Toward adjoinable MPI

Toward Adjoinable MPI

Jean Utke*†, Laurent Hascoët‡, Patrick Heimbach§, Chris Hill§, Paul Hovland†, and Uwe Naumann¶
*University of Chicago, Chicago, IL, USA
†Argonne National Laboratory, Argonne, IL, USA, Email: [utke|hovland]@mcs.anl.gov
‡INRIA Sophia-Antipolis, Valbonne, France, Email: laurent.hascoet@sophia.inria.fr
§EAPS, MIT, Cambridge, MA, USA, Email: [heimbach|cnh]@mit.edu
¶Department of Computer Science, RWTH Aachen University, Aachen, Germany, Email: naumann@stce.rwth-aachen.de

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.

Automatic differentiation (AD) is a technique for computing the analytic derivatives of numerical functions given as computer programs. Unlike finite difference approximations these analytic derivatives are computed with machine precision. AD 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 [1]. The adjoint (or reverse) mode of AD 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 – unlike finite difference gradient approximations – is independent of the number of input variables. This makes reverse mode AD the only technology that can feasibly compute gradients of large scale numerical models that can have 10^8 or more input variables.

The adjoint mode of automatic differentiation 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 AD 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 $\mathtt{v}{=}\phi(\mathtt{u})$ in the original program $\mathcal{P}$ results in an update of the form $\bar{\mathtt{u}}{+}{=}\bar{\mathtt{v}}{*}(\partial\mathtt{v}/\partial\mathtt{u})$. Typically ϕ is some intrinsic function like `sin, cos` etc. in the programming language of the numerical model to be adjoined. The assignment $\mathtt{v} = \phi(\mathtt{u})$ 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 $\bar{\mathtt{v}}{=}0$ subsequent to the increment of $\bar{\mathtt{u}}$. Therefore, the simple assignment $\mathtt{v}{=}\mathtt{u}$ has as adjoint the two statements $\bar{\mathtt{u}}{+}{=}\bar{\mathtt{v}}\mathtt{;}\ \bar{\mathtt{v}}{=}0$. 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, $\partial\mathtt{v}/\partial\mathtt{u}$, 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 [2]. However, for simplicity and without loss of generality, in this paper we

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

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 AD to these applications, mechanisms are needed that reverse the flow of information through MPI messages. Previous work [3], [4], [5], [6], [7] has examined the AD 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 on 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. Typical MPI Idioms

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.

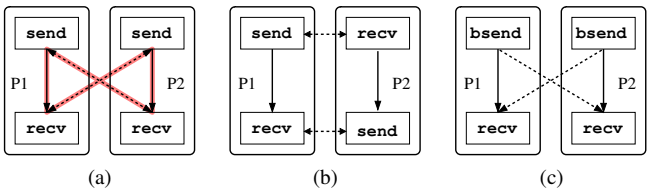


Figure 1. Deadlock(a), reordering(b), buffering(c)

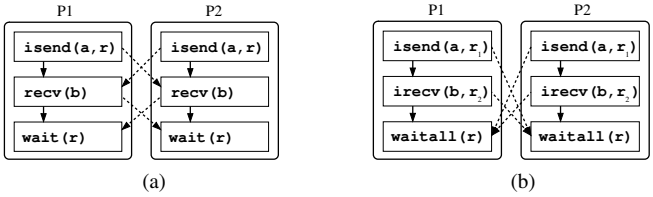


Figure 2. Immediate return `send` (a) and combination of `isend` and `irecv` with `waitall` (b)

Two commonly used models for message-passing communications are the MPI control flow graph (MPICFG) [8] and the communication graph [9, 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. In Fig. 1(a) we show a cycle (in red) indicating a deadlock when two processes P1 and P2 want to send data to each other at the same time. We can reorder the calls, see Fig. 1(b), to break the deadlock. Alternatively we could keep the order but unblock the `send` call by using the "buffered" version `bsend`, thus making the communication dependency edges unidirectional, see Fig. 1(c). 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(a) 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. 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. Another nonblocking variant is a sequence `irecv` – `send` – `wait`. A read (or overwrite) of the receive buffer prior to the return of the corresponding `wait` 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(b), where we combine `isend` and `irecv`. 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.

