Toward Adjoinable MPI

Jean Utke* Laurent Hascoët† Chris Hill‡ Paul Hovland§
Uwe Naumann¶

Abstract

Automatic differentiation is the primary means of obtaining analytic derivatives from a numerical model given as a computer program. Therefore, it is an essential productivity tool in numerous computational science and engineering domains. Computing gradients with the adjoint (also called reverse) mode via source transformation is a particularly beneficial but also challenging use of automatic differentiation. To date only ad hoc solutions for adjoint differentiation of MPI programs have been available, forcing automatic differentiation tool users to reason about parallel communication dataflow and dependencies and manually develop adjoint communication code. Using the communication graph as a model we characterize the principal problems of adjoining the most frequently used communication idioms. We propose solutions to cover these idioms and consider the consequences for the MPI implementation, the MPI user and MPI-aware program analysis. The MIT general circulation model serves as a use case to illustrate the viability of our approach.

keywords: MPI, automatic differentiation, source transformation, reverse mode

1 Introduction

In many areas of computational science, it is necessary or desirable to compute the derivatives of functions. In numerical optimization, gradients and sometimes Hessians are used to help locate the extrema of a function. Sensitivity analysis of computer models of physical systems can provide information about how various parameters affect the model and how accurately certain parameters must be measured. Moreover, higher-order derivatives can improve the accuracy of a numerical method, such as a differential equation solver, enabling, for example, longer time steps.

*corresp. author: utke@mcs.anl.gov, University of Chicago and Argonne National Laboratory, Argonne, IL, USA
†INRIA Sophia-Antipolis, Valbonne, France
‡EAPS, MIT, Cambridge, MA, USA
§Mathematics and Computer Science, Argonne National Laboratory, Argonne, IL, USA
¶Department of Computer Science, RWTH Aachen University, Aachen, Germany

Automatic differentiation is a technique for computing the analytic derivatives of numerical functions given as computer programs. Automatic differentiation exploits the associativity of the chain rule and the finite number of intrinsic mathematical functions in a programming language to automate the generation of efficient derivative code [8]. The adjoint (or reverse) mode of automatic differentiation is particularly attractive for computing first derivatives of scalar functions, because it enables one to compute gradients at a cost that is a small multiple of the cost of computing the function and is independent of the number of input variables.

The adjoint mode of automatic differentiates combines partial derivatives according to the chain rule, starting at the output (dependent) variable and proceeding (in a direction opposite to the control and data flow of the original function computation) to the input (independent) variables. For any variable u in the original program $\mathcal{P}$, the automatic differentiation procedure creates an adjoint variable $\bar{u}$ in the adjoint program $\bar{\mathcal{P}}$. This variable represents the derivative of the output variable with respect to u. Consequently, a statement of the form `v=`ϕ`(u)` in the original program $\mathcal{P}$ results in an update of the form $\bar{\texttt{u}}$`+=`$\bar{\texttt{v}}$`*(`∂`v/`∂`u)`. Typically ϕ is some intrinsic like `sin, cos` etc. in the programming language of the numerical model to be adjoined. The assignment `v =` ϕ`(u)` may overwrite a previously used value of v. The generic formulation of the adjoint statement as an increment of the adjoint counterparts of the original right-hand-side arguments necessitates to set $\bar{\texttt{v}}$`=0` subsequent to the increment of $\bar{\texttt{u}}$. Therefore, the simple assignment `v=u` has as adjoint the two statements $\bar{\texttt{u}}$`+=`$\bar{\texttt{v}}$`;`$\bar{\texttt{v}}$`=0`. In the following we will see that this plays an important role in the practical implementation of message-passing adjoints. Because the derivative of `v` with respect to `u`, ∂`v/`∂`u`, may depend on the value of `u` and because the variable `u` may be reused and overwritten many times during the function evaluation, the derivative code must record or recompute all overwritten variables whose value is needed in derivative computation. In practice, domain-specific data flow analysis is used to identify variables whose values must be recorded, partial derivatives are "pre-accumulated" within basic blocks, and complex incremental and multilevel checkpointing schemes are employed to reduce memory requirements [21]. However, for simplicity and without loss of generality, in this paper we assume that a program (or program section) $\mathcal{P}$ is transformed into a new program section $\mathcal{P}^* = \mathcal{P}^+\bar{\mathcal{P}}$, where $\mathcal{P}^+$ runs forward, recording all overwritten variables, and $\bar{\mathcal{P}}$ runs backward, computing partial derivatives and combining them according to the chain rule. The "backward" execution is accomplished by reversing the flow of control. This implies a reversal of the statement order within basic blocks including calls to communication library subroutines.

Many large-scale computational science applications are parallel programs implemented by using MPI message passing. Consequently, in order to apply the adjoint mode of automatic differentiation to these applications, mechanisms are needed that reverse the flow of information through MPI messages. Previous work [2, 3, 9, 12, 13] has examined the automatic differentiation of parallel programs, but this work has focused primarily on the forward mode of automatic differentiation or has relied on the user to implement differentiated versions of

communication routines or other ad hoc methods. In this paper we introduce a mechanism for the adjoint mode differentiation of MPI programs, including MPI programs that use nonblocking communication primitives. Given this context we focus on qualitative statements regarding the ability to automatically create adjoint code for the most common MPI idioms and the preservation of the basic characteristics of the communication idiom. The latter plays a role in ensuring the correctness of the transformation and retaining the generic performance advantages for which a given MPI idiom may have been chosen in the original model. Incremental runtime improvements or suggestions how to improve the communication interface as a whole are beyond the scope of this paper. Consequently, we do not present the timings of the forward and the adjoint communication. Instead, we use a case study to show the principal feasibility of our approach.

In Sec. 2 we introduce the MPI idioms of concern in this paper. Section 3 briefly covers the transformation of plain point-to-point communication and details possible solutions for more complex idioms that are the main contribution of this paper. In Sec. 4 we highlight how the approach was used to automate the transformation of the communication in the MIT general circulation mode. We summarize the results in Sec. 6.

2 MPI Idioms and Program Transformation

In the following sections we omit the `mpi_` prefix from subroutine and variable names and also omit parameters that are not essential in our context. This section briefly introduces the message-passing concepts relevant to our subject. An automatic transformation of message-passing logic has to be aware of the efficiency considerations that are the reason for different communication modes and the constraints that are implied by these communication modes.

Two commonly used models for message-passing communications are the MPI control flow graph (MPICFG) [18] and the communication graph [19, pp. 399–403]. A central issue for correct MPI programs is to be deadlock free. Deadlocks can be visualized as cycles in the communication graph. A cycle indicating a deadlock and the use of reordering and buffering to resolve it are shown in Fig. 1. For the plain (frequently called "blocking") pairs of `send`/`recv` calls, the edges linking the vertices are bidirectional because the MPI standard allows a blocking implementation; that is, the `send`/`recv` call may return only after the control flow in the counterpart has reached the respective `recv`/`send` call. In complicated programs the deadlock-free order may not always be apparent. For large data sets the buffered `send` may run out of buffer space, thereby introducing a deadlock caused by memory starvation.

A third option to resolve the deadlock shown in Fig. 2(left) uses the nonblocking `isend(a,r)`, which keeps the data in the program address space referenced by variable `a` and receives a request identifier `r`. The program can then advance to the subsequent `wait(r)`, after whose return the data in the send buffer `a` is known to have been transmitted to the receiving side.

