

Semantics Driven Adjoints of the Message Passing Interface

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH
Aachen University zur Erlangung des akademischen Grades eines Doktors der
Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Michel Schanen

aus

Luxemburg, Luxemburg

Berichter: Univ.-Prof. Dr. rer. nat. Uwe Naumann
Univ.-Prof. Dr.-Ing. Martin Bückner

Tag der mündlichen Prüfung: 24.10.2014

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Contents

Contents	3
1 Introduction	11
1.1 Background	11
1.2 Message Passing Paradigm Meets Algorithmic Differentiation	12
1.3 Prior Work	13
1.4 Problem Statement	14
1.5 Summary of the Results	14
2 Foundations	17
2.1 Message-Passing Interface	17
2.2 Algorithmic Differentiation	18
2.2.1 Textbook Example	19
2.2.2 Finite Differences	20
2.2.3 Tangent-Linear Model	20
2.2.4 Adjoint Model	21
2.3 Technical Considerations	22
2.3.1 Source Transformation versus Operator Overloading	22
2.3.2 Association by Name versus Association by Address	23
2.3.3 Contiguous versus Interleaved Memory Layout	23
2.3.4 Implications for MPI	23
2.4 Single Assignment Code	24
2.4.1 Nullification of the Adjoint	25
2.5 Partitioned Global Address Space	26
2.6 MPI Extended SAC	27
2.6.1 Order of Statements and Nondeterminism in MPI	29
2.6.2 Locks	29
3 Adjoint Communication	31
3.1 Observations	31
3.1.1 Efficiency	31
3.1.2 Adjoint Memory Wall	33
3.2 Generation and Correctness of Adjoint Patterns	35
3.3 Blocking Point-to-Point Communication	37
3.3.1 Pattern Runtime	39
3.4 Nonblocking Point-to-Point Communication	40
3.4.1 Pattern Runtime	42
3.4.2 Antiwait	43

3.4.3	Persistent Communication	43
3.5	Nondeterminism in MPI	44
3.6	Collective Communication	45
3.6.1	Broadcast	45
3.6.2	Reduction	47
3.6.3	Allreduce	55
3.7	Barrier	59
3.8	One-Sided Communication	59
3.8.1	Fence	62
3.8.2	Get	62
3.8.3	Put	66
3.8.4	Accumulate	68
3.8.5	Passive Target Synchronization	75
3.9	Guidelines for an Adjoinable MPI Code	76
4	Adjoint MPI Library	77
4.1	Usability	77
4.2	Adjoint MPI Implementation	79
4.2.1	Portable Data Reversal Mechanism	79
4.2.2	Nonblocking Point-to-Point Communication	81
4.2.3	Collective Reduction Communication	82
4.3	Second-Order and Higher Adjoint Communication	83
4.3.1	Tangent-Linear Communication	84
4.3.2	Second-Order Code with <code>dcc</code>	85
4.3.3	Higher Adjoints	89
5	Applications	91
5.1	Hardware Setup	91
5.2	Overview: Heat Equation	92
5.3	Use Case: NAG Fortran Compiler and Sisyphe	95
5.3.1	AD-enabled NAG Fortran Compiler (CompAD)	96
5.3.2	AMPI and CompAD	96
5.3.3	Sisyphe	99
5.3.4	Results	102
5.4	Use Case: <code>dco/c++</code> and PETSc	104
5.4.1	Integration into <code>dco/c++</code>	104
5.4.2	Motivation	107
5.4.3	PETSc	108
5.4.4	Continuous Nonlinear Solvers	109
5.4.5	Results	109
6	Discussion	113
7	Summary and Conclusion	115
8	Future Work	117

A	User Documentation	119
A.1	AD Tool Developer	120
A.2	AD Tool User	121
A.2.1	Installation	121
A.2.2	Usage	122
A.2.3	Fortran	122
A.2.4	Example	123
B	Developer Documentation	127
	Bibliography	157
	Index	163

Abstract

Access to correct derivative information is crucial in numerical simulations and optimization. While finite differences easily provide derivative approximations through perturbing a function's inputs, the adjoint derivative model is the only way of acquiring a function's gradient both at machine precision and at the same time complexity as the initial function evaluation. However, the adjoint model implies a complete data flow reversal of an executed program. The same implication holds for the Message Passing Interface (MPI) of a parallel implementation. Every communication pattern has to be reversed when the adjoint model is applied.

This work establishes a framework for the semantic analysis of MPI communication patterns. It formulates a semantic driven generation of adjoint patterns of the corresponding original patterns. The MPI standard defines the semantics of every MPI communication in English language. A more abstract representation of the MPI semantics is extracted and used in order to apply the logic of Algorithmic Differentiation (AD). Based on these adjoint pattern representations a generic adjoint MPI library is implemented that may be used semi-automatically with any AD tool. Moreover, the runtime expectation of such an implementation on current cluster systems is analyzed. The outcome is tested with two software packages used in numerical science. One is the Portable Extensible Toolkit for Scientific computation (PETSc). It is currently one of the most robust frameworks for parallel linear and nonlinear solvers that exist. The other one is Sisyphé, a sediment transport simulation software used in the context of the fluid solver OpenTELEMAC.

Acknowledgment

I am very thankful to the following people without whom this work would have been bound for failure.

- To Jan Riehme who, while integrating AMPI into Sisyphe, taught me a lot about programming in numerical science. From simple bash scripts to general approaches (result analysis, code style,...), he increased my productivity and insight by several orders of magnitude. Unfortunately, I failed at the coupled model.
- To Laurent Hascoët and Jean Utke who were perhaps the only people that listened to my adjoint MPI gibberish. I thank both for dismantling my reasonings whenever I was wrong and helping me out.
- To my supervisor Uwe Naumann who always emphasized one's strengths in doubtful times.
- To Klaus Leppkes for his hacking skills.
- To all the people at the STCE for the proofreading and keeping my spirit high. Not a single dispute in all these years tells me something.
- To the Fonds National de la Recherche of Luxembourg for the financial support under grant PHD-09-145.
- And last but not least to my girlfriend Paule Apel for proofreading my work and for the support during stressful times.

Chapter 1

Introduction

1.1 Background

This dissertation was created at the Software and Tools for Computational Engineering (STCE) institute of RWTH University in Aachen. At the time of this writing the institute’s research was primarily focused on Algorithmic Differentiation (AD). Every research topic that is linked to AD or that revolves around accumulating derivatives is potentially of interest to the research unit. This work is situated at the cross section of AD and the parallel programming paradigm defined by the message-passing interface (MPI). It describes where both worlds and their models meet and emphasizes the difficulties as well as the potential gains using both numerical tools together in the scientific computing field.

AD arose from the idea that every computer program is composed of a sequence of algebraic statements and assignments. If one neglects the I/O operations or hardware commands, every computer program in the end models a mathematical function. For any computer or numerical scientist it is clear that this abstraction itself already bears a lot of potential problems. Looking at Figure 1.1, a sine function is implemented. In mathematics the function is defined on the continuous domain of real numbers. This domain is conveyed to the numerical world of a computer through the type of the variable. Here it is the type “double” that represents a number with double precision in C++. This abstraction ignores the fact that this variable is not actually continuous anymore, but is instead bound to the machine’s precision. Furthermore, cancellation and truncation errors arise due to the internal representation and computation of numbers inside a computer’s discrete structure. The next abstraction is actually a language property. Imperative programming languages are not translatable into mathematics and vice versa. What is the meaning of for example the equal sign “=”? In a programming code there is the notion of memory locations called “reference by address”, whereas mathematics only deal with “reference by name”. A variable in a program may change its location or it may be overwritten. The equal sign relates to the contents of the memory location, whereas the mathematical equal sign relates to the logical truth of a statement.

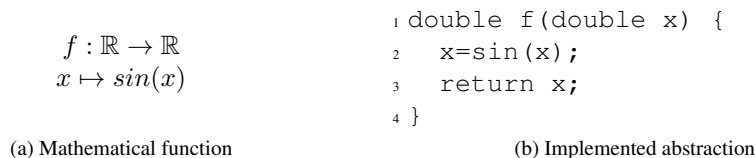


Figure 1.1: The fundamental assumption in AD

These issues will not be covered in this work and it is assumed that if any function is mathematically differentiable, the same holds for its implementation. This link between mathematics and computer code is the core idea of AD. It is its strength as well as its weakness.

In mathematics, humans are able to differentiate functions using the chain rule. The application of the chain rule is rather mechanical and does not involve any creative process. It is a task, easily described by an algorithm that is taught at high school. The differential calculus is tedious and boring, causing errors due to a slip of the pen or lacking attention. Computer algebra tools may be used to symbolically differentiate functions and relieve us humans from this ever boring task. With respect to this automatic generation, AD is the algebra tool equivalent at the source code level. The source code is manipulated in a way that it becomes the implementation of the differentiated function. Having linked mathematical functions and actual implementations and having access to enough computational power, we are today able to implement differentiation tools that apply the chain rule exactly in the same mechanical way as humans do. The development of such tools are the outcome of research in the field of AD. It assumes that if this function is continuous and partially differentiable, the same holds true for its implementation. It follows that the aforementioned computer program is differentiable as a sequence of statements by applying the chain rule to all statements in the code. Considering an implementation as a sequence of differentiable statements, we are in theory able to compute semi-automatically the gradient or Jacobian of an arbitrary code. Although the application of the chain rule is straightforward, the next question we face is about the complexity of such a computation. What is the cost of accumulating the gradient or the Jacobian compared to the original implementation? What about sparse Jacobians, gradients, conversion behaviour of the derivatives and so on?

The context when dealing with these issues is mainly in the field of large-scale numerical simulations. Currently, any large numerical code is optimized to run on computer clusters interconnected by some communication network. The de facto standard in such an abstraction of communication networks is the Message Passing Interface (MPI). This work tries to find an answer to the question whether the methods of AD may be applied to MPI communication the same way as to any sequential code.

1.2 Message Passing Paradigm Meets Algorithmic Differentiation

In parallel computing, message passing is one of the first paradigms to describe the distributed computation of a task. Sequential processes communicate over a network in order to synchronize tasks. Each process has its own memory space and is only able to access information on another process through this communication network. There is no notion of shared-memory, where different processes work in the same address space. First implementations appeared during the 80s and culminated into the open Message Passing Interface (MPI) standard in 1994 [39]. The use of MPI is not restricted to any particular field in high-performance computing. However, in this work the focus is on numerical simulation science, since this is the main field of application for AD.