3. Adjoining MPI Idioms

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 $\bar{a}$`+=`$\bar{b}$`;` $\bar{b}$`=0`. Applying the above analogy we can express the adjoint as `send(`$\bar{b}$`);` $\bar{b}$`=0` as the adjoint of the original `recv` call and `recv(t);` $\bar{a}$`+=t` as the adjoint of the original `send` call. This has been repeatedly discovered and used in various contexts (e.g., [6], [10]) and is the extent to which automatic transformation has been supporting MPI until now.

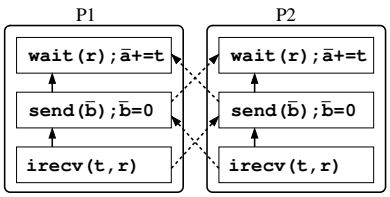


Figure 3. Adjoint of Fig. 2(a).

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. 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. Considering the modes of `send` and `recv` calls, we can derive a set of patterns where simple rules suffice for the adjoint program transformation. Table 1 shows 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. 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);` $\bar{a}$`+=t`, that is, `recv(t);` $\bar{a}$`+=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 [11]. As

Table 1. Adjoining rules

in $\mathcal{P}$		in $\bar{\mathcal{P}}$	
call	paired with	call	paired with
`isend(a,r)`	`wait(r)`	`wait(r);`$\bar{a}$`+=t`	`irecv(t,r)`
`wait(r)`	`isend(a,r)`	`irecv(t,r)`	`wait(r)`
`irecv(b,r)`	`wait(r)`	`wait(r);`$\bar{b}$`=0`	`isend(`$\bar{b}$`,r)`
`wait(r)`	`irecv(b,r)`	`isend(`$\bar{b}$`,r)`	`wait(r)`
`bsend(a)`	`recv(b)`	`recv(t);`$\bar{a}$`+=t`	`bsend(`$\bar{b}$`)`
`recv(b)`	`bsend(a)`	`bsend(`$\bar{b}$`);`$\bar{b}$`=0`	`recv(t)`
`ssend(a)`	`recv(b)`	`recv(t);`$\bar{a}$`+=t`	`ssend(`$\bar{b}$`)`
`recv(b)`	`ssend(a)`	`ssend(`$\bar{b}$`);`$\bar{b}$`=0`	`recv(t)`

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 [3] 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, `swait(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(`$\bar{\mathtt{b}}$`);`$\bar{\mathtt{b}}$`=0` - `wait(r);`$\bar{\mathtt{a}}$`+=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. 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. Collective operations reduction and broadcast in $\mathcal{P}$ are all connected with bidirectional communication edges among themselves. The adjoint inverts the control flow but keeps the same bidirectional communication edges in $\bar{\mathcal{P}}$.

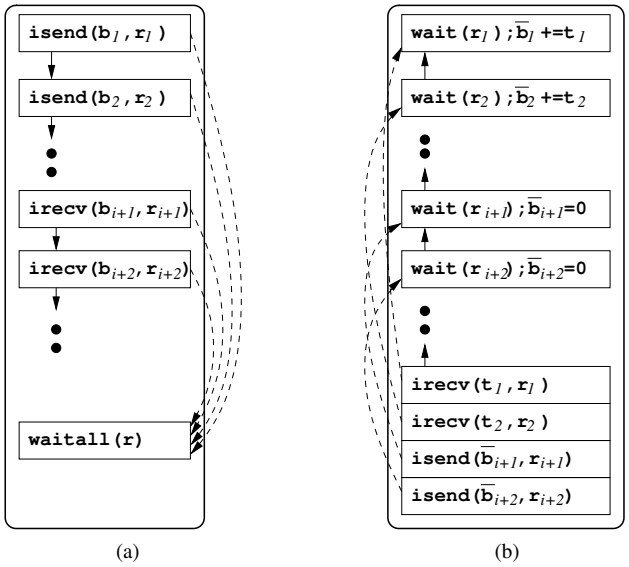


Figure 4. Completion `waitall` with multiple in-edges (a), and adjoint variant with individual `wait` calls (b).