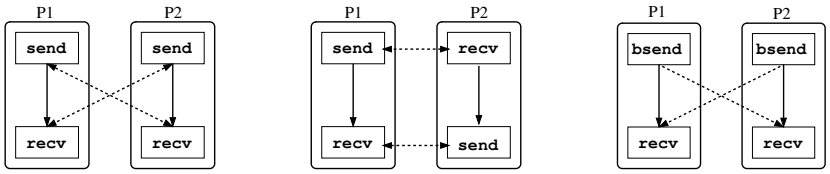

Figure 1: Two processes P1 and P2 want to send data to each other at the same time and deadlock (left). We can reorder the calls (center) to break the deadlock. We could instead keep the order but unblock the send call by using the "buffered" version `bsend`, thus making the communication dependency edges unidirectional (right) and removing the cycle.

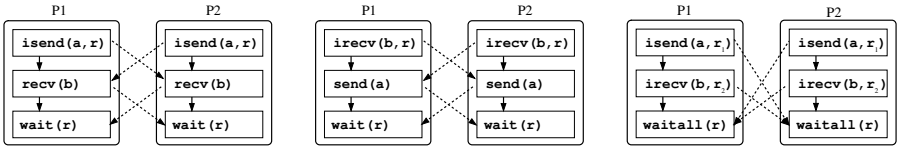

Figure 2: We use the nonblocking send variant `isend` followed by `wait` to break the deadlock scenario in Fig. 1 (left). The same result can be achieved using the nonblocking `irecv` (center) or a combination of the two (right), where we also group the `wait` operations into `waitall`.

We assume here that the input program $\mathcal{P}$ is deadlock free. However, the automatic transformation has to ensure that the transformed MPI program $\bar{\mathcal{P}}$ is also deadlock free. Thus, the transformation has to be cognizant of specific communication patterns in $\mathcal{P}$ to retain their ability to break potential deadlocks.

Other than permitting an immediate return, the variety of different modes for `send` (and `recv`) calls has its rationale in efficiency considerations. Unlike `bsend`, an `isend` avoids copying the data to an intermediate buffer but also requires that the send buffer not be overwritten until the corresponding `wait` call returns. Similarly, for an `irecv`, see Fig. 2(center), a read (or overwrite) of the receive buffer prior to the return of the corresponding `wait` returns yields undefined values. While the transformation should retain efficiency advantages for $\bar{\mathcal{P}}$, it also has to satisfy the restrictions on the buffers. Because one would like to minimize artificially imposed order on the message handling, often the individual `wait` calls are collected in a single `waitall` call; see Fig. 2(right). The `waitall` vertices in the communication graph typically have more than one communication in-edge. Two other common scenarios causing multiple communication in- and out-edges are collective communications (for instance, broadcasts and reductions) and the use of wildcard for the tag or the source parameter. In Sec. 3 we explain the consequences of multiple communication in- and out-edges.

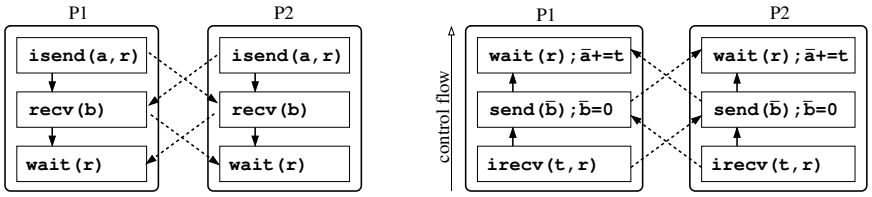

Figure 3: As in Fig. 2 we use `isend` to break a deadlock. This communication pattern can be adjoined while remaining deadlock free by replacing the MPI calls and reversing the direction of the communication edges and the control flow (right).

3 Adjoining MPI

In this section we explain the construction of the adjoint $\bar{\mathcal{P}}$ of our program section of interest $\mathcal{P}$. A direct application of a source transformation tool to an MPI implementation is impractical for many reasons. One obvious reason is that we would merely shift the need to prescribe adjoint semantics to communication operations to some lower level not covered by the MPI standard. For the transformation we will consider certain patterns of MPI library calls and propose a set of slightly modified interfaces that we can then treat as atomic units in a transformation that implements the adjoint semantic.

As in sequential programs, the adjoint $\bar{\mathcal{P}}$ will require certain variable values during its execution. These values might have been recorded in the accompanying augmented forward section $\mathcal{P}^+$. However, the particular means of restoring these values is not the subject of this paper and does not affect what is proposed here. Consequently, we do not specify $\mathcal{P}^+$ for the following examples, and we omit from $\bar{\mathcal{P}}$ any statements related to restoring the values.

One can consider a `send(a)` to be a use of the data in variable `a` and the corresponding `recv(b)` into a variable `b` to be a setting of the data in `b` that is equivalent to writing a simple assignment statement `b=a`. As explained in Sec. 1 the adjoint statements corresponding to this assignment are `ā+=b̄; b̄=0`. Applying the above analogy we can express the adjoint as `send(b̄); b̄=0` as the adjoint of the original `recv` call and `recv(t); ā+=t` as the adjoint of the original `send` call. This has been repeatedly discovered and used in various contexts (e.g., [3, 4]) and is the extent to which automatic transformation has been supporting MPI until now.

3.1 Required Context

The semantics of the adjoint computation as introduced in Sec. 1 implies that both the control flow and the communication edges have to be reversed. We already mentioned the need to preserve certain features of the communication patterns to keep the communication efficient and deadlock free; see, for example, Fig. 3. Considering the modes of `send` and `recv` calls, we can derive a

Table 1: Rules for adjoining a restricted set of MPI `send`/`recv` patterns. We omit all parameters except for the buffers `a`, `b`, a temporary buffer `t`, and the request parameter `r` for nonblocking calls.

	in $\mathcal{P}$		in $\bar{\mathcal{P}}$	
	call	paired with	call	paired with
1	`isend(a,r)`	`wait(r)`	`wait(r);`$\bar{\texttt{a}}$`+=t`	`irecv(t,r)`
2	`wait(r)`	`isend(a,r)`	`irecv(t,r)`	`wait(r)`
3	`irecv(b,r)`	`wait(r)`	`wait(r);`$\bar{\texttt{b}}$`=0`	`isend(`$\bar{\texttt{b}}$`,r)`
4	`wait(r)`	`irecv(b,r)`	`isend(`$\bar{\texttt{b}}$`,r)`	`wait(r)`
5	`bsend(a)`	`recv(b)`	`recv(t);`$\bar{\texttt{a}}$`+=t`	`bsend(`$\bar{\texttt{b}}$`)`
6	`recv(b)`	`bsend(a)`	`bsend(`$\bar{\texttt{b}}$`);`$\bar{\texttt{b}}$`=0`	`recv(t)`
7	`ssend(a)`	`recv(b)`	`recv(t);`$\bar{\texttt{a}}$`+=t`	`ssend(`$\bar{\texttt{b}}$`)`
8	`recv(b)`	`ssend(a)`	`ssend(`$\bar{\texttt{b}}$`);`$\bar{\texttt{b}}$`=0`	`recv(t)`