In the message passing paradigm data is moved by sending a message composed of a source, a sink and a value. This paradigm is used in parallel computing to model the interprocess communication that takes place on a network between connected computer nodes. The communication abstraction is based on the physical communication network between the nodes. However, this paradigm could easily be applied to the local memory of a computer. Every copy or movement of data in the memory can be modelled as a communication. But virtually all computer languages vowed for an assignment (=) operation emerging from the mathematical equal sign (=). Looking more closely at the assignment operation one sees that this operation has actually more in common with a communication of the right-hand side to the left-hand side rather than with the mathematical equal sign. For example the statement $x = x + 2$ in Fortran does not make any sense using the semantics of mathematics. The right-hand side and left-hand side do not have the same properties. In computer programs it actually means that first the right-hand side is

evaluated and then its contents are moved (or communicated) to the left hand side. Since the assignment is rather a communication two questions need clarification. Is a communication also an assignment and in particular can the same logic that is applied to assignments in AD also be applied to communication?

1.3 Prior Work

Over the last 10-20 years AD has expanded into a separate field of research with its own community. There are regular meetings with the international conference on AD every four years and the yearly European Workshops on Automatic Differentiation. Research is split into the practical research revolving around the AD tools and theoretical research about the mathematical theory behind the methods put in into practice in AD. The community supports a website¹ where among other things the biannual meetings as well as the publications are announced.

Exploiting parallelism has become an important subject in AD due to the ubiquity of parallelism in simulation codes. It is being approached on several levels, while the focus today is on the commonly used parallelization standards OpenMP and MPI. Moreover, adjoint computations on GPUs are also making their first entry into AD.

First, algorithms have been developed that exploit the inherent parallelism of AD. This applies for example to the tangent-linear mode, adjoint mode or vector mode. Most of these methods rely on OpenMP. For example one can exploit data dependencies of first-order and second-order tangent-linear code to compute both orders of differentiation in parallel [37, 9, 8]. Or the trace of the adjoint computation may be split among the memory of several computer nodes on a cluster system and interpreted partly in parallel.

Second, given a particular class of code that is automatically parallelizable, the method in [?] generates a parallelized automatically differentiated AD code, thus combining automatic differentiation and automatic parallelization. The codes consist of matrix-matrix multiplications on a theoretical hybrid computer architecture based on arrays and trees of processing elements.

This thesis is situated at the third level of parallelization in AD where the parallelism of the original code should be conveyed to the differentiated code. The user introduces parallelism in the original code. However, the AD tool has to have the capability of differentiating the parallel statements. No tools today have a straightforward, user friendly and automatic way of differentiating any parallelism. OpenMP exploits thread based parallelism inserted through pragmas at the compiler level whereas MPI is a process based communication runtime library for parallelism at the process level. This work deals with inter process communication in MPI whereas a PhD thesis with a similar goal in OpenMP was achieved in close collaboration [16].

First attempts have been made in 1999 by automatically adjoining `MPI_Send`, `MPI_Recv` and `MPI_Reduce` in the source code transformation tool Odyssée. Odyssée automatically reverses the send and receive as well as the sum reduction (`MPI_Reduce` with `MPI_SUM`) [15, 14]. Only in recent years AD tools have become aware of any parallelism. A solution for the tangent-linear mode has been presented by using derived data types, although this is again not applied automatically [10]. TAF claims to have OpenMP and MPI support without being clear about the specifics. ADOL-C claims to have support for the MPI communications `MPI_Send`, `MPI_Recv`, `MPI_Broadcast` and `MPI_Reduce`. In general, a substantial intervention into the original code is required [52] or the support is limited to the tangent-linear mode [55] (ADIC and ADIFOR).

Sketches have been outlined about how an AD tool should adjoin MPI with the communication reversal being the main issue [29]. More complex patterns like nonblocking communication have been addressed in [62]. A general differentiation approach for MPI communication, especially for the advanced ones like for example one-sided communication, is still missing.

¹<http://www.autodiff.org/>

Automatically differentiating MPI is primarily driven by a joint effort of the Argonne National Laboratories, INRIA and our institute at the RWTH Aachen University. The first goals had been set in 2009 [62]. This is the time where this work started. The common target is a generic adjoint MPI library that may be incorporated into any AD tool in order to differentiate MPI enabled codes automatically.

1.4 Problem Statement

When applying AD to the Message-Passing Interface, whether automatically or manually, four questions need to be clarified.

- **Feasibility:** Does a reversal scheme exist? Is a given MPI pattern adjointable?
- **Closure:** Is the adjoint communication pattern still expressible in MPI and is there a possible trade-off?
- **Correctness:** Is the generated adjoint MPI code correct?
- **Efficiency:** How do the complexities of the original and adjoint communication pattern relate?

In order to prove feasibility and closure, one has first to develop an abstract framework which enables us to prove the correctness of an adjoint pattern. This abstraction is driven by the semantics of the MPI standard. The MPI standard relies for the most part on the English language as its defining language. In order to apply AD, the theoretical part of this work will be about the extraction of the semantics in a more mathematical framework. This framework allows us to prove the correctness of adjoint communication patterns that are based on heuristics. Feasibility and closure are linked to the efficiency. One has to subjectively decide whether a given code may still be considered feasible if there is a substantial decrease in efficiency. In particular, the adjoint pattern should stay in the same complexity class as the original pattern.

An adjoint MPI library is implemented for the core routines of the MPI 2.2 standard. If there exists an adjoint communication inside the MPI 2.2 standard, the adjoint generation is considered to be enclosed in MPI. However, we also require the adjoint patterns of the different classes of MPI communications (point-to-point, collective, one-sided) to be closed. Adjoining one-sided communication should for example not resort to blocking point-to-point communication.

1.5 Summary of the Results

All the classes of MPI communications in the MPI 1.1 [39] standard yield efficient adjoint communication patterns including the reduction `MPI_Reduce` which applies arithmetic operations on the buffer elements. The MPI 2.2 standard [40] introduces one-sided communication which includes nondeterministic behaviour. Nondeterministic behaviour in MPI 1.1 was restricted to wildcards, `WAIT_ANY` and `TEST_ANY`. They posed no threat to the feasibility of an efficient adjoint computation because MPI comes with the necessary information to trace the nondeterministic instance of the communications. This changes with one-sided communication. It is for the first time analyzed in detail and adjoint pattern descriptions are discussed. The outcome is that the accumulation `MPI_Accumulate`, in conjunction with the product operation `MPI_PROD`, is the first MPI communication that is proven to be not adjointable at a reasonable runtime cost. Especially passive target synchronization proves to be impossible to adjoint without user input. Nondeterminism proves to be at the heart of future development of adjoint MPI.

On top of the theoretical results and insights, an adjoint MPI library is designed and implemented. Its design requires reflection on the theoretical results as well as on the generic AD tool structure. The

implementation is then tested in Sisyphé (sediment transport simulation) and PETSc (linear and non linear parallel solver framework).

At the end, the combination of continuous and discrete adjoints seems more important than ever on current high performance computing platforms where memory is rather limited. Discrete adjoints is the straightforward application of the adjoint model at the statement level of a function's implementation. Continuous adjoints is the implementation of the mathematically derived adjoint function of the original function. Purely discrete adjoints prove to be very hard on current cluster systems due to their large memory footprint (see Section 3.1.2), whereas continuous adjoints potentially lead to a lower memory footprint than discrete adjoints (see Section 5.4). Although this is not subject of this work, the performance of AMPI is linked to the performance of discrete adjoints in general. With discrete adjoints being far from trivial on cluster machines, the use case of AMPI is currently severely limited. AMPI proves to be a great tool for adjoining the distribution and combination of data that takes place around a highly optimized computational core of a numerical simulation. Whether it is suitable for fully discrete codes remains an open question.

Chapter 2

Foundations

This chapter lays out the formalism on which this work will build upon. Starting with a brief overview of MPI (Section 2.1) and AD (Section 2.2), possible technical constraints are anticipated in Section 2.3. The single assignment code (Section 2.4) from AD is then combined with the parallel formalism of the partitioned global address space (Section 2.5). To cope with the MPI semantics additional constructs are introduced in Section 2.6.

2.1 Message-Passing Interface

The first Message-Passing Interface (MPI) standard was published in November 1992 [39]. This standard was the result of a long evolution that has its roots in one of the oldest parallel computation paradigms. It tries to abstract the combination of single computers that are merged together to form a computer *cluster*. At that time its unique feature was its open standard description. The standard was established by a committee composed of various entities ranging from industry to academia. Before, a computer cluster was a black box which handles the parallelization on a specialized hardware opaque to the user, only accessible through a special programming interface. This is still the case for some specialized shared-memory systems today. It is hard to find a common abstraction that matches all the shared-memory systems due to their highly specialized hardware. A code written in an abstract language may run efficiently on one system, while being a complete failure on another system. General open shared-

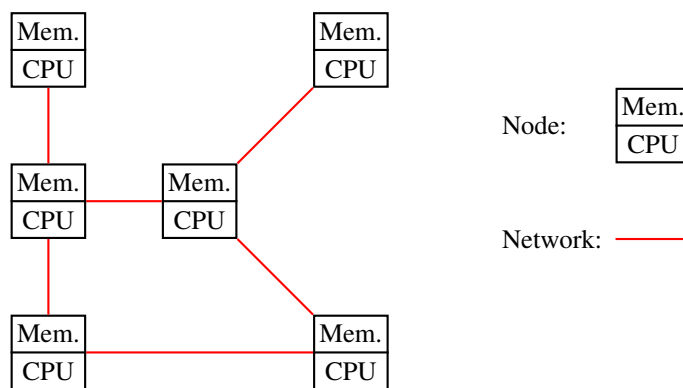


Figure 2.1: Hardware abstraction in MPI

memory standards only appeared very recently (e.g. OpenMP 1997) .