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 [4]. 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 [12]. This approach requires only two passes over the tree and thus is less costly than any approach using the explicit computation 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. The completion `waitall` has multiple commu-

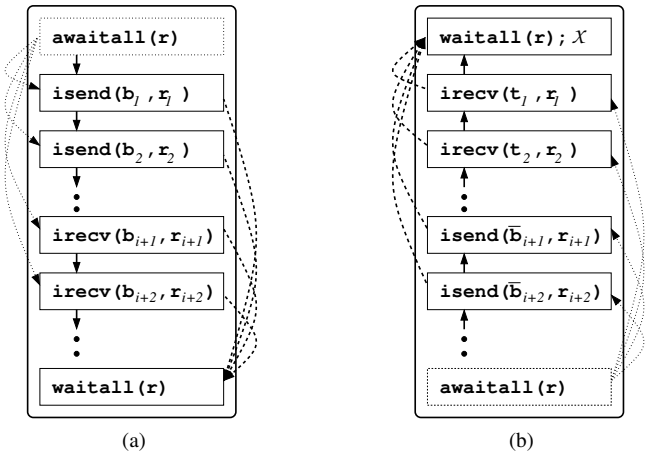


Figure 5. Introduce an `awaitall` vertex (a), and the adjoint of this graph (b).

nication in-edges, see Fig. 4(a), that make a simple vertex based adjoint transformation impossible. For simplicity we assume all processes have the same behavior, and we show only the *condensed MPI CFG* [8]. 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 be transformed into the respective `isend` and `irecv` calls shown in Fig. 4(b), 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`. 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. 5(a). In $\mathcal{P}$ no semantics are assigned to `awaitall`, and the vertex and the communication edges can be considered nonoperational. The adjoint transformation shown in Fig. 5(b) makes them operational, the `awaitall` turns into a `waitall` and in symmetrical fashion it 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 of Fig. 5(b) denotes the adjoint buffer updates $\bar{\mathtt{b}}_j$+=$\mathtt{t}_j, j = 1,\ldots,i$ and $\bar{\mathtt{b}}_j = 0, j = i+1,\ldots$ that have to wait for completion of the nonblocking calls. 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` follows from the prove of the correctness of this transformation that can be done using a framework described in [11].

3.4. 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 AD 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 to the respective ends. 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.

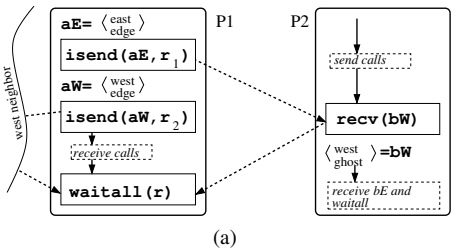
From the above it is obvious that an approach based on pragma-identified communication channels would be the most beneficial for general purposes. It is clearly also a rather complicated choice for something that can also be implemented with a set of subroutines that wrap the MPI calls and supply all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

4. Case Study: MITgcm

The MIT General Circulation Model (MITgcm) is an ocean and atmosphere geophysical fluids simulation code [13], [14] 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 arbitrary cost functions [15], [16], [17], [18] for which adjoints can be generated with the AD 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 [19], [7] 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 [10]. 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.

The communication pattern is an east-west/north-south exchange of ghost cells between neighbors in the domain decomposition. The communication graph for the data exchange for receiving the data from the western neighbor P1 into P2's ghost cells is shown in Fig. 6(a). In $\widetilde{\mathcal{P}}$ the adjoint of this operation is to increment the adjoint data in P1 by the adjoint ghost cell data from P2. In practice the data exchange is of course symmetric, periodic, and two dimensional. 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`. In Fig. 6(b) 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`



(a)

```
call ampi_awaitall(exchNReqsX(1,bi,bj),&
                   exchReqIdX(1,1,bi,bj), &
                   mpiStatus, mpiRC)
```

send data to eastern neighbor

```
call ampi_isend(westSendBuf_RL(1,eBl,bi,bj),&
                theSize, theType, theProc, theTag, &
                MPI_COMM_MODEL,&
                exchReqIdX(pReqI,1,bi,bj), &
                exchNReqsX(1,bi,bj),&
                mpiStatus , mpiRc)
```

```
call ampi_wrecv(westRecvBuf_RL(1,eBl,bi,bj),&
                theSize, theType, theProc, theTag,&
                MPI_COMM_MODEL ,&
                exchReqIdX(pReqI,1,bi,bj), &
                exchNReqsX(1,bi,bj), &
                mpiStatus, mpiRc)
```

receive data from eastern neighbor

```
call ampi_waitall(exchNReqsX(1,bi,bj),&
                  exchReqIdX(1,1,bi,bj), &
                  mpiStatus, mpiRC)
```

(b)

Figure 6. Ghost cell exchange graph (a) and user code snippet for the adjoinable MPI interface (b).