set of patterns where simple rules suffice for the adjoint program transformation. For simplicity we consider `send(a);` to be equivalent to `isend(a,r); wait(r);` and similarly for `recv`. For `send(a)`, that is `isend(a,r); wait(r)`, we apply rule 1 and reverse the control flow, obtaining `irecv(t,r); wait(r);` $\bar{\texttt{a}}$`+=t`, that is, `recv(t);` $\bar{\texttt{a}}$`+=t`. When all communication patterns in a program $\mathcal{P}$ match one of the rules listed in Table 1, then one can replace the respective MPI calls as prescribed. Together with control flow reversal orchestrated by the regular adjoint transformation, the correct reversal of the communication edges then is implied. A framework to formally prove these rules can be found in [17] and we prove a specific case in Sec. 3.4. As evident from the table entries the proper adjoint for a given call depends on the context in the original code. One has to facilitate the proper pairing of the `isend`/`irecv` calls with their respective individual `wait`s for rules 1–4 (intra-process) and also of `send` mode for a given `recv` for rules 5–8 (inter-process). An automatic code analysis will often be unable to determine the exact pairs. Instead one could either use the notion of communication channels identified by pragmas [12] or wrap the MPI calls into a separate layer. This layer needs to encapsulate the required context information (e.g., via distinct `wait` variants) and potentially passes the respective user space buffer as an additional argument; for example, `sendwait(r,a)` may be paired with `isend(a,r)`. Likewise the layer would introduce distinct `recv` variants; for example, `brecv` would be paired with `bsend`. Note that combinations of nonblocking, synchronous and buffered send and receive modes not listed in Table 1 can be easily derived. For instance, the adjoint of a sequence of `ibsend(a,r)` - `recv(b)` - `wait(r)` involves rule 2 for the `wait` and rule 5 for the `recv`, resulting in the adjoint sequence `irecv(t,r)` - `bsend(`$\bar{\texttt{b}}$`);`$\bar{\texttt{b}}$`=0` - `wait(r);`$\bar{\texttt{a}}$`+=t`.

3.2 Wildcards and Collective Communication

The adjoining recipes have so far considered only cases where the vertices in the communication graph have single in- and out-edges. Using the MPI wildcard values for parameters `source` or `tag` implies that a given `recv` might be paired with any `send` from a particular set; that is, the `recv` vertex has multiple communication in-edges only one of which at any time during the execution is actually traversed. Transforming the `recv` into a `send` for the adjoint means that we need to be able to determine the destination. A simple solution is to *record* the values of the tag and source parameters in the augmented forward version $\mathcal{P}^+$ at run-time.[1] Conceptually this could be interpreted as a run-time incarnation of the communication graph in which the set of potential in- or out-edges has been replaced by the one communication that actually takes place. Thus, the single in- and out-edge property is satisfied again. In $\bar{\mathcal{P}}$ the wildcard parameters are replaced with the actual values that were recorded during the execution of $\mathcal{P}^+$, thus ensuring that we traverse the correct, inverted communication edge. One can show that for any deadlock-free run-time incarnation of the communication graph, one can construct a corresponding adjoint communication graph that will also be deadlock free.

For collective communications the transformation of the respective MPI calls is essentially uniform across the participating calls. To illustrate the scenario, we can consider a summation reduction followed by a broadcast of the result, which could be accomplished by calling `allreduce` but here we want to do it explicitly. In $\mathcal{P}$ we sum up to the rank 0 process `reduce(a,b,+)` (i.e. $\mathtt{b}_0 = \sum \mathtt{a}_i$) followed by `bcast(b)` (i.e. $\mathtt{b}_i = \mathtt{b}_0 \forall i$). The corresponding adjoint statements in $\bar{\mathcal{P}}$ with a temporary variable `t` and reversed control flow are $\mathtt{t}_0 = \sum \bar{\mathtt{b}}_i$ followed by $\bar{\mathtt{a}}_i$`+=`$\mathtt{t}_0 \forall i$, which, expressed as MPI calls, are `reduce(`$\bar{\mathtt{b}}$`,t,+)` followed by `bcast(t);`$\bar{\mathtt{a}}_i$`+=t`. In short, a reduction becomes a broadcast and vice versa. The respective communication graphs are shown in Fig. 4. For the purpose of the automatic differentiation program transformation the multiple communication in- or out-edges can conceptually be treated as single communication channel visualized as a hyperedge in the communication graph.[2]

To expose an efficiency concern, we modify the above example slightly to perform a product reduction instead of the summation. The transformation remains the same except for the increments $\bar{\mathtt{a}}_i$`+=`$(\partial \mathtt{b}_0 / \partial \mathtt{a}_i)\mathtt{t}_0 \forall i$ that follow the `bcast` in $\bar{\mathcal{P}}$. The above formula for the $\bar{\mathtt{a}}_i$ does not suggest how exactly to compute the partials $\partial \mathtt{b}_0 / \partial \mathtt{a}_i$. In principle, the partials could be explicitly computed by using prefix and suffix reduction operations during the recording sweep [13]. Alternatively one could record the $\mathtt{a}_i$ per process in $\mathcal{P}^+$ and then in $\bar{\mathcal{P}}$ first restore the $\mathtt{a}_i$, then compute all the intermediate products from the leaves to the root in the reduction tree, followed by propagating the adjoints from the root to the leaves [1]. This approach requires only two passes over the tree and thus is less costly than any approach using the explicit computation

[1] They are returned with the `status` of the `recv` call.

[2] Assuming a hyperedge in general might suggest a synchronization among the participating collective calls, which is not required by the standard.

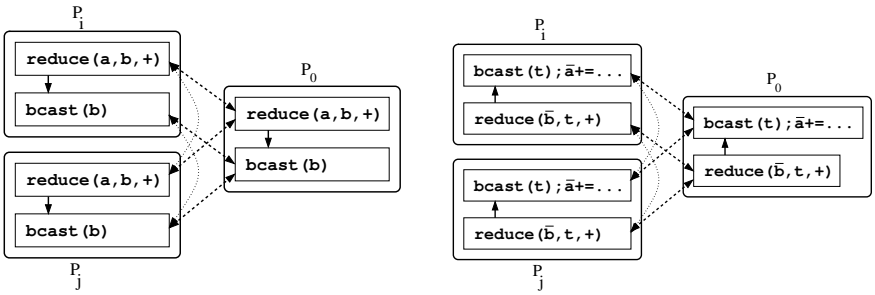

Figure 4: Collective operations reduction and broadcast in $\mathcal{P}$ are all connected with communication edges among themselves (left). If we can identify the rank 0 process among the collective calls, the dotted communication edges can be removed. The adjoint inverts the control flow but keeps the bidirectional communication edges grouped in the same hyper edges in $\bar{\mathcal{P}}$ (right).

of the partials. Unlike the explicit partials computation using pre- and postfix reductions, MPI does not provide interfaces facilitating the two-pass approach; consequently, one would have to implement it from scratch.

3.3 Grouping `wait` Operations

The grouping of sets of `wait` operations into a call to `waitall` or `waitsome` can increase the communication efficiency by removing the often artificial order among the requests. It also leads to multiple communication in-edges; see Fig. 5(left).[3] Typically, more than one or even all of these in-edges are traversed, thereby distinguishing the scenario from the wildcard receive case we considered in Sec. 3.2. A transformation solely based on the rules in Table 1 would require first modifying $\mathcal{P}$ such that all the grouped `wait` operations are split into individual `wait`s. While they could then can be transformed into the respective `isend` and `irecv` calls shown in Fig. 5(right), we would in the process lose the potential performance advantage that prompted the use of `waitall` in the first place. Without loss of generality we consider a sequence of `isend` calls, followed by a sequence of `irecv` calls, followed by a `waitall`, as shown in Fig. 5. While there are no communication edges directly between the `isend`s and `irecv`s, we know that in principle we want to turn send into receive operations and vice versa. Replacing `isend`s and `irecv`s in $\mathcal{P}$ with `irecv`s and `isend`s in $\bar{\mathcal{P}}$ begs the question of where in $\bar{\mathcal{P}}$ the corresponding `wait`s should go. This gives us the rationale to introduce a symmetric counterpart to `waitall` into $\mathcal{P}$ that we call `awaitall`, which stands for *anti-waitall*. We illustrate the scenario in Fig. 6(left). In $\mathcal{P}$ there no semantics are assigned to `awaitall`, and the vertex and the communication edges can be considered nonoperational. The adjoint transformation shown in Fig. 6(right) makes them operational and in symmet-

[3] For simplicity we assume all processes have the same behavior, and we show only the *condensed MPI CFG* [18].