For distributed memory systems the abstraction is far easier, even if the hardware is highly specialized. Each single computer system inside the cluster is a *node* that is connected to other nodes through a *network* (see Figure 2.1). Each running process has access to all the hardware of a single node [53, 22]. In general the hardware is described by one CPU and one memory address space. In order for one process to access memory of another node, the data has to be communicated through the network. This is achieved by the dual operation of sending and receiving messages. First message passing libraries were mostly tailored to specific hardware and tied to a specific system. Soon it became apparent that the assumption of having nodes and a network was common to all the implementations. A common interface was finally defined and published in Fortran and C in 1992. MPI defines some common operations that should be executable on the computer cluster. The simplest one is a send or receive. Other operations are for example collective operations like the gathering or scattering of data. MPI cannot know how your hardware and your network with its topology may efficiently execute these operations. Hence, there is no reference implementation. Anyone may implement an MPI library while strictly adhering to the interface standard, so vendors have still to develop specialized libraries. However, the developer costs dropped dramatically, since any MPI code is portable to any cluster that supports the standard without any code amendments. This huge gain in portability had no disadvantage to other non standardized libraries and finally meant a huge success for MPI. Today, MPI has received widespread acceptance in the high performance computation community and has become the de facto standard in high performance computing. There are a wide range of MPI libraries, some for general purpose while other implementations only run on a specific hardware of one cluster type. All TOP 500 clusters support the MPI interface, meaning that any MPI enabled code runs on any of these TOP 500 clusters as well as potentially on any consumer laptop or even embedded devices.

2.2 Algorithmic Differentiation

AD revolves around the computation of derivatives using the chain rule of differential calculus. Algorithms are developed that exploit the property of the chain rule. The user is relieved from the effort of mechanically differentiating code (see motivation in Section 1.1). The error-proneness of handwritten differentiated code may drive costs higher than any budget could ever cover. One advantage of AD is indeed a considerably reduced maintenance cost of the adjoint code.

What algorithms to use in the end is a vast field of research. Symbolic differentiation has its beginning in 1953 [23]; the time around which computers were powerful enough to generate code in assembler. In 1976 the term “Automatic Differentiation” made its first appearance in the context of Fortran. Kedem observed that any subroutine $f_{\circ\circ}$ in Fortran could be transformed into a subroutine $f_{\circ\circ}'$ that computes the derivatives of subroutine $f_{\circ\circ}$ [32]. The final spark was in the 80s with Rall in 1981 [54] and Griewank around 1989 [18], making AD an independent research field.

In numerical simulation and optimization access to exact derivative information is crucial for the performance and correctness of the underlying simulation [30]. In AD it is assumed that every numerical code may be treated as a multivariate vector function $\mathbf{y} = F(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with $\mathbf{x}$ being the inputs and $\mathbf{y}$ being the outputs. Since any code may be interpreted as an input code for AD, the output code of an AD tool is again valid input for an (another) AD tool. This recursive reapplication is a way of applying second order or higher derivative models.

Extracting derivative information amounts to extracting the level of dependence between the n inputs $\mathbf{x}$ and the m outputs $\mathbf{y}$. The mathematical theory behind it, is the differential calculus denoted by partial derivatives $\frac{dy}{dx}$. Its main field of application in numerical science is optimization [13] and sensitivities, ranging from simple non-linear functions to iterative solvers or sensitivities in shallow water simulations [56]. There is no limitation to the extend of the code base as long as the AD tool is able to differentiate

the code at the statement level for a given programming language. The needed derivative information may vary from single derivative values or gradients up to entire Jacobians or Hessians.

The accumulation of derivative information yields subproblems in theoretical computer science. For single derivatives there are two differentiation models, tangent-linear and adjoint, with different runtime behaviour (see Section 2.2.3 and Section 2.2.4). Another method, called edge or vertex elimination, may be used for example in the Jacobian accumulation where all the entries of the Jacobian are computed. Finding the minimal number of floating point operations needed to accumulate the entire Jacobian has been proven to be NP-complete [42]. This last method of finding the minimal number of operations (see [21] on Accumulation by Vertex-Elimination) uses edge or vertex elimination on the directed acyclic graphs in order to extract the differentiated code. This method is not subject of this work.

2.2.1 Textbook Example

The following problem serves as a textbook example of an AD application the way it commonly occurs in simulation science. It is structurally a data assimilation problem with, for the purpose of simplification, a basic cost function. Data assimilation [31] problems with scalar cost functions emphasize the benefits of the adjoint model which is not available while using finite differences. Furthermore, it is easily decomposable into subproblems and thus embarrassingly parallel (MPI). Our code where AD is applied consists primarily of three parts:

- subproblem
- cost function
- optimization algorithm

Without the loss of generality the *subproblem* or simulation model is assumed to be a multivariate function u that computes m values with respect to inputs $\mathbf{x} \in \mathbb{R}^n$ and some undetermined or unknown parameters $\mathbf{p} \in \mathbb{R}^l$. $\mathbf{p}$ may be composed of model parameters or may just consist of the initial values of u . For some input $\mathbf{x}$, o observed values $u^{ob}(\mathbf{x})$ are measured through experiments in the real world. The discrepancy between computed values $u_i(\mathbf{x}_i, \mathbf{p})$ and observed values $u_i^{ob}(\mathbf{x}_i)$ for some input values $\mathbf{x}_i$ is the simulation error and should be 0 with an ideal simulation model. For a given number of observations o , this error is summed up as a *cost function*

$$J(\mathbf{p}) = \sum_{i=1}^o (u_i(\mathbf{x}_i, \mathbf{p}) - u_i^{ob}(\mathbf{x}_i))^2 \quad .$$

which is assumed to be only dependent on the parameters $\mathbf{p}$ for a given number of observations and input values $\mathbf{x}_i$. In a data assimilation problem like for example 3D-Var [31] the cost function J may be far more complex. For some observed values u_i^{ob} we try to minimize $J(\mathbf{p})$ through adapting the simulated values u_i by optimizing the values of $\mathbf{p}$.

For the minimization of J we use the *optimization algorithm* Steepest Descent or Gradient Descent. It is a first-order derivative based optimization algorithm that is guaranteed to converge to a local minimum.

$$\mathbf{p}_{k+1} = \mathbf{p}_k - \alpha \nabla J(\mathbf{p}_k)$$

It iteratively minimizes $J(\mathbf{p}_k)$ by computing a new value $\mathbf{p}_{k+1}$ through the subtraction of $\nabla J(\mathbf{p}_k)$. Usually, the scaling value α is determined through a line search algorithm. More details and numerical properties of this method are described in [51].

In each optimization step we want to access derivative information that describes the dependence of the l parameter inputs with respect to the single output J . Whether first-, second- or higher-order

derivatives are needed is up to the optimization method (here Steepest Descent). This example will serve as the prime motivation for the adjoint model and should illustrate why in the end, adjoints in an MPI setting are indeed important. Both case studies, PETSc in Section 5.4 and Sisyphe in Section 5.3.3, are motivated by a similar setting.

In the next sections each of the different mathematical methods of accumulating derivatives is presented and how these methods relate to this example. In particular, the focus is on the implications for MPI enabled code. For each method we consider a multivariate vector function $\mathbb{R}^n \rightarrow \mathbb{R}^m$ with $\mathbf{y} = F(\mathbf{x})$. Function F has n inputs and m outputs. We relate runtime complexity of the differentiated code with respect to the n inputs and m outputs.

2.2.2 Finite Differences

Finite differences approximation is perhaps the most widely used way of accumulating derivative information. It avoids the handwriting that results from the application of the chain rule and uses the original code to compute derivatives. It relies on the finite differences quotient to compute approximated partials $\nabla_{x_i} F$

$$\nabla_{x_i} F(\mathbf{x}) \approx \frac{F(\mathbf{x} + h \cdot \mathbf{e}_i) - F(\mathbf{x})}{h},$$

for the Cartesian basis vector $\mathbf{e}_i \in \mathbb{R}^n$ and the inputs $\mathbf{x} \in \mathbb{R}^n$. It is a direct consequence of the definition of the derivative, first described by Newton [50] and Leibniz [34]. To accumulate the entire Jacobian of our function F , we have to perturb each of the inputs and therefore rerun the entire code n times. Finally the unperturbed values have to be computed once. Thus the complexity amounts to $\mathcal{O}(n) \cdot \text{cost}(F)$. However, the finite difference quotient is only an approximation of the partial derivatives.

The reason why finite differences are being so widely used is that this method is particularly easy to implement. The original code of the function F may be used unaltered. This is especially true for any library calls like MPI where no special treatment is necessary. The code is just run once for each perturbed input and once with no perturbation at all. The approximation error represents a major drawback for codes that implement highly nonlinear systems, resulting in truncation and cancellation errors or simply providing wrong results. By applying the Taylor expansion to the second-order centered finite differences quotient, a machine precision induced approximation error of $\frac{\epsilon}{h^2}$ is derived, with ϵ being the rounding error. An additional drawback is that the runtime is always dependent on the number of inputs n . Finite differences do not provide a mode that shifts the runtime dependence from the inputs to the outputs. As we will later see, AD provides such a mode. On the upside, one may observe in practice a regularizing effect of finite differences when using a step based algorithm. Where AD may lead to non convergence, finite differences may succeed due to its smoothing effect.

MPI

As for any library, using finite differences is straightforward and no special treatment is necessary for MPI enabled codes. The underlying parallelization is used unaltered and the performance should be similar to the original code while it has to be executed $n + 1$ times, once for each direction and once for the function evaluation.

2.2.3 Tangent-Linear Model

The tangent-linear model is one of the two models in AD. Its advantage over the finite difference quotient is that it yields derivatives with machine precision. It essentially comes down to symbolic differentiation at a statement level in our code of F . The original code of F with $\mathbf{x}$ being the inputs and $\mathbf{y}$ the outputs

is transformed into the tangent-linear code manually or through an AD tool. All the details will not be provided, because the focus is on the consequences for an MPI enabled code. The implementation in tangent-linear mode F of a function f using the tangent-linear model is defined by

$$F(\underset{\downarrow}{\mathbf{x}}, \underset{\downarrow}{\mathbf{y}}) \xrightarrow{AD} F^{(1)}(\underset{\downarrow}{\mathbf{x}}, \underset{\downarrow}{\mathbf{x}}^{(1)}, \underset{\downarrow}{\mathbf{y}}, \underset{\downarrow}{\mathbf{y}}^{(1)}) \quad ,$$

where

$$\mathbf{y}^{(1)} = \nabla f(\mathbf{x}) \cdot \mathbf{x}^{(1)}$$

and $\mathbf{y} = f(\mathbf{x}) \quad ,$

and computes the directional derivative $\mathbf{y}^{(1)}$ of the outputs $\mathbf{y}$ with respect to the inputs $\mathbf{x}$. In the tangent-linear model the input direction $\mathbf{x}^{(1)}$ is *seeded* whereas the derivative values in the outputs $\mathbf{y}^{(1)}$ are *harvested*. For an entire Jacobian accumulation the model has to be run n times while seeding one of the n Cartesian basis vectors. So we end up with the same runtime complexity of $\mathcal{O}(n) \cdot \text{cost}(F)$ as finite differences, while the constant factor due to the overhead of the implementation is obviously higher. In addition to the original function values, the tangents have to be evaluated which are more expensive than the values. A remedy for this would be the vector mode, where all the n tangents could be propagated at once, but with only a single function evaluation. However, in practice vector mode may potentially increase cache misses.

