic updates. Depending on the application, confining these to the variables that actually need them might already yield the desired speedup.

6 SUMMARY AND CONCLUSION

With OpDiLib, we have designed a tool for the automatic differentiation of OpenMP parallel codes that is fast, flexible, reusable and easy to apply. Our interpretation of AD augmentations as events alongside the OpenMP workflow allowed us to cover an unprecedented extent of OpenMP mechanisms in an operator overloading AD tool. We provide this functionality in the new open-source software OpDiLib as a universal add-on for operator overloading AD tools. We publish it together with bindings for CoDiPack and hope that it is also a useful addition to other operator overloading AD tools. In order to couple OpDiLib to another AD tool, that tool must be made thread-safe in the first step. The coupling performed in the second step provides capabilities for handling OpenMP constructs including an automatic deduction of a parallel reverse pass. With the OMPT backend, we achieve full automatization, that is, it does not require additional code changes. We do not restrict the data access patterns either. Hence, if a serial code that was already differentiated with the classical AD tool is parallelized by means of OpenMP, an initial parallel differentiated code is obtained immediately. While we apply the conservative strategy of atomic updates on the adjoint
variables, we do not prevent further optimization and equip the user with the option to disable atomic evaluation for any parts of the code. We complemented the OMPT features by an alternative macro backend, which serves as a fallback solution and also covers compilers without OMPT support. OpDiLib’s flexible design allowed us to support the different use cases and modes of operation presented in this article and will likely benefit future modifications and applications. We have demonstrated in the performance tests that all configurations of OpDiLib preserve the serial AD memory footprint and that very good parallel performance can be achieved in practice. We will enhance and improve OpDiLib in the future and work on additional features and configurations is in progress. OpDiLib’s OpenMP support as presented in this article covers all directives, clauses, and runtime functions of the OpenMP 2.5 specification [OpenMP ARB 2005] with the exception of atomics and the flush directive, as we explained in Section 3.1. This resembles the core feature set of OpenMP for classical CPU architectures. We additionally demonstrated OpDiLib’s applicability for the pure operator overloading automatic differentiation of OpenMP-MPI hybrid parallel codes. Hence, already now, OpDiLib should be of interest to many simulation codes executed on CPU computing clusters.

REFERENCES

D. Auroux and V. Groza. 2017. Optimal parameters identification and sensitivity study for abrasive waterjet milling model. Inverse Probl. Sci. Eng. 25, 11 (2017), 1560–1576. DOI: https://doi.org/10.1080/17415977.2016.1273916

A. Refik Bahadır. 2003. A fully implicit finite-difference scheme for two-dimensional Burgers’ equations. Appl. Math. Comput. 137, 1 (2003), 131–137. DOI: https://doi.org/10.1016/S0096-3003(02)00091-7

Atılım Günes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2017. Automatic differentiation in machine learning: A survey. J. Mach. Learn. Res. 18, 153 (2017), 1–43. Retrieved from https://jmlr.org/papers/v18/17-468.html.

J. Biazar and H. Aminikhah. 2009. Exact and numerical solutions for non-linear Burger’s equation by VIM. Math. Comput. Modell. 49, 7 (2009), 1394–1400. DOI: https://doi.org/10.1016/j.mcm.2008.12.006

Christian Bischof, Niels Guertler, Andreas Kowarz, and Andrea Walther. 2008. Parallel reverse mode automatic differentiation for OpenMP programs with ADOL-C. In Advances in Automatic Differentiation. Christian H. Bischof, H. Martin
OpenMP Architecture Review Board. 2018. OpenMP Application Programming Interface Version 5.0. (November 2018). Retrieved from https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf.

OpenMP Architecture Review Board. 2021. OpenMP Application Programming Interface Version 5.2. (November 2021). Retrieved from https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

Emre Özkaya and Nicolas R. Gauger. 2010. Automatic transition from simulation to one-shot shape optimization with Navier-Stokes equations. *GAMM-Mitteilungen* 33, 2 (2010), 133–147. DOI: https://doi.org/10.1002/gamm.201010011

Joachim Protze, Tim Cramer, Simon Convent, and Matthias S. Müller. 2019. OMPT-Multiplex: Nesting of OMPT tools. In *Tools for High Performance Computing 2017*. Christoph Niethammer, Michael M. Resch, Wolfgang E. Nagel, Holger Brunst, and Hartmut Mix (Eds.), Springer, Cham, Switzerland, 73–83. DOI: https://doi.org/10.1007/978-3-030-11987-4_5