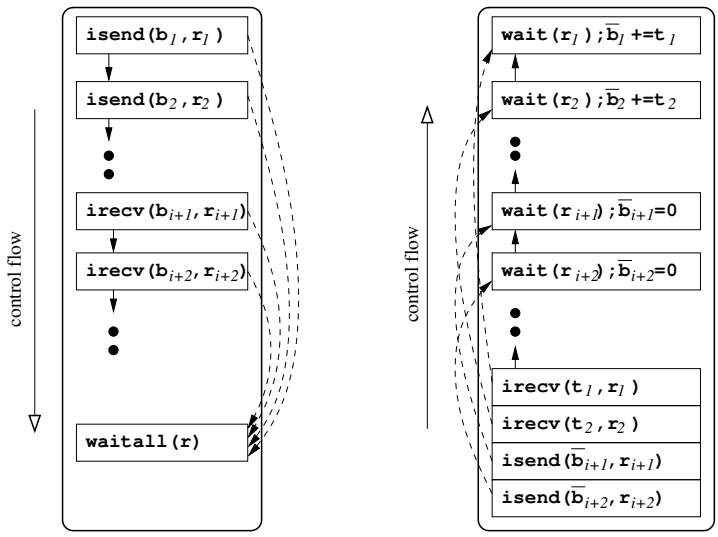

Figure 5: The completion `waitall` has multiple communication in-edges that make a simple vertex based adjoint transformation impossible (left). Without further information, the `waitall` is split into individual `wait`s, producing a less efficient adjoint code (right) where there is no `waitall`.

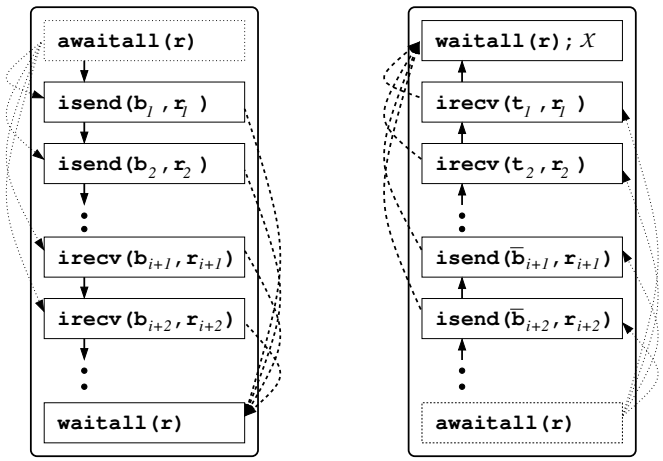

Figure 6: We introduce into $\mathcal{P}$ a symmetric nonoperational counterpart to `waitall` called `awaitall` (left). The adjoint transformation to $\bar{\mathcal{P}}$ turns `awaitall` into `waitall`, `isend` into `irecv` and vice versa (right).

rical fashion renders nonoperational the out-edges of the `awaitall` vertex in $\bar{\mathcal{P}}$ that corresponds to the `waitall` in $\mathcal{P}$. The final $\mathcal{X}$ at the top Fig. 6(right) denotes the adjoint buffer updates $\bar{\texttt{b}}_j\texttt{+=t}_j, j = 1, \ldots, i$ and $\bar{\texttt{b}}_j = 0, j = i+1, \ldots$ that have to wait for completion of the nonblocking calls. The following section proves the correctness of this transformation and gives the rationale for symmetrically extending the restrictions on writing and reading the `isend` and `irecv` buffers to the entire section between the `awaitall` and the `waitall`.

3.4 Correctness of Adjoint Communication Patterns

The technique used in this section can be applied to prove all the adjoint transformation recipes discussed in this paper. We illustrate it using one of the recipes. Consider the *partitioned global address space* (PGAS) versions $\mathcal{P}^=$ of the message-passing program $\mathcal{P}$ involving n processes $p_1, \ldots, p_n$; see also [5]. The partitioning augments all program variables with an additional dimension of length n. Communications are translated into assignments between the augmented program variables. Additional variables are introduced for buffered communication. Barriers in asynchronous communication yield a set of sequentialized versions for a given message-passing program. For example, the processes p_1 and p_2 performing a single nonblocking send/receive

s_0; `if(this is` p_1`)` `isend(`a`,`r`);` s_1; `if(this is` p_2`)` `irecv(`b`,`r`);` s_2; `wait(`r`);` s_3

with some unspecified statements $s_0, \ldots, s_3$ yield the following six distinct PGAS variants.

1:	$s_0;\ s_1^2;\ b^2 = a^1;\ s_1^1;\ s_2;\ s_3$	**2:**	$s_0;\ s_1;\ b^2 = a^1;\ s_2;\ s_3$
3:	$s_0;\ s_1;\ s_2^2;\ b^2 = a^1;\ s_2^1;\ s_3$	**4:**	$s_0;\ s_1;\ s_2^1;\ b^2 = a^1;\ s_2^2;\ s_3$
5:	$s_0;\ s_1^2;\ s_2^2;\ b^2 = a^1;\ s_1^1;\ s_2^1;\ s_3$	**6:**	$s_0;\ s_1;\ s_2;\ b^2 = a^1;\ s_3$

The superscripts indicate the executing process or address subspace, respectively. Note that $(s_i^1;\ s_i^2) = (s_i^2;\ s_i^1)$ as a result of the disjoint address spaces. Hence, the sequence of statements $s_i;\ s_{i+1}$ yields the following six semantically equivalent PGAS sequences:

1:	$s_i^1;\ s_{i+1}^1;\ s_i^2;\ s_{i+1}^2$	**2:**	$s_i^2;\ s_{i+1}^2;\ s_i^1;\ s_{i+1}^1$	**3:**	$s_i^1;\ s_i^2;\ s_{i+1}^1;\ s_{i+1}^2$
4:	$s_i^1;\ s_i^2;\ s_{i+1}^2;\ s_{i+1}^1$	**5:**	$s_i^2;\ s_i^1;\ s_{i+1}^1;\ s_{i+1}^2$	**6:**	$s_i^2;\ s_i^1;\ s_{i+1}^2;\ s_{i+1}^1$

The partial order of the statements is induced by $s_i^j < s_{i+1}^j$. Any two statements from s_i^j and s_{i+1}^k can be executed in arbitrary order for $j \neq k$. Further combinations resulting from feasible (with respect to data dependence) switches of the buffer assignment and statements in certain s_i^j lead to an exponential number of possible actual execution orders that need to be taken into account when proving properties of PGAS programs. The `isend` restrictions imply that a^1 not be written by s_1 or s_2. Similarly, b^2 may not be read by s_2. To prove the correctness of an adjoint of a message-passing program we need to show that its adjoint PGAS versions $\overline{(\mathcal{P}^=)}$ are equivalent to the PGAS versions of its adjoint

$(\bar{\mathcal{P}})^=$. We will do so for the pattern shown in Fig. 6 rewritten as follows.