MPI

The application of the tangent-linear model in MPI enabled code implies a doubling of the amount of communication. For each value x , an additional tangent-linear component $x^{(1)}$ has to be communicated. The data flow and communication patterns are preserved. However, the data that is communicated undergoes a type change. Each activated variable in the original is assumed to be either of type `MPI_DOUBLE` or `MPI_FLOAT`. Therefore, the idea is to define a new derived data type consisting of two `MPI_DOUBLE`s or two `MPI_FLOAT`s. Additionally, the missing basic MPI operations for the reductions of the newly created type have to be implemented. As an alternative to a new derived data type one may resort to communication *shadowing* by doubling the amount of communication calls; one communication for values and one for the tangents (see Section 4.3.1). The implementation of a tangent-linear MPI has been achieved as a side project of adjoint MPI.

2.2.4 Adjoint Model

$$F(\underset{\downarrow}{\mathbf{x}}, \underset{\downarrow}{\mathbf{y}}) \xrightarrow{AD} F_{(1)}(\underset{\downarrow}{\mathbf{x}}, \underset{\downarrow}{\mathbf{x}}_{(1)}, \underset{\downarrow}{\mathbf{y}}, \underset{\downarrow}{\mathbf{y}}_{(1)}) \quad ,$$

where

$$\mathbf{y} = f(\mathbf{x})$$

and $\mathbf{x}_{(1)} = \mathbf{x}_{(1)} + \nabla f(\mathbf{x})^T \cdot \mathbf{y}_{(1)} \quad .$

(2.1)

Exploiting the associativity of the chain rule of derivative calculus leads us to the adjoint model of differentiation depicted in Equation (2.1) [45], where the adjoint model of the function f is given by the implementation F using an AD tool in the adjoint mode. It increments *adjoints* $\mathbf{x}_{(1)} \in \mathbb{R}^n$ of the inputs $\mathbf{x} \in \mathbb{R}^n$ for some given adjoints $\mathbf{y}_{(1)} \in \mathbb{R}^m$ of the outputs $\mathbf{y} \in \mathbb{R}^m$. With the adjoint model we are able to compute gradients with respect to some output y_i by setting $\mathbf{y}$ to the i -th Cartesian basis vector of $\mathbb{R}^m$. The main benefit of adjoint code is its runtime dependence on the number of m outputs as opposed to the inputs with the tangent-linear model or finite-differences. Each Cartesian basis vector has to be set m

times for a Jacobian accumulation. In each run, one gradient of one output y_i with respect to all inputs $\mathbf{x}$ is computed. This is a huge gain for gradient based methods because $m = 1$.

The adjoints $\mathbf{x}_{(1)}$ of the inputs $\mathbf{x}$ are computed with respect to the adjoints $\mathbf{y}_{(1)}$ of the outputs $\mathbf{y}$. Hence, the adjoint implementation is split into two sections; the *forward section* and the *reverse section*. In the forward section the values are evaluated in the same order as in the original program. In the reverse section the adjoints are computed in the reverse order of the original program. For the adjoints $\mathbf{x}_{(1)}$ of statements that involve non-linear functions, access to the values of $\mathbf{x}$ is required for the computation of $\nabla F(\mathbf{x})$. This implies a complete data flow reversal programs with non-linear functions F . This is discussed later in more detail in Section 2.4. As the available memory is a hard limit, one can use checkpoints in order to recompute the values of $\mathbf{x}$, gaining memory efficiency but losing runtime performance. The optimal trade-off between memory efficiency and runtime performance for a given memory bound has been proven to be NP-complete [43] in the context of call tree reversal. But even considering a given code as a DAG, it has been proven NP-complete to generate a reversal scheme through edge elimination with a minimal number of operations [44].

MPI The data flow reversal imposed by the adjoint model implies a communication reversal for all MPI calls. In the forward section the values are communicated while in the reverse section the adjoints need to be passed in reverse order. Besides this theoretical requirement, there are also practical consequences for adjoints when they are computed on MPI distributed systems, in particular the memory wall that is discussed in Section 3.1.2.

2.3 Technical Considerations

This section is directed at AD tool developers. It is helpful to be familiar with the technical details of at least one AD tool. An application of an AD tool on a code is called *activation*. Operations or data structures linked to the derivative computation are called *active*. A brief overview of the different classes of AD tools is presented with certain common traits that form a categorization. These categories should be kept in mind for later as they have consequences for the handling and the structure of the active data types with regard to MPI. All categories influence each other. For example a source transformation tool tends to rely on associating by name whereas operator overloading tools tend to use association by address.

2.3.1 Source Transformation versus Operator Overloading

The first classification of an AD tool relates to the technique that is used to semantically transform the program from computing function values to computing derivatives (ADIFOR [7], Tapenade [24], OpenAD [63]). The most portable and generic method is source transformation. The source code is transformed during a compilation process by essentially replacing the compiler front end. Instead of generating only the function values, the desired differentiation model is applied to the original code. The output may be an intermediate code that is read by the back end of the compiler or the output may be again source code.

The other technique targets the very instance of a program execution at runtime (FadBad [5], ADOL-C [19], TaDiff [6], NAGWare Fortran Compiler [48], dco [58]). Changing a program semantically is attained through changing the variable type. Overloaded operators then operate on this new variable type according to the differentiation model. Of course, the underlying programming language has to support overloading of functions and operators.

2.3.2 Association by Name versus Association by Address

Before the compilation process a variable is generally associated by its name. There is no notion of data locality. By augmenting the code with statements that compute derivatives, the desired differentiation model is applied with respect to a variable's name.

Listing 2.1: "Code augmentation relying on association by name"

```

1 // Original code
2 w=u*v;
3 // Augmented code
4 w=u*v;
5 dw=du*v+dv*u;
```

In the adjoint model, required values that are overwritten have to be saved for example onto a stack. The stack pushes variables through their name.

The other option is to refer to data locality and apply the differentiation model to the contents of data. This is commonly a direct consequence of overloading tools, which introduce a new type and overloaded operators. Each time an operation is executed, the result together with a link to its arguments are saved on a trace or tape. The chain rule is not based on the variable names but on their locations on the tape. Hence, this is called association by address.

2.3.3 Contiguous versus Interleaved Memory Layout

A structural difference between the tools is their memory layout (see Figure 2.2). Depending on the implementation, values or adjoints may reside in a contiguous memory space with no disparity. Source transformation tools tend to have a contiguous layout, since they probably do not change the type of the variables. Instead, they just add additional adjoint variables. Operator overloading tools on the other hand have to introduce an active type. This active type is probably different from the passive type. So we have to deal with interleaved memory if the values in the active type or the adjoints on the tape have to be accessed.

Note that this is only a rule of thumb. One could imagine a source transformation tool with interleaved types for example when storing additional information inside the variables, thus changing their type. And there exist operator overloading tools like ADOL-C [19] that force a contiguous memory layout for the values and the adjoints for some technical reasons.

2.3.4 Implications for MPI

The association by name versus association by address have both no direct consequence for an adjoint MPI code. MPI selects data to be communicated via a buffer address. Thus, the minimum requirement for writing such code is that a source transformation tool handles pointers (C) or arrays (Fortran).

The memory layout however has big implications for an adjoint MPI code. MPI only uses pointers or arrays to access the data. There is no built in support for disparities except for special communications e.g. `MPI_Gather/MPI_Scatter`. There are two choices. Either the entire active and adjoint type is communicated with all the additional information that is stored beyond the value and the adjoint. This may lead to clashes and is not guaranteed to work. The other way is to copy the values and adjoints in the active types into a continuous buffer that is then communicated. So when dealing with interleaved data types there is a trade off. When requiring zero copy, the interleaved types have to be converted to contiguous storage by the AD tool. A generic interface that is compatible with all AD tools leads to an additional copying of the data.

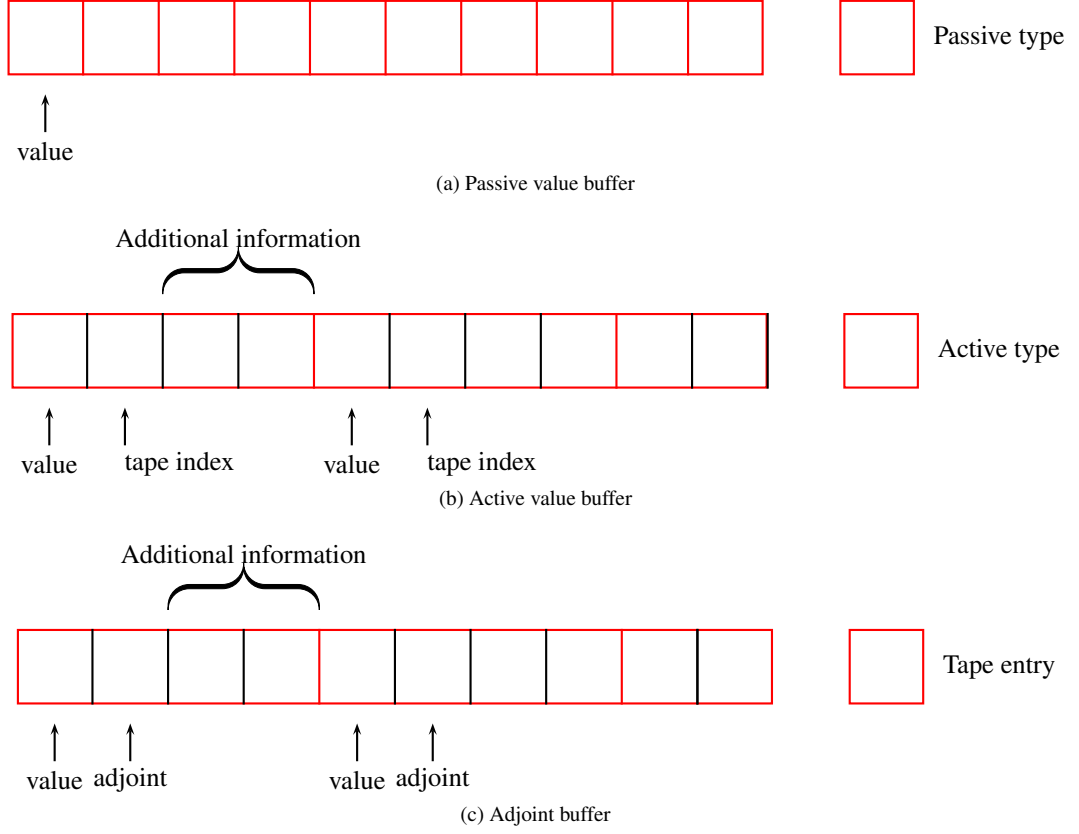


Figure 2.2: Non contiguous memory access

2.4 Single Assignment Code

Without the loss of generality, it is assumed that every statement that does not involve MPI amounts to an assignment. It follows that the entire code is decomposed as follows into a sequence of single assignments called Single Assignment Code (SAC) [45]:

$$\begin{aligned} &\text{for } j = n + 1, \dots, n + p + m \\ &\quad v_j = \varphi_j(v_i)_{i \prec j} \end{aligned} \quad (2.2)$$

where $i \prec j$ denotes a direct dependence of v_j on v_i . The result of each function φ_j is assigned to a unique auxiliary variable v_j . The n *independent inputs* $x_i = v_i$, for $i = 1, \dots, n$, are mapped onto m *dependent outputs* $y_j = v_{n+p+j}$, for $j = 1, \dots, m$, and involve the computation of the values of p *intermediate variables* v_k , for $k = n + 1, \dots, n + p$. The functions φ_j amount to the intrinsic functions or operators of a programming language. Being able to differentiate every intrinsic function and operator allows us to differentiate arbitrary code written in a given programming language.

Applying the tangent-linear or adjoint model (see Section 2.2.4 and Section 2.2.3) leads us to the tangent-linear code generation

for $j = n + 1, \dots, n + p + m$

$$v_j^{(1)} = \sum_{i \prec j} \frac{\partial \varphi_j}{\partial v_i} \cdot v_i^{(1)} \quad .$$

or the generation of adjoint code

for $j = n + p + m, \dots, n + 1$

$$(v_{(1)i})_{i \prec j} = v_{(1)j} \cdot \left(\frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k} \right)_{k \prec j} \quad .$$

of arbitrary SAC as stated in [45]. Notice that in the SAC the incremental adjoint model is not used. This avoids the issue of nullification of the adjoints, which is explained in the next section.

Extending the MPI interface to the SAC notation would allow us to handle the MPI communication in a mathematical consistent way and apply both AD models seamlessly. The missing element is an abstraction of the MPI semantics that allows us to transform MPI calls into a SAC. This requires a mapping of all MPI functions onto intrinsic mathematical functions with inputs and outputs. Having an abstract representation of the semantics of the MPI communication would then allow us to apply the adjoint model seamlessly.

2.4.1 Nullification of the Adjoint

In a SAC, as described in the last section, every intermediate variable v_j is used once. This is generally not the case in real code as variables are allowed to be used multiple times. Here is an example written in SAC but with the variable $v0$ being used twice to store a value. Once in the statement $v0=x$ and again in statement $v0=v1$. Although the value $y = 2x = 6$ is computed, the adjoints are wrong if they are not appropriately reset to 0.

```

1 // Forward section
2 x=3;
3 v0=x;
4 v1=v0+v0;
5 v0=v1;
6 y=v0;
7 // Reverse section without nullification
8 a1_y=1 // seeding
9 a1_v0+=a1_y; // =1
10 a1_v1+=a1_v0; // =1
11 a1_v0+=(1+1)*a1_v1;
12 a1_x+=a1_v0; // =3, wrong!
13 // Reverse section with nullification
14 a1_y=0; a1_x=0; a1_v0=0 ; a1_v1=0;
15 a1_y=1; // seeding
16 a1_v0+=a1_y;
17 a1_y=0;
18 a1_v1+=a1_v0;
19 a1_v0=0;
20 a1_v0+=(1+1)*a1_v1; // =2
21 a1_v1=0;
```

```

22 a1_x+=a1_v0; // =2, correct
23 a1_v0=0

```

Listing 2.2: Multiple variable access

In Listing 2.2 two times 3 is computed. First x is set to 3 (line 2) and then assigned to $v0$ (line 3). $v1$ is computed by the sum $v0+v0$ (line 4). $v1$ is assigned to $v0$ (line 5), which is then the second time a value is assigned to $v0$. And last, $v0$ is assigned to the output variable y (line 6). In the reverse want to compute the derivative of y with respect to x according to the adjoint model (see Section 2.2.4). Therefore, $a1_y$ is set to one (seeding) in line 8. The incremental adjoint model is applied to each statement and we end up with a wrong gradient of 3 instead of 2. This is due to the missing nullification of the adjoints, in particular of $a1_v0$ before it is incremented a second time in line 11 after having been incremented in line 9. The conservative approach is to set every adjoint to 0 after its contribution is used in an incremental update. There is more on this subject in the book [45]. Moreover, due to the increment, it has to be guaranteed that every adjoint is set to 0 at the beginning of the reverse section (line 14). The required nullification amounts to an additional zero assignment in the SAC:

$$\begin{aligned}
 &\text{for } j = n + p + m, \dots, n + 1 \\
 &\quad (v_{(1)i})_{i \prec j} += v_{(1)j} \cdot \left(\frac{\partial \varphi_j(v_i)_{i \prec j}}{\partial v_k} \right)_{k \prec j}, \\
 &\quad v_{(1)j} = 0.
 \end{aligned} \tag{2.3}$$

In this work we take the conservative approach and assume that every adjoint has to be set to 0 after it has been used in an incremental update. This nullification has to take place before the next increment.

2.5 Partitioned Global Address Space

Partitioned Global Address Space (PGAS) serves as an abstraction formalism to convey MPI semantics to the SAC notation in AD. PGAS is a rather new formalism in parallel computing driven by the development of the PGAS language Unified Parallel C (UPC) in the late 90s [11]. It unites the distributed memory spaces that belong to a specific process by associating each memory location or variable with the process rank. This is usually achieved by prefixing a variable with the rank of the process it belongs to. The distributed memory space is thus modelled as a shared memory space with non uniform memory access. Each process has only access to the local variables. For a process to access a remote memory location, it has first to transfer a variable to the local address space. This is modelled through an assignment of a remote memory location to a local memory location (e.g. $1.x = 0.x$). An assignment with different memory locality on the left-hand and right-hand side has the same properties as a communication of variables.

In practice, PGAS has never gained widespread traction. It is easy to write PGAS code, however it is hard to make it perform well in a generic setting. There has to be either an offline or online pattern analysis, otherwise statements only model single sends and receives. More complex patterns like a reduction have to be detected by the system or reintroduced through workshare constructs similar to OpenMP. However, modelling distributed memory architectures using PGAS proves to be extremely powerful in a theoretical analysis. In particular it has been utilized to prove the correctness of nonblocking adjoint message passing programs [46].

```

1 ...
2 int rank=0;
3 MPI_Comm_rank(MPI_COMM_WORLD, &rank);

```

```

4 double x=(double) rank;
5 MPI_Bcast (&x,1,MPI_DOUBLE,0,MPI_COMM_WORLD);
6 ...

```

Listing 2.3: 'MPI code'

First we want to illustrate how the distributed memory in an MPI setting is transformed into PGAS. In Listing 2.3 we have an MPI code where the rank is determined using `MPI_Comm_rank`. The rank is then assigned to a double precision variable `x`. The following `MPI_Bcast` distributes the value of `x` on process 0 to all the other processes. To transform this code into PGAS, the name space of all the variables is changed. Every variable is prepended with an integer representing the rank of the process the variable is currently allocated on and followed by a separating dot `'.'`. This prefix is unique and unambiguous in all MPI supported languages (Fortran and C) since their syntax states that no variable's name is allowed to start with an integer.

```

1 ...
2 int r.rank=0;
3 MPI_Comm_rank (MPI_COMM_WORLD, &r.rank);
4 double r.x=(double) r.rank;
5 // Broadcast in PGAS here
6 ...

```

Listing 2.4: 'PGAS code for process r'

Following this straightforward search and prepend logic, the code of Listing 2.3 is then transformed into Listing 2.4, while omitting the `MPI_Bcast` for the time being. At each occurrence of the variables `x` and `rank`, they are replaced by `r.x` and `r.rank`. The drawback of this transformation to PGAS is that it models a particular instance of a parallel execution. Each process has its own instance. Fortunately, MPI is very clear on the specific behaviour of a communication. For the broadcast it distinguishes between the root `r` process and the rest of the processes; being a master/slave setup. The standard leaves little room for interpretation or nondeterminism, hence the order of the PGAS statements is very clear, but not unique. A PGAS single assignment code has to be found that is an abstraction of all the original MPI code instances and that yields the correct adjoints when it is differentiated. There are other ingredients beside the variable prefix that will be addressed in the next section.

2.6 MPI Extended SAC

```

for(int i=0;i<n;i++) i.x=0.x; for(int i=n-1;i>=0;i--) 0.x+=i.x;
(a) PGAS                                     (b) Adjoint PGAS

```

Figure 2.3: `MPI_Bcast` in PGAS

The MPI induced PGAS code described in the previous and in this section will be differentiable and allows us to apply AD and use all the logic that may be used for any sequential code. In Listing 2.4 there are single assignments where the differentiation models may be applied to. The only missing element are MPI calls like the `MPI_Bcast` which was left out in the previous section. The standard explicitly describes in detail what the input and the output of an `MPI_Bcast` should be. The root variable `r.x` is broadcast to all the other processes inside the MPI communicator (discarded for simplification purposes). For instance, due to semantical analysis, we may generate the PGAS code presented in Figure 2.3a that corresponds to an `MPI_Bcast`. By interpreting the semantics of MPI any runtime instance of an MPI

enabled code is transformable into a PGAS extended SAC. The communicated messages in MPI are reduced to assignments, while the communication pattern is imitated by control flow constructs. The link between MPI and AD is established. Figure 2.3b corresponds to the adjoint statement in PGAS of Figure 2.3a.