M. Sagebaum, T. Albring, and N. R. Gauger. 2018. Expression templates for primal value taping in the reverse mode of algorithmic differentiation. *Optim. Methods Software* 33, 4–6 (2018), 1207–1231. DOI: https://doi.org/10.1080/10556788.2018.1471140

Max Sagebaum, Tim Albring, and Nicolas R. Gauger. 2019. High-performance derivative computations using CoDiPack. *ACM Trans. Math. Softw.* 45, 4 (2019), 26 pages. DOI: https://doi.org/10.1145/3356900

Max Sagebaum, Johannes Blühdorn, and Nicolas R. Gauger. 2020. Index Handling and Assign Optimization for Algorithmic Differentiation Reuse Index Managers. (2020). arXiv:2006.12992. Retrieved from https://arxiv.org/abs/2006.12992.

Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes. 2013. Algorithmic differentiation of a complex C++ code with underlying libraries, In *Proceedings of the 2013 International Conference on Computational Science*. Vassil Alexandrov, Michael Lees, Valeria Krzhizhanovskaya, Jack Dongarra, and Peter M. A. Sloot (Eds.), 208–217. DOI: https://doi.org/10.1016/j.procs.2013.05.184

Michel Schanen, Michael Förster, Johannes Lotz, Klaus Leppkes, and Uwe Naumann. 2012. Adjoining hybrid parallel code. In *Proceedings of the 5th International Conference on Engineering Computational Technology*. B. H. V. Topping (Ed.), Vol. 100, Civil-Comp Press, Kippen, Stirlingshire, UK, 18. DOI: https://doi.org/10.4203/cpp.100.7

Michel Schanen, Uwe Naumann, Laurent Hascoët, and Jean Utke. 2010. Interpretative adjoints for numerical simulation codes using MPI, In 2010 International Conference on Computational Science. *Procedia Comput. Sci.* 1, 1 (2010), 1825–1833. DOI: https://doi.org/10.1016/j.procs.2010.04.204

Sebastian Schlenkrich, Andrea Walther, Nicolas R. Gauger, and Ralf Heinrich. 2008. Differentiating fixed point iterations with ADOL-C: Gradient calculation for fluid dynamics. In *Modeling, Simulation and Optimization of Complex Processes*. Hans Georg Bock, Ekaterina Kostina, Hoang Xuan Phu, and Rolf Rannacher (Eds.), Springer, Berlin, Germany, 499–508. DOI: https://doi.org/10.1007/978-3-540-79409-7_36

Markus Towara and Uwe Naumann. 2013. A discrete adjoint model for OpenFOAM, In *Proceedings of the 2013 International Conference on Computational Science*. Vassil Alexandrov, Michael Lees, Valeria Krzhizhanovskaya, Jack Dongarra, and Peter M. A. Sloot (Eds.), 429–438. DOI: https://doi.org/10.1016/j.procs.2013.05.206

Tom Verstraete, Lasse Müller, and Jens-Dominik Müller. 2017. Adjoint-based design optimisation of an internal cooling channel u-bend for minimised pressure losses. *Int. J. Turbomach. Propuls. Power* 2, 2 (2017), 13. DOI: https://doi.org/10.3390/ijtpp2020010

S. Vitale, M. Pini, and P. Colonna. 2020. Multistage turbomachinery design using the discrete adjoint method within the open-source software SU2. *Journal Propul. Power* 36, 3 (2020), 465–478. DOI: https://doi.org/10.2514/1.B37685

Andrea Walther. 2009. Getting started with ADOL-C. In *Combinatorial Scientific Computing*. Uwe Naumann, Olaf Schenk, Horst D. Simon, and Sivan Toledo (Eds.), Vol. 9061, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 10. Retrieved from https://drops.dagstuhl.de/opus/volltexte/2009/2084.

Hongqing Zhu, Huazhong Shu, and Meiyu Ding. 2010. Numerical solutions of two-dimensional Burgers’ equations by discrete Adomian decomposition method. *Comput. Math. Appl.* 60, 3 (2010), 840–848. DOI: https://doi.org/10.1016/j.camwa.2010.05.051

Received 28 September 2021; revised 30 June 2022; accepted 22 October 2022