$$s_{i-1}\texttt{; awaitall(}r\texttt{);}\ s_{i+1}\texttt{;}\ \ldots\ s_{j-1}\texttt{; isend(}a\texttt{,}r_\nu\texttt{);}\ s_{j+1}\ \ldots$$
$$\ldots\ s_{k-1}\texttt{; irecv(}b\texttt{,}r_\nu\texttt{);}\ s_{k+1}\texttt{;}\ \ldots\ s_{l-1}\texttt{; waitall(}r\texttt{);}\ s_{l+1}$$

We consider n processes for this pattern in $\mathcal{P}$ with $\nu = 1, \ldots, n$, $a \neq b$, $j = j(\nu)$, and $k = k(\nu)$. Reversing the control flow and then applying the recipe yields for $\bar{\mathcal{P}}$

$$\overline{s_{l+1}}\texttt{; awaitall(}r\texttt{);}\ \overline{s_{l-1}}\texttt{;}\ \ldots\ \overline{s_{k+1}}\texttt{; Xisend(}\bar{b}\texttt{,}r_\nu\texttt{);}\ \overline{s_{k-1}}\ \ldots$$
$$\ldots\ \overline{s_{j+1}}\texttt{; Xirecv(}\bar{a}\texttt{,}r_\nu\texttt{);}\ \overline{s_{j-1}}\texttt{;}\ \ldots\ \overline{s_{i+1}}\texttt{; waitall(}r\texttt{);}\ \overline{s_{i-1}}$$

where we fold some buffer operations[4] into the respective send and receive call such that

$$\texttt{Xisend(}\bar{b}\texttt{,}r_\nu\texttt{)} \equiv \text{isend}(\bar{b}\texttt{,}r_\nu\texttt{)} \rightsquigarrow \bar{b}\texttt{=0}$$

and

$$\texttt{Xirecv(}\bar{a}\texttt{,}r_\nu\texttt{)} \equiv \text{irecv}(t\texttt{,}r_\nu\texttt{)} \rightsquigarrow \bar{a}\texttt{+=}t$$

When we require that a not be written and b not be read by any statement $s_{i+1}, \ldots, s_{l-1}$, we can show that the $\bar{\mathcal{P}}$ shown above is correct as follows.

Proof: Consider the possible $\mathcal{P}^=$ versions with placeholders $\triangleright$ and $\triangleleft$ for the respective wait statements in positions i and l and $\bigtriangledown$ for the `irecv` statement in position k:

$$\begin{aligned}
&\textbf{1:}\ s_{i-1};\ \triangleright;\ s_{i+1}\ \ldots\ s_{k-1};\ \bigtriangledown;\ b^{\nu_2} = a^{\nu_1};\ s_{k+1}\ \ldots\ s_{l-1};\ \triangleleft;\ s_{l+1}\\
&\qquad\vdots\\
&\textit{l–k:}\ s_{i-1};\ \triangleright;\ s_{i+1}\ \ldots\ s_{k-1};\ \bigtriangledown;\ s_{k+1}\ \ldots\ s_{l-1};\ b^{\nu_2} = a^{\nu_1};\ \triangleleft;\ s_{l+1}
\end{aligned}$$

With the reversal of control flow their adjoints are as follows.

$$\begin{aligned}
&\textbf{1:}\ \overline{s_{l+1}};\ \triangleleft;\ \overline{s_{l-1}}\ \ldots\ \overline{s_{k+1}};\ \bar{a}^{\nu_1}\texttt{+=}\bar{b}^{\nu_2};\ \bar{b}^{\nu_2}\texttt{=0};\ \bigtriangledown;\ \overline{s_{k-1}}\ \ldots\ \overline{s_{i+1}};\ \triangleright;\ \overline{s_{i-1}}\\
&\qquad\vdots\\
&\textit{l–k:}\ \overline{s_{l+1}};\ \triangleleft;\ \bar{a}^{\nu_1}\texttt{+=}\bar{b}^{\nu_2};\ \bar{b}^{\nu_2}\texttt{=0};\ \overline{s_{l-1}}\ \ldots\ \overline{s_{k+1}};\ \bigtriangledown;\ \overline{s_{k-1}}\ \ldots\ \overline{s_{i+1}};\ \triangleright;\ \overline{s_{i-1}}
\end{aligned}$$

Because $\mathcal{P}$ satisfies the `isend`/`irecv` restrictions, the variable a^{ν_1} is not written by any of the statements $s_{j+1}, \ldots, s_{l-1}$, nor is b^{ν_2} read or written by $s_{k+1}, \ldots, s_{l-1}$. The variable a^{ν_1} may be read by any of the statements $s_{j+1}, \ldots, s_{l-1}$ implying that $\bar{a}^{\nu_1}$ may be incremented in $\overline{s_{l-1}}, \ldots, \overline{s_{j+1}}$ while $\bar{b}^{\nu_2}$ remains unchanged by $\overline{s_{l-1}}, \ldots, \overline{s_{k+1}}$. The only read operation on $\bar{a}^{\nu_1}$ within $\overline{s_{l-1}}, \ldots, \overline{s_{j+1}}$ is performed as part of these possibly present increment operations. In exact arithmetic the increment order for $\bar{a}^{\nu_1}$ has no impact on its final value. Hence, the statement $\bar{a}^{\nu_1}\texttt{+=}\bar{b}^{\nu_2}$ followed by $\bar{b}^{\nu_2}\texttt{=0}$ can be inserted at any position between $\overline{s_{l+1}}$ and $\overline{s_{k-1}}$. In short, we constructed $\overline{(\mathcal{P}^=)}$ and found the adjoints of all PGAS versions are equivalent.

We now consider $(\bar{\mathcal{P}})^=$ where $\triangle$ is the placeholder for the `isend` statement in position j

[4] In an implementation, these buffer operations have to be delayed until the data is transmitted or received, respectively.

$$
\begin{array}{ll}
\textbf{1:} & \overline{s_{l+1}};\ \lhd;\ \overline{s_{l-1}}\ \ldots\ \overline{s_{j+1}};\ \triangle;\ t{=}\bar{b}^{\nu_2};\ \bar{b}^{\nu^2}{=}0;\ \bar{a}^{\nu_1}{+}{=}t;\ \overline{s_{j-1}}\ \ldots\ \overline{s_{i+1}};\ \rhd;\ \overline{s_{i-1}} \\
& \vdots \\
\textbf{\textit{l–k:}} & \overline{s_{l+1}};\ \lhd;\ \overline{s_{l-1}}\ \ldots\ \overline{s_{j+1}};\ \triangle;\ \overline{s_{j-1}}\ \ldots\ \overline{s_{i+1}};\ t{=}\bar{b}^{\nu_2};\ \bar{b}^{\nu^2}{=}0;\ \bar{a}^{\nu_1}{+}{=}t;\ \rhd;\ \overline{s_{i-1}}
\end{array}
$$

At this point the need to impose additional restrictions on $\bar{a}^{\nu_1}$ and $\bar{b}^{\nu_2}$ becomes apparent. The temporary variable t is neither read nor written by any of the statements $\overline{s_{l-1}}, \ldots, \overline{s_{i-1}}$. It can be removed as the result of copy-propagation. In the $(\bar{\mathcal{P}})^{=}$ versions the adjoint computation takes place between $\overline{s_{l+1}}$ and $\overline{s_{k-1}}$. The same computation is performed in $\overline{(\mathcal{P}^{=})}$ between $\overline{s_{j+1}}$ and $\overline{s_{i-1}}$. Hence, the variable $\bar{b}^{\nu_2}$ must not be written by any of the statements in $\overline{s_{j-1}}, \ldots, \overline{s_{i+1}}$. Moreover, $\bar{a}^{\nu_1}$ may only be incremented by these statements. Therefore, in $\mathcal{P}$ the variable a^{ν_1} may not be written and b^{ν_2} may not be read or written by $s_{i+1}, \ldots, s_{j-1}$. This reflects exactly our additional requirement, and therefore the proposed transformation is shown to be correct ■