Every statement in the PGAS extended SAC is an assignment. A local statement only involves an assignment of variables from the same process whereas an MPI call involves at least two variables of distinct process origin. Every statement in PGAS is defined by extending the SAC as follows

$$\begin{aligned} &\text{for } j = n + 1, \dots, n + p + m \\ &\quad r_p.v_j = \varphi_j(r_l.v_i)_{i \prec j} \quad , \end{aligned}$$

where r_p and r_l are the rank of the participating processes p and $0 \leq l < \text{numprocs}$. This way we are able to map any MPI enabled SAC onto a PGAS SAC, thus allowing us to differentiate the resulting code using the tangent-linear or adjoint model. One difference between a SAC derived from a sequential program and a PGAS extended SAC derived from an MPI enabled code is the potential nondeterministic behaviour introduced in parallel code execution. The SAC for an instance of a sequential program is unique and only dependent on parameters that were set before the execution started. In parallel computation there are nondeterministic effects that yield different PGAS extended SACs. The ordering of the statements is not unique for example in case of two independent local statements on two different processes. Fortunately, MPI forces some statement order through implicit or explicit synchronization.

There are three issues that need to be addressed to make the semantic extraction of a PGAS extended SAC of MPI enabled code a generic tool:

1. Variable operations
2. Concurrency and nondeterminism
3. Variable locks

The variable operations inside communications are equivalent to standard SAC operations. So we are left with 2 and 3.

Communication is equivalent to an assignment in PGAS. However, MPI calls should be treated as intrinsic functions unrelated to an assignment. This has certain benefits for high level MPI calls or for one-sided communication. The MPI communication ought to be mapped onto mathematical functions with input, outputs and inherent semantics. The semantics may dictate some additional constraints on variable access (non-blocking, one-sided) or operate on the variables (reduction). These semantics are directly derived from the MPI standard. It is a mathematical interpretation, since the standard itself is primarily written in prose. Any MPI code that does not fulfill the semantics interpretation is not a valid MPI code and thus potentially not adjoinable.

The code in Figure 2.4 is the extended SAC of a nonblocking Send/Receive pair, while omitting the nullification of the adjoints. Data is being sent from process a to process b . The function that transfers the data is the intrinsic function *SendRecv*. It is executed on both processes with the receiving and sending ranks defined through the PGAS notation. $b.x$ is the output on the left-hand side whereas $a.x$ is the function argument and serves as the input to the *SendRecv*. Without additional information, this would correspond to a blocking send and receive. The information on the right is explained in the following two sections. This first is *order of the statements* through the constraint $i < k < j$. This states that the communication takes place anytime in between statement s_i and s_j . The second are the variable access constraints described by the MPI standard modelled through *locks* on $a.x$, $a.x_{(1)}$, $b.x$ and $b.x_{(1)}$. For a more in depth analysis of nonblocking communication please refer to Section 3.4.

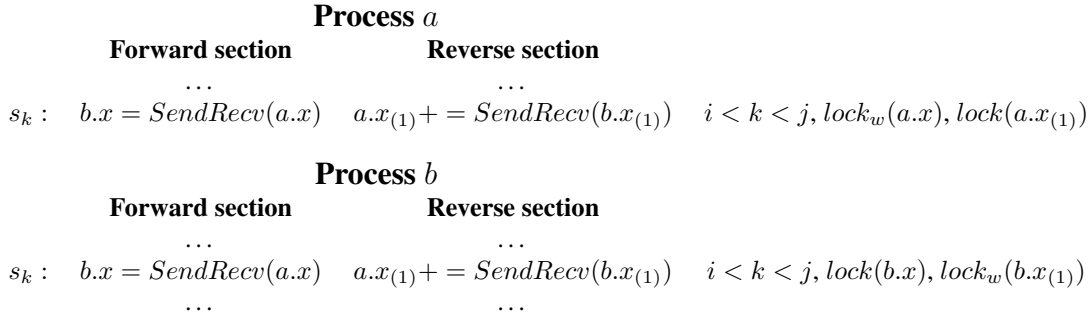


Figure 2.4: Extended SAC with nondeterminism and with access restrictions through locks

2.6.1 Order of Statements and Nondeterminism in MPI

Nondeterminism is a strong relaxation in parallel programming. There is no strict order of execution for threads in any shared memory model or between processes on distributed systems. The execution order on shared memory systems is generally determined by the operating system scheduler. For distributed systems there is no link between the execution on two nodes.

To avoid data races, write-read and read-read access conflicts in the shared memory model synchronization methods have to be introduced. The same is valid for distributed systems. MPI, as its name hints, is an interface for communicating messages. Both processes have to actively participate, with passive target one-sided communication being the only exception to this rule. With blocking point-to-point and collective communication there is no nondeterminism in MPI. Each time processes interact with each other, they are at a well defined statement in the code; the only state where the communication may take place; data races are impossible. This changes with non-blocking and one-sided communication. In both models, MPI introduces nondeterminism by making the time of communication undetermined. The only constraint is that the communication may happen at any time in between two synchronization calls. This opens the door for all the parallel coder's nightmares. Fortunately, the fact that MPI clearly defines the undetermined period with synchronization calls makes it easy to handle in a SAC. Instead of a communication taking place at a well defined statement, a range of statements is provided through the order of statements via constraints. For given statements s_i , s_j and s_k , $i < k < j$ indicates that the communication in statement s_k may take place anytime in between s_i and s_j . Any interleaving with s_k in between s_i and s_j is valid and needs to be considered. Essentially, due to distributed memory, we do not look at all possible interleavings of the statements on the processes, but we only look at all possible statements where a communication potentially takes place. This is because communications are the only statements where distributed memory locations interact. In the case of MPI, this substantially reduces the complexity of our abstraction.

2.6.2 Locks

Restrictions to program variables access are well defined in the MPI standard. They are not enforced explicitly. If such an access happens, the state of the program is considered as not *well-defined*. The nondeterministic execution then yields a nondeterministic result. In this work, we will strictly adhere to the MPI standard with regard to variable access. In practice, there may be extreme cases where a nondeterministic result may very well be reasonable and within some known error bound. However, these issues are not addressed with regard to the adjoints. A generic approach would just be too dependent on the algorithmic properties of the program. Dealing with the adjoints of nondeterministic algorithms is not

part of this work.

$$s_k : \quad b.x = \text{SendRecv}(a.x) \quad \text{lock}_w(a.x), \text{lock}(b.x)$$

Figure 2.5: Implicit access restrictions required by the MPI standard during a communication

MPI distinguishes between read and write access to variables and in particular the implicit restriction thereof. Whenever a variable is read during a communication, MPI prohibits any write access to this variable (see variable $a.x$ in Figure 2.5) and whenever a variable is written during a communication, MPI prohibits any write and read access to this variable (see variable $b.x$ in Figure 2.5). The access restrictions are denoted as *locks* where $\text{lock}(x)$ is a complete read and write restriction and $\text{lock}_w(x)$ a write restriction to a variable x .

Notice that in the SAC notation, every statement $v_j = \varphi_j(v_i)$ is transformed to the statement $v_{(1)i} \text{ += } v_{(1)j} \cdot \frac{\partial \varphi_j(v_i)}{\partial v_k}$ and the nullification $v_{(1)j} = 0$ (see Equation (2.2) and Equation (2.3)). The crucial part is that every variable v_i that is read leads to its corresponding adjoint $v_{(1)i}$ being written and read in the reverse section, whereas every variable v_j that is written in the forward section leads to its corresponding adjoint $v_{(1)j}$ being read in the reverse section. The nullification has to be dealt with separately since reading an adjoint and nullifying it cannot happen simultaneously, but only after the write lock (lock_w) has been released. Applying the same reasoning of MPI access restriction to adjoints as to the values leads to the rules in Table 2.1.

Forward section		Reverse section	
Read access	$\text{lock}_w(x)$	Write and read access (+=)	$\text{lock}(x_{(1)})$
Write access	$\text{lock}(x)$	Read access	$\text{lock}_w(x_{(1)})$

Table 2.1: Adjoint transformation rules for locks

Chapter 3

Adjoint Communication

This chapter is the main contribution of the thesis. It starts by finding an explanation for some general observations originating from the combination of AD and parallel computation. The first one is on the anticipated runtime behaviour (Section 3.1.1) of adjoining MPI, while the second one is about memory properties (Section 3.1.2) of AD on current parallel systems. We then move over to the theoretical framework of this thesis. Based on the previously defined PGAS extended SAC (Section 2.6), adjoint communication patterns in the PGAS extended SAC notation are developed. These allow us to derive the adjoint patterns for our adjoint MPI implementation in Section 4.2. A communication pattern, in this work just *pattern*, is considered to be a particular kind of MPI communication as described in the MPI standard. A pattern usually involves multiple MPI calls. For example the pattern blocking communication involves both a blocking send `MPI_Send` a blocking receive `MPI_Recv`. In line with the MPI standard the patterns are organized in three categories: point-to-point communication (Section 3.3 and Section 3.4), collective communication (Section 3.6) and one-sided communication (Section 3.8).

3.1 Observations

Although having a technique to extract adjoint communication is helpful, there is no guarantee that a corresponding implementation performs well. The first question is if and when the adjoining of MPI communication is recommended or even feasible. In Chapter 2 it has been discussed that the adjoint model of AD is memory bound because values have to be preserved in the forward section. This is the prime concern when implementing the adjoint model whereas the runtime behaviour is limited by the given adjoint model and the efficiency of the tool. Parallel computer systems have a different behaviour than sequential systems both set by theoretical limits like Amdahl's law or by physical hardware limits like for example the memory wall. This section gives an overview of what to expect from a differentiated MPI enabled code on current cluster systems.