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 restrictions which in essence in the original code permit a replacement of the `send` with an `isend`; see also [11]. Clearly, this "limitation" has to be 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 $\mathcal{P}$. The fact that the

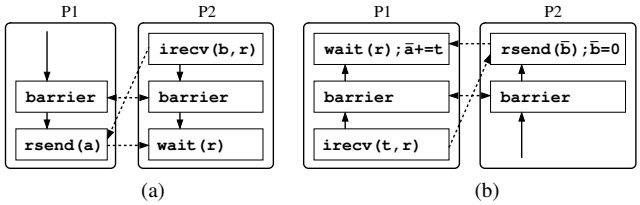


Figure 7. Graph for `rsend` (a) and its adjoint (b).

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 AD context relates to the conceptually simpler forward mode starting with [3]. 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 AD in the past [4], [5] and is an issue for the adjoint transformation as well. Regarding the adjoint mode in particular, one finds statements restricted to plain `send` - `recv` pairs [6], [10] or descriptions of the hand-written program sections that "manually" adjoin the communication [19], [7] 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`. The standard requires that a `recv` has to be posted by the time `rsend` is called, which typically necessitates a `barrier`, see Fig. 7(a). In a logically correct program we can leave the `barrier` call in place for the adjoint transformation. Similarly to the cases in Table 1, a vertex transformation recipe for the `rsend` adjoint requires context information and a nonoperational counterpart to make the pattern symmetric. For instance, we can introduce an *anti* `rsend` or an appropriate communication channel pragma. The adjoint pattern is shown in Fig. 7(b).

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 AD 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

Automating the adjoint transformation of message-passing programs is necessary for efficient gradient computation via AD and is difficult to achieve 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 resulting 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. 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 let 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 AD 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. Heimbach, Hill and Utke are partially supported by the NASA Modeling Analysis and Prediction Program.

References

[1] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed., ser. Other Titles in Applied Mathematics. Philadelphia, PA: SIAM, 2008, no. 105.

[2] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch, "OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes," *ACM Transactions on Mathematical Software*, vol. 34, no. 4, 2008.

[3] P. Hovland, "Automatic differentiation of parallel programs," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1997.

[4] P. D. Hovland and C. H. Bischof, "Automatic differentiation of message-passing parallel programs," in *Proceedings of IPPS/SPDP*. Los Alamitos, CA: IEEE Computer Society Press, 1998, pp. 98–104.

[5] A. Carle and M. Fagan, "Automatically differentiating MPI-1 datatypes: The complete story," in *Automatic Differentiation of Algorithms: From Simulation to Optimization*, ser. Computer and Information Science, G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, Eds. New York: Springer, 2002, ch. 25, pp. 215–222.

[6] 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*. World Scientific, 2000, pp. 478–485.

[7] P. Heimbach, C. Hill, and R. Giering, "Automatic generation of efficient adjoint code for a parallel Navier-Stokes solver," in *Computational Science – ICCS 2002*, ser. Lecture Notes in Computer Science, J. Dongarra, P. Sloot, and C. Tan, Eds. Berlin: Springer, 2002, vol. 2331, pp. 1019–1028.

[8] 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.

[9] M. Snir and S. Otto, *MPI - The Complete Reference: The MPI Core*. Cambridge, MA, USA: MIT Press, 1998.

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

[11] U. Naumann, L. Hascoët, C. Hill, P. Hovland, J. Riehme, and J. Utke, "A framework for proving correctness of adjoint message-passing programs," in *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 316–321.

[12] 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.*, vol. 78. Bristol, Philadelphia: IOP, 2007, p. 012058(10pp).

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

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

[15] 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*, vol. 104, no. C12, pp. 29,529–29,547, 1999.

[16] C. Hill, V. Bugnion, M. Follows, and J. Marshall, "Evaluating carbon sequestration efficiency in an ocean circulation model by adjoint sensitivity analysis," *J. Geophysical Research*, vol. 109, no. C11005, 2004, doi:10.1029/2002JC001598.

[17] 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*, vol. 108, no. C1, pp. 3007–3029, 2003.

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

[19] C. Hill, A. Adcroft, D. Jamous, and J. Marshall, "A strategy for terascale climate modeling," in *Proceedings of the Eighth ECMWF Workshop on the Use of Parallel Processors in Meteorology*. World Scientific, 1999, pp. 406–425.