3.5 Placement Flexibility and Implementation Choices

In the program section between the `awaitall` and the `waitall` our augmented restriction on the `isend` and `irecv` buffers is symmetric. Just like the `waitall`, the placement of the `awaitall` will have to be done by the MPI programmer who wishes to use the automatic differentiation transformation. While the main goal of this paper is a transformation via recipes applied at the level of a subroutine call, we note that the restrictions in turn permit some flexibility to move the `isend` and `irecv` calls within this program section. The situation is illustrated in Fig. 7. This would afford the maximal time the message-passing system can spend to process communication requests before further computation in the participating processes is halted pending the return of the respective `waitall`. Unlike the transformation recipes we proposed so far, such a modification of the original program requires detailed data dependency information. With a few exceptions, practical message-passing programs will likely not be amenable to an automatic program analysis that can provide the data dependencies with sufficient accuracy. On the other hand, it is perfectly reasonable to consider as a starting point that communication channels in the program are identified by pragmas. Together with pragmas that serve as semantic placeholders for the `awaitall` position, all the information required to apply the adjoint transformation recipes would be present. Obviously, standard program analyses still are required to establish data dependencies for all the other parameters in the MPI calls, their position in the control flow graph, and so forth.

From the above it is obvious that an approach based on pragma-identified communication channels would be the most beneficial for general purposes. It is clearly also a rather complicated choice for something that can also be implemented with a set of subroutines that wrap the MPI calls and supply all the required context information via parameters and context-specific versions. An example, mentioned in Sec. 3.1, is a specific `sendwait`, which takes as an additional parameter the corresponding `isend` call's sendbuffer. The wrapper

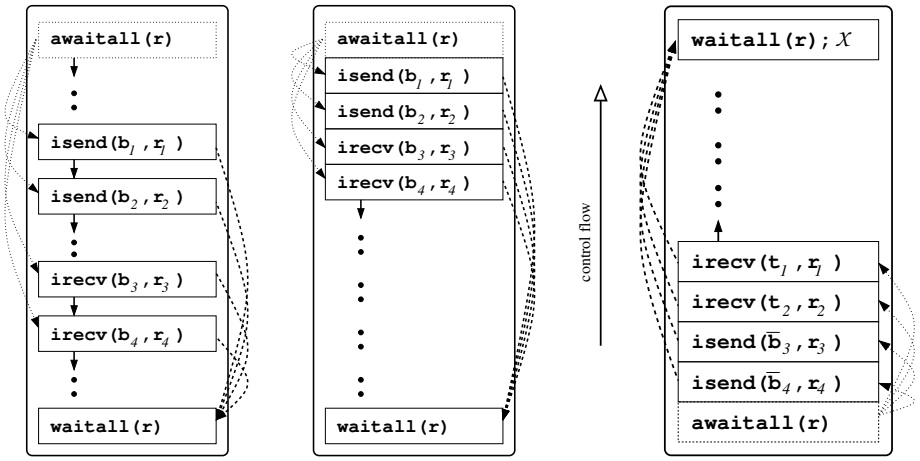

Figure 7: Because the restrictions on the communication buffers are symmetric in $\mathcal{P}$ (left) we can maximize the time between posting a communication request and waiting for the completion by shifting the calls for $\mathcal{P}^+$ (center) and $\bar{\mathcal{P}}$ (right).

routines then can switch their behavior, perhaps via some global setting, between the original and the respective adjoint semantics indicated by the recipe. Some additional bookkeeping for the delayed buffer operations to be executed by the adjoint semantic for `awaitall` is in principle all that is needed to accomplish the task. More details on this can be found in Sec. 4 related to our wrapper-based prototype implementation.

Motivated by efficiency considerations for computing the partial derivatives in Sec. 3.2 and folding the buffer semantics into an `Xisend` or an `Xirecv` call as in Sec. 3.4, one might attempt to integrate such interfaces into an MPI implementation, thereby avoiding an extra layer of calls and replicated bookkeeping logic. Such implementation details are beyond the scope of this paper and subject to ongoing research and discussion in the automatic differentiation community.

4 Case Study: MITgcm

The MIT General Circulation Model (MITgcm) is an ocean and atmosphere geophysical fluids simulation code [15, 16] that is widely used for both realistic and idealized studies and runs on both serial desktop systems and large-scale parallel systems. It employs a grid-point based, time-stepping algorithm that derives its parallelism from spatial domain decomposition in two horizontal dimensions. Coherence between decomposed regions, in both forward and reverse mode adjoint computations, is handled explicitly by a set of hand-written communication and synchronization modules that copy data between regions using either shared-memory or MPI-based messaging. The MITgcm code supports ar-

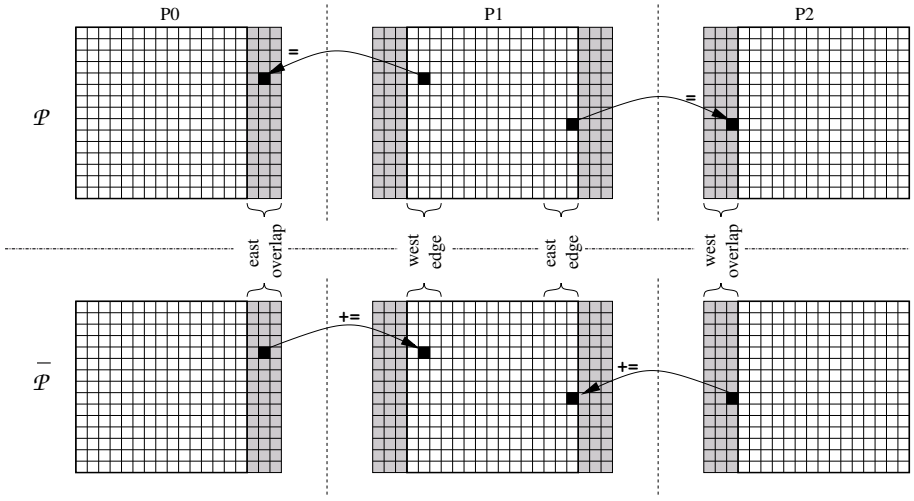

Figure 8: In $\mathcal{P}$ the data at the eastern and western edge of P1 are copied to the respective overlap arrays (shaded gray) of its neighbors P0 and P2. In $\bar{\mathcal{P}}$ the adjoint of this operation is to increment the adjoint edge data in P0 by the adjoint overlap date from its neighbors P0 and P2.

bitrary cost functions [7,11,14,20] for which adjoints can be generated with the automatic differentiation tools TAF and OpenAD/F. Until now, however, the automatic adjoint transformation did not extend to the MPI communication layer. Instead, hand-written "adjoint forms" of the MITgcm communication and synchronization modules have to be maintained [9,10] and substituted into the code generated automatically for the other parts of the MITgcm. The lack of tool support required other ocean model developers to adopt the same strategy [4]. Creating and maintaining these hand-written adjoint sections are arduous and highly error-prone tasks, particularly when multiple discretization and decomposition options require many variants of the communication logic. This situation provided the impetus to investigate to what extent automatic transformation might support communication patterns that are more sophisticated than plain `send-recv` pairs.