3.1.1 Efficiency

A generic approach is taken in our runtime analysis in order to understand what the effects of applying AD are. In particular, possible wrong expectations about runtime behaviour should be emphasized. Given a parallel speedup $S(p)$ of the original passive implementation with p being the number of processes, T_1 is the runtime of the sequential code and T_p is the runtime with p processes:

$$S(p) = \frac{T_1}{T_p} \quad .$$

According to Amdahl's law, the maximum speedup is given by

$$S(p) = \frac{1}{(1 - \alpha) + \frac{\alpha}{p}} ,$$

where α is the proportion of code run in parallel. In essence, it formulates that the number of processes p only affects the parallel code, whereas the runtime of the sequential part is not affected. In this special case $T_1 = 1$ and $T_p = (1 - \alpha) + \frac{\alpha}{p}$. The general case of T_1 and T_p is

$$T_1 = t_1$$

$$\text{and } T_p = (1 - \alpha)t_1 + \frac{\alpha \cdot t_1}{p} ,$$

with t_1 being the runtime of the sequential code. Without the loss of generality, we assume that in a message passing context the only non parallel part of the code consists of the MPI communication operations. We also assume an embarrassingly parallel code with a linear speedup p with the parallel computation time being $\frac{t_{comp}}{p}$. A communicated message, even if split, is not assumed to be communicated faster with increasing p . Hence, the communication time increases with the number of processes p . The entire computation time t_{comp} is equal to the runtime with one process T_1 where no communication takes place. Thus, we have

$$T_1 = t_{comp} \text{ and}$$

$$T_p = t_{comm}(p) + \frac{t_{comp}}{p} .$$

The delay function $t_{comm}(p)$ is unknown. However, it is assumed that t_{comm} is increasing and $t_{comm}(1) = 0$. Our speedup amounts to the ratio

$$S(p) = \frac{t_{comp}}{t_{comm}(p) + \frac{t_{comp}}{p}} .$$

Assuming a “store everything approach” and neglecting any recomputation scheme through checkpointing [17], we conclude a constant slowdown of δ_a for the computation and a delay of δ_c for the communication in the adjoint code. This gives us the following speedup of the adjoint code:

$$S_{(1)}(p) = \frac{\delta_a \cdot t_{comp}}{\delta_c \cdot t_{comm}(p) + \frac{\delta_a \cdot t_{comp}}{p}} .$$

The significant factor is the efficiency which marks the evolution of the speedup with increasing processes p . Neglecting super linear speedup, an efficiency of 1 would be a perfect speedup for an arbitrary number of processes. The original efficiency $E(p) = \frac{S(p)}{p}$ is now compared with the efficiency of the adjoint code $E_{(1)}(p) = \frac{S_{(1)}(p)}{p}$.

$$\frac{E_{(1)}(p)}{E(p)} = \frac{\delta_a \cdot t_{comp} \cdot (t_{comm}(p) \cdot p + t_{comp})}{(p \cdot \delta_c \cdot t_{comm}(p) + \delta_a \cdot t_{comp}) \cdot t_{comp}}$$

Finally, we look at what happens in an exascale environment where $p \rightarrow \infty$.

$$\lim_{p \rightarrow \infty} \frac{E_{(1)}(p)}{E(p)} = \lim_{p \rightarrow \infty} \frac{\delta_a \cdot t_{comp} \cdot t_{comm}(p)}{\delta_c \cdot t_{comm}(p) \cdot t_{comp}} = \frac{\delta_a}{\delta_c}$$

For the adjoint computation the important factor is the average ratio of operations in the adjoint code with respect to the original code. For a multiplication in SAC notation where each variable is only

	Original	Adjoint forward	Adjoint reverse	$\approx \delta_c$
Point-to-point	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(2n)$	3
Broadcast	$\mathcal{O}(n \log(p))$	$\mathcal{O}(n \log(p))$	$\mathcal{O}(2n \log(p))$	3
Reduction (sum)	$\mathcal{O}(2n \log(p))$	$\mathcal{O}(2n \log(p))$	$\mathcal{O}(n \log(p))$	$\frac{3}{2}$
Reduction (prod) v1	$\mathcal{O}(2n \log(p))$	$\mathcal{O}(2n \log(p) + np)$	$\mathcal{O}(n \log(p))$	NC
Reduction (prod) v2	$\mathcal{O}(2n \log(p))$	$\mathcal{O}(3n \log(p))$	$\mathcal{O}(n \log(p))$	2
Allreduction (sum)	$\mathcal{O}(3n \log(p))$	$\mathcal{O}(3n \log(p))$	$\mathcal{O}(3n \log(p))$	2
Allreduction (prod) v1	$\mathcal{O}(3n \log(p))$	$\mathcal{O}(3n \log(p) + np)$	$\mathcal{O}(3n \log(p))$	NC
Allreduction (prod) v2	$\mathcal{O}(3n \log(p))$	$\mathcal{O}(3n \log(p))$	$\mathcal{O}(3n \log(p))$	2
Get	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(2n)$	3
Put	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(2n)$	3
Accumulate (sum)	$\mathcal{O}(2n)$	$\mathcal{O}(2n)$	$\mathcal{O}(2n)$	2
Accumulate (prod) v1	$\mathcal{O}(2n)$	NC	NC	NC
Accumulate (prod) v2	$\mathcal{O}(2n)$	$\mathcal{O}(4n)$	$\mathcal{O}(5n)$	4.5

Table 3.1: Summary of the adjoint pattern complexities and the estimated slowdown factor δ_c . Constants in big-O notation hint at the constant ratio between original and adjoint pattern. Patterns with a non constant slowdown ratio are marked as NC (non constant).

used once (no increment and no nullification in the adjoint code, see Section 2.4), it is for example 1 operation for the value ($z = x \cdot y$) and 2 for the adjoints ($x_{(1)} = y \cdot z_{(1)}$ and $y_{(1)} = x \cdot z_{(1)}$). In that case the slowdown of the adjoint computation is at best $\delta_a = 3$ (one value versus one value and two adjoint operations). Suppose that this is the general slowdown of the adjoint code then this means that if the communication slowdown is smaller than $\delta_c = 3$ there is an increase in scalability for our adjoint code. In particular, an AD tool with a rather high value of δ_a may lead to an apparent good scalability, which may be attributed mistakenly to the adjoint MPI implementation. This is a very rough estimate; specialized communication patterns may indeed yield a more complex adjoint pattern described in the coming sections.

An estimation of the communication slowdown factor δ_c is provided for each of the communication patterns in Section 3.3 (point-to-point), Section 3.4 (point-to-point nonblocking), Section 3.6 (collective) and Section 3.8 (one-sided). Notice that these pattern implementations do not rely upon an implementation of an adjoint MPI library. These benchmarks were conducted by implementing the patterns directly with MPI. No data handling besides the communication itself is measured. The tests were conducted on the *RWTH Compute Cluster* and the institute workstation *Heisenberg* (see Section 5.1). The code is available on the CD in the folder `patterns` of the adjoint MPI repository (see Appendix A). A summary of the pattern complexities is provided in Table 3.1. For the collective communications it is assumed that the network has a binary tree topology, thus leading for example to a communication complexity of $\mathcal{O}(n \log(p))$ for the reduction, with n being the message length and p the number of processes. Point-to-point communication is assumed to have a linear complexity of $\mathcal{O}(n)$. The ratio of the original runtime complexity and the adjoint pattern complexity defines the expected slowdown factor δ_c .

3.1.2 Adjoint Memory Wall

Adjoint mode AD is a memory-bound problem, because intermediate values computed in the forward section have to be stored either on a trace called *tape* (overloading) or onto a stack (source transformation) (see Section 2.2.4). For each floating point value involved in a non-linear operation in the forward section, the value or partial has to be stored in order to compute the derivatives in the reverse section. Of course only those values that are dependent on the inputs and contribute to the outputs need to be recorded.

This is called *to-be-recorded* analysis (TBR) [25]. There are three ways of tackling the high demand in memory.

The first one is to simply provide enough memory through the memory hierarchy implying that the memory is theoretically expandable ad infinitum by adding high latency memory storage (e.g. hard drives, tapes). However, this is in general not a practical solution on cluster systems, since cluster nodes often do not even have local disc access anymore. If disk access is required, the access will be provided through the network. Writing the trace on disk, by all nodes in parallel and at a reasonable speed, would just cause a network congestion.

The second option for reducing the memory footprint is *checkpointing* [21]. Checkpointing reduces memory consumption through recomputation of the values. It is a trade-off between memory consumption and runtime performance. Figure 3.1 is an example where 10 sub-steps need to be adjoined, but only one

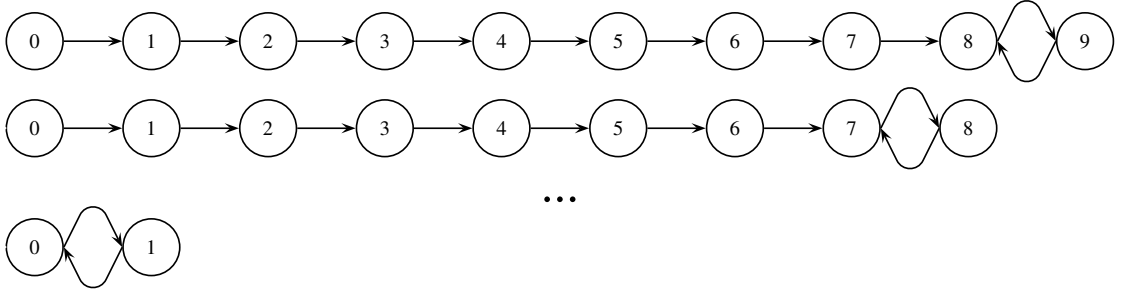


Figure 3.1: For 10 sub-steps $9+8+\dots+1$ recomputations are needed. The adjoint memory footprint increase is only that of one adjoined sub-step.

sub-step fits into memory. One solution is to restart the program and recompute $i - 1$ sub-steps in order to adjoin the i -th sub-step. We do not want to cover the technical details of checkpointing. However, it should be clear that the number of recomputations is related to the available memory of a computer system. Using this simple checkpointing scheme, there is essentially only one checkpoint at the start of the program. The number of recomputations is $c = \frac{n(n-1)}{2}$, where n is the number of sub-steps. The runtime complexity is increased by a factor of n^2 . The memory consumption of the adjoint code increases quadratically with the floating point operations (FLOPS) of the original code execution.