Figure 8 illustrates the approach for a hypothetical two-dimensional array (the hatched blocks on the diagram) that has been block-decomposed along one dimension over three processes. The gray-shaded areas, hold copies of data that is updated by the respective neighbor. In order to maintain coherence, in forward simulation mode, data is sent from the appropriate index regions on P1 (labeled "west edge" and and "east edge") and received by processes P0 and P2 into the index regions (labeled "east overlap" and "west overlap"). The arrows in the $\mathcal{P}$ row in Fig. 8 illustrate this data traffic for the updates that P1 sends to its neighbors. The respective adjoint operation is shown in $\bar{\mathcal{P}}$ row. The communication graph for the data exchange in Fig. 8 is shown in Fig. 9. In practice

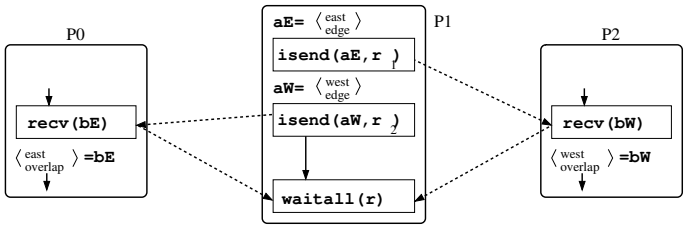

Figure 9: The communication graph for the data exchange in $\mathcal{P}$ illustrated in Fig. 8.

the data exchange is of course symmetric (P1 also receives data from P0 and P2), periodic (P0 and P2 are also neighbors), and the decomposition is done in two dimensions. In order to avoid issues of buffer overflow and deadlock, we use `isend()`. The subsequent `waitall` covers the `isend` calls to all neighbors. Note that many lines of ancillary code may occur between the posting of the `isend` operations and the call to the balancing `waitall`. Without automatic transformation capabilities those program section will have to be manually adjoined as well.

To apply our recipe to the `waitall` operation requires the insertion of the `awaitall`. Because of the aforementioned symmetry of the communication in practice all processes behave the same and we can illustrate the adjoint communication by the condensed MPI-CFG shown in Fig. 10(left). On the right we show the wrapper routines inserted into the code in place of the original calls. To reach a correct solution we again consider inverting the edge direction in the communication pattern made symmetric by the insertion of the `awaitall`. Consequently the `recv` is transformed into `isend`, and the `isend` into a `recv`. Regarding the `recv` turned `isend` we could either impose restrictions on the `recv` buffer that are identical to the restrictions imposed on an `irecv` buffer in the `awaitall` - `waitall` section or accept the spatial overhead of using a temporary buffer instead. Because the restrictions would have required considerable code changes we employed the temporary buffer option. As part of the symmetric pattern the user code passes a request parameter to the originally blocking call. The primary reason is of course the need to accommodate the passing of the actual request in the adjoint but one will observe that adding these parameters to the interface reflects the very same symmetry that is the basis of our adjoining recipe. At this point it may be worthwhile to point out that the superficially similar sequence `irecv` - `send` - `waitall` would *not* permit bypassing additional restrictions by means of introducing a temporary buffer. Here, the `irecv` as the adjoint counterpart of the `send` will have to rely on the restrictions outlined in Sec. 3.4 which in essence in the original code permit a replacement of the `send` with an `isend`. Clearly, this "limitation" has to weighed against the less efficient fall back option of manually splitting the `waitall` call into individual `wait`s on the one hand, or writing the code in $\mathcal{P}$ to satisfy the `isend` restriction which in turn can improve communication performance in $\mathcal{P}$.

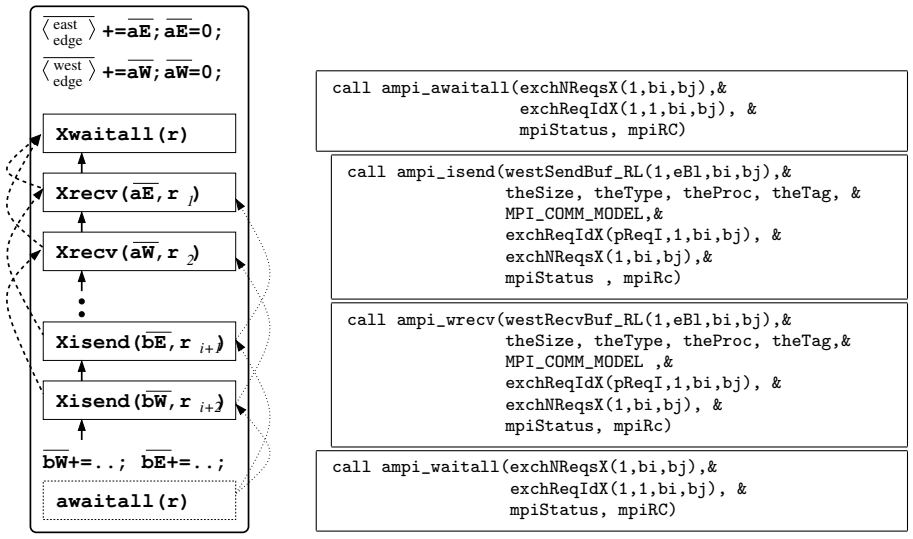

Figure 10: The condensed MPI CFG for the adjoint (left) of the fully symmetric and periodic data exchange uses wrapper routines that encapsulate buffer operations here labeled `Xisend`, `Xrecv`, and `Xwaitall`. The user code is shown in the snippets on the right.

The fact that the pairing of `isend`s with `irecv`s is not only preferable from the overall message-passing performance point of view but also permits an easier program transformation is a rather neat confluence of concepts.

The additional buffer parameters are not strictly necessary. The wrapper could internally associate requests with buffers. On the other hand, the parameters are a simple reminder to the user how far the scope of the buffer must extend. The wrapping approach permits a source transformation with a simple recipe that directly applies to the wrapped calls and does not require additional pragma information. Consequently it does not have the same utility for MPI-aware data-flow analysis. Aside from the extra subroutine call, another source of overhead is the need to retain separate receive buffers.

5 Related Work and Outlook

Most of what has been published regarding message passing in the automatic differentiation context relates to the conceptually simpler forward mode starting with [12]. The correct association between program variables and their respective derivatives under MPI might be considered a negligible implementation issue but has been a practical problem for the application of automatic differentiation in the past [2, 13] and is an issue for the adjoint transformation as well. Regarding the adjoint mode in particular the description either restricts itself to plain `send` - `recv` pairs [3, 4] or describe the hand-written program sections that

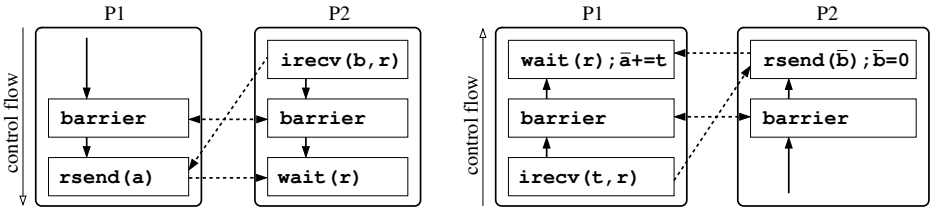

Figure 11: The standard requires that a `recv` has to be posted by the time `rsend` is called, which typically necessitates a `barrier` (left). The adjoint pattern (right) requires a placeholder for inserting the `wait` in P1.

"manually" adjoin the communication [9, 10] without an automatic generation concept for more sophisticated communication patterns.

The aim of our paper is to show an approach to the programming of message-passing logic that guarantees an automatic adjoint transformation can be carried out by applying rules to subroutine calls. Whether the rules are identified by pragmas or by a specific set of modified interfaces is an implementation issue. We have as of yet not covered all communications patterns supported by the MPI standard. One frequently used MPI call is `barrier`, for instance in the context of an `rsend`, see Fig. 11. In a logically correct program we can leave the `barrier` call in place for the adjoint transformation. For the adjoint the `barrier` call stays in place, however, a vertex transformation recipe again requires context information and a nonoperational counterpart to the `rsend` to make the pattern symmetric, e.g. we can introduce an *anti* `rsend` or an appropriate communication channel pragma. We cannot claim to have a prototype with complete coverage of all constructs provided by the current MPI standard. However, just like the MPI standard itself evolves to meet user demands we can expand the coverage of an adjoinable MPI paired with automatic differentiation tools based on the techniques explained in this paper. The prototype implementation done for the MITgcm use case can serve as a starting point but reaching a consensus among the main tool developers how an adjoinable MPI should be implemented is the eventual goal.

6 Summary

We provide the rationale for automating the adjoint transformation of message-passing programs with the need for efficient gradient computation afforded by automatic differentiation and the difficulty of achieving this goal by other means. The paper discusses the options for automatically generating an adjoint program for frequently used communication patterns in message-passing programs. We show necessary and sufficient requirements to ensure a subroutine call based set of transformation recipes yields a correct result. The basis for deriving the recipes are communication graphs. The adjoining semantics requires the inversion of the communication edge direction and we need to keep the result-

ing program deadlock free and efficient. To achieve both goals we introduce additional edges and vertices which make the communication graph symmetric with respect to the edge direction. A PGAS representation of the basic communication patterns provides the basic framework to prove the correctness of the transformations we propose and we show the proving technique on one particular pattern. The automatic transformation tool has to be able to recognize the communication calls participating in a particular pattern. Because we want to guarantee the automatic adjoinability of the message-passing program in question we do not want to rely on program analysis that may or may not be able to discern the patterns correctly. Instead, we propose to have the application programmer either distinguish patterns by means of pragma-identified communication channels or via using a set of specific wrapper routines that distinguish message-passing operations otherwise identical in MPI based on their pattern context. Compared to the alternative of having to hand-code the adjoint communication the added effort required from the application programmer is rather negligible. Pursuing the approach of identifying communication channels permits improved data flow analysis and opens opportunities of program modifications beyond the generation of adjoints. A use case for our approach was the communication logic implemented in the MITgcm ocean model. We demonstrated the ability to replace the hand-written adjoint communication layer with an automatically generated one. Our future work will concentrate on exploring the implementation options and provide a comprehensive solution that can be used with multiple automatic differentiation tools.

Acknowledgments

We would like to thank Bill Gropp, Rusty Lusk, Rajeev Thakur and Darius Buntinas for providing insights into the MPI standard rationale and the MPICH implementation. Hovland and Utke are supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy under Contract DE-AC02-06CH11357. Hill and Utke are partially supported by the NASA Modeling Analysis and Prediction Program.

References

[1] E. Boman, D. Bozdag, U. Catalyurek, K. Devine, A. Gebremedhin, P. Hovland, A. Pothen, and M. Strout. Enabling high performance computational science through combinatorial algorithms. In *SciDAC 2007, J. Phys.: Conf.Ser.*, volume 78, page 012058(10pp), Bristol, Philadelphia, 2007. IOP.

[2] Alan Carle and Mike Fagan. Automatically differentiating MPI-1 datatypes: The complete story. In Corliss et al. [6], chapter 25, pages 215–222.

[3] C.Faure, P.Dutto, and S.Fidanova. Odysée and parallelism: Extension and validation. In *Procceedings of The 3rd European Conference on Numerical Mathematics and Advanced Applications, Jyväskylä, Finland, July 26-30, 1999*, pages 478–485. World Scientific, 2000.

[4] B. Cheng. A duality between forward and adjoint MPI communication routines. In *Computational Methods in Science and Technology*, pages 23–24. Polish Academy of Sciences, 2006.

[5] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47, New York, USA, 2005. ACM.

[6] George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors. *Automatic Differentiation of Algorithms: From Simulation to Optimization*, Computer and Information Science, New York, 2002. Springer.

[7] S. Dutkiewicz, M.J. Follows, P. Heimbach, and J. Marshall. Controls on ocean productivity and air-sea carbon flux: An adjoint model sensitivity study. *Geophys. Res. Let.*, 33:L02603, 2006.

[8] Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, 2000.

[9] P. Heimbach, C. Hill, and R. Giering. Automatic generation of efficient adjoint code for a parallel Navier-Stokes solver. In J.J. Dongarra, P.M.A. Sloot and C.J.K. Tan, editor, *Computational Science – ICCS 2002*, volume 2331 of *Lecture Notes in Computer Science*, pages 1019–1028. Springer-Verlag, Berlin (Germany), 2002.

[10] Chris Hill, Alistair Adcroft, Daniel Jamous, and John Marshall. A strategy for terascale climate modeling. In *Proceedings of the Eighth ECMWF Workshop on the Use of Parallel Processors in Meteorology*, pages 406–425. World Scientific, 1999.

[11] Chris Hill, Véronique Bugnion, Mick Follows, and John Marshall. Evaluating carbon sequestration efficiency in an ocean circulation model by adjoint sensitivity analysis. *J. Geophysical Research*, 109(C11005), 2004. doi:10.1029/2002JC001598.

[12] P. Hovland. *Automatic Differentiation of Parallel Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[13] Paul D. Hovland and Christian H. Bischof. Automatic differentiation of message-passing parallel programs. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 98–104, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[14] J. Marotzke, R. Giering, K. Q. Zhang, D. Stammer, C. Hill, and T. Lee. Construction of the Adjoint MIT Ocean General Circulation Model and Application to Atlantic Heat Transport Sensitivity. *J. Geophysical Research*, 104(C12):29,529–29,547, 1999.

[15] J. Marshall, C. Hill, L. Perelman, and A. Adcroft. Hydrostatic, quasi-hydrostatic and nonhydrostatic ocean modeling. *J. Geophysical Research*, 102, C3:5,733–5,752, 1997.

[16] MIT general circulation model. http://mitgcm.org. source code, documentation, links.

[17] Uwe Naumann, Laurent Hascoët, Chris Hill, Paul Hovland, Jan Riehme, and Jean Utke. A framework for proving correctness of adjoint message passing programs. Technical Report AIB-2008-06, RWTH Aachen University, 2008. accepted for proceedings of EuroPVM/MPI 2008; online at http://aib.informatik.rwth-aachen.de/2008/2008-06.ps.gz.

[18] D. Shires, L. Pollock, and S. Sprenkle. Program flow graph construction for static analysis of MPI programs. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 99)*, 1999.

[19] Marc Snir and Steve Otto. *MPI - The Complete Reference: The MPI Core*. MIT Press, Cambridge, MA, USA, 1998.

[20] D. Stammer, C. Wunsch, R. Giering, C. Eckert, P. Heimbach, J. Marotzke, A. Adcroft, C. Hill, and J. Marshall. Volume, heat, and freshwater transports of the global ocean circulation 1993–2000, estimated from a general circulation model constrained by World Ocean Circulation Experiment (WOCE) data. *J. Geophysical Research*, 108(C1):3007–3029, 2003.

[21] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, 34(4), 2008.