	nodes	Sum Peak Perf. GFlops	Sum Memory (GB)	λ
RWTH cluster	1098	161k	26352	0.16
RWTH single node	1	1k	2048	2
Oak Ridge	18688	20000k	598k	0.03

Table 3.2: Ratio of GFlops with respect to available memory on the RWTH Compute Cluster (as of 2013)

Hence, the crucial factor for the adjoint performance is the ratio $\lambda \left[\frac{\text{Bytes}}{\text{FLOPS}} \right]$ of the memory consumption with respect to the floating point operations per second (FLOPS). Unfortunately, the last years have not shown a favorable evolution to AD of this ratio. Indeed, the memory wall becomes an ever greater concern on recent cluster systems. λ is decreasing on current clusters in comparison to older clusters due to the fact that memory is more costly as compared to computation cores. To illustrate this, the specifications of the RWTH University and the Oak Ridge cluster are provided in Table 3.2. In particular, looking at the evolution of the TOP500 list¹, a rough empirical estimation hints at a logarithmic-wise decrease of

¹<http://www.top500.org/>

λ with an increase in FLOPS performance. A final conclusion cannot be drawn at this time, because there is a lack of data. Only recently, systems on the TOP500 list do provide their available memory. On older systems this information is rarely provided; memory was considered not being a performance indicator. So we are left with the adaptation of the checkpointing scheme and in particular the increase of the number of checkpoints. Less memory per node means more checkpoints per node and thus more recomputations. In a parallel environment the number of checkpoints is influenced by two factors instead of one: the problem size n and additionally the number of nodes c . The memory footprint of an adjoint program without checkpoints is directly related to the runtime complexity of the original program. A code with the runtime complexity of $\mathcal{O}(n^2)$ will have at least an adjoint memory footprint of $\mathcal{O}(n^2)$. Using our simple checkpointing scheme allows us to decrease this factor, but it will introduce recomputations that then increase the runtime complexity by an exponent of two, thus ending up with a problem in $\mathcal{O}(n^4)$ for the sequential program. The best known checkpointing scheme leads to a logarithmic growth of the memory complexity with respect to original runtime complexity [17, 20].

The third option not mentioned so far is the analytical computation of adjoints. Instead of generating code that computes adjoints discretely where the differentiation happens at the statement level, the adjoint model could just be applied to an entire mathematical subproblem. This could be a linear solver, iterations thereof or even an entire non-linear solver. This is so-called *continuous* differentiation as opposed to the aforementioned discrete method [47]. A continuously differentiated subproblem may potentially dramatically reduce the memory footprint as has been shown for example in the case of linear or nonlinear solver [47]. However, discrete adjoints may be required inside a continuous model (e.g. continuous adjoints of a nonlinear solver). Moreover, a continuous adjoint may not be equivalent to a discrete adjoint. Already at the level of iterative solvers, the difference between the two is clear [12]. Additionally, an adjoint model of the original simulation model may not exist or it may be mathematically challenging to derive. In summary, continuous adjoints are even more worth the effort in parallel computing than in a serial setting.

So what could be a general approach to the memory problem in an exascale environment? What design choices should be made when starting a new code base for a project? Whenever the runtime complexity of the original code is above the projected memory evolution with respect to the number of nodes, continuous adjoints are the only potentially scalable solution; given that a continuous version exists. With discrete adjoints, the number of recomputations would increase, thus not fulfilling the expectations of the user. However, the evolution of the projected memory per node is unknown for the future. Everything that is below this unknown threshold can and should be adjointed using discrete adjoints generated by AD tools, including the MPI communication.

3.2 Generation and Correctness of Adjoint Patterns

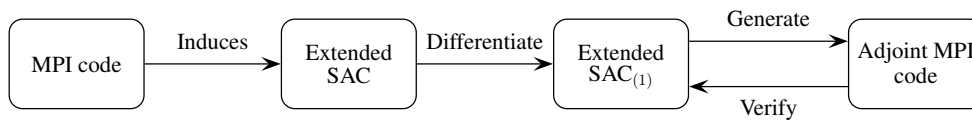


Figure 3.2: Method of adjoint MPI code generation and potential verification

Chapter 2 laid out the foundations of our adjoint communication analysis. This section describes how the adjoint communication generation in this work is derived in three distinct steps and what to expect from the correctness or the verification of an adjoint MPI implementation.

First, the MPI standard serves as the basis for the adjoint code generation. It describes in English text the requirements for an MPI interface implementation. It lacks any mathematical or computer scientific

abstraction and is mainly composed of function signatures. The text then describes the arguments of these functions and the behaviour of the communication. There is no proof of correctness or coherence. The validation or incoherences are mainly detected empirically through implementations of MPI libraries and runtime tests thereof. Hence, we introduced an abstraction that allows us to model MPI communication mathematically and convey MPI code into extended SAC notation (see Section 2.6). The first step is to extract the necessary arguments from a function's signature. The arguments may be split into three categories:

1. arguments defining the communicated buffer (e.g. send buffer, receive buffer, reduction buffer),
2. arguments specifying where the data is being sent to or received from (e.g. rank) and when (`MPI_Request`, `MPI_Fence`),
3. MPI specific arguments that are not relevant for the logic of the communication pattern itself (e.g. communicators, tags).

For the induction of the extended SAC code, we only need the first two types of arguments to map them on the communication function *SendRecv* introduced in Section 2.6. To model the destination and source of a communication the PGAS notation (see Section 2.5) is first used and applied to all variables. Each variable is prepended by the location's process rank. A receive is for example written as $b.x = \text{SendRecv}(a.x)$. The first argument defines the buffer (variable x on process a) that is being sent from process a to the variable x on process b . In the SAC notation MPI functions do only have their input buffer as an argument, whereas the output buffer is the returning value of the function. The second type of arguments defining when and to whom the data is being sent or received from are modelled entirely by the prefix of the PGAS notation. The last missing element are access restrictions described in the MPI standard that we model using locks Section 2.6.2. These four abstractions, PGAS, extended SAC and locks allow us to induce an extended SAC. This is purely an interpretation of English text with no proof of correctness. It is assumed that the English text is interpreted correctly. This step is called *induction* (see Figure 3.2).

The second step is to apply the adjoint model to the extended SAC. The *SendRecv* function is adjoint just like any regular intrinsic function. The difficult part, not present in a regular SAC are the lock statements. They are adjointed according to the rules derived in Section 2.6.2. The output of the *differentiation* step is an adjoint SAC (see Figure 3.2). The SAC abstracts the original MPI code by describing the data flow, the concurrency and the data access. Moreover the same is true for the adjoint SAC. It describes the constraints of an adjoint MPI implementation.

Hence, the third works in two ways and is either called *verification* or *generation*. The adjoint SAC gives us hints as to how an adjoint implementation might look like (generation). Moreover, it allows us to verify an adjoint MPI implementation (verification), because this adjoint code can be mapped onto an extended SAC the same way as the original code. This leads us to the downside of this method. The extended SAC with its constraints does not uniquely and seamlessly translate into an adjoint MPI implementation. It does not allow us to generate adjoint MPI code based on the original code. There has to be human input. The non uniqueness of an adjoint implementation is inherent in MPI. It is not a standard that needs to be read and interpreted by compilers, thus it is also not transformable through a grammar similarly to what has been done with adjoints of OpenMP code [16].

For the communication patterns we distinguish between blocking, nonblocking, collective and one-sided communication. The signature of the MPI functions is presented and the arguments that are relevant for the adjoining of the communication are picked. Our method of adjoining a given MPI communication pattern is then applied. The blocking communication serves as an illustrative example for all the other patterns and should be seen as part of our method description. In general, it is assumed that the reader has an in depth knowledge of MPI.

3.3 Blocking Point-to-Point Communication

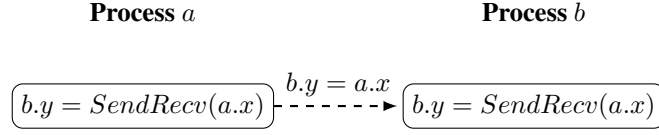


Figure 3.3: Schematics of a blocking point-to-point MPI communication using the *SendRecv* construct of the SAC notation. A variable x on process a is sent to process b and saved in the variable y amounting to the PGAS assignment $b.y = a.x$.

The foundations of adjoining single send and receives have been presented in [62]. Here it is used to illustrate the method of combining PGAS, SAC and MPI semantics introduced in Chapter 2. The proof of the adjoint pattern is more detailed and will be more condensed for the patterns in the following sections. Blocking send and receives are the most basic form of communication in MPI. They represent a data array being sent from one node to another. Each send of one process must be matched by a receive on another process and vice versa. It describes core properties of a message communication. Namely, it has a source and a destination (or sink). Both of them are specified at runtime through the send and receive MPI calls defined as follows:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
```

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int src,
int tag, MPI_Comm comm, MPI_Status *status)
```

with

- **buf**: address of send or receive buffer
- **count**: number of elements in the buffer
- **datatype**: data type of each buffer element
- **dest/src**: rank of destination or source
- **tag**: message tag
- **comm**: communicator
- **status**: (only receive) status object

The arguments are divided into the three aforementioned categories (buffer definition, remote rank and MPI specifics). **buf**, **count** and **datatype** define the data in the buffer that is being sent or received. **dest/src** defines the source and the destination of the message. **tag**, **comm** and **status** are MPI specific arguments that are of no relevance for the adjoining of the message. By calling a send or receive, the execution dives into the MPI library where the buffer **buf** is sent or received to or from a remote process. Only when this communication is finished, the execution returns from the MPI library.

Theorem 3.1. *The adjoint communication of a blocking send (MPI_Send) with buffer x is a receive (MPI_Recv) with buffer $t_{(1)}$ followed by an increment of the adjoints $x_{(1)} += t$. The adjoint communication of a blocking receive (MPI_Recv) with buffer y is a send (MPI_Send) with buffer $y_{(1)}$ followed by a nullification of the adjoints $y_{(1)} = 0$.*

Proof. Without the loss of generality, we assume that the communication buffer is elemental with one single value. As has been laid out in Section 3.2, MPI functions are treated as intrinsic functions. From a global perspective, a send/receive is one intrinsic function that is executed on two processes. The output of the function is on the receiving side, whereas the input of the function is on the sending side. Moreover the PGAS notation is used by prepending every variable with the process it belongs to. The intrinsic function corresponding to a send/receive pair may then be written as:

$$b.y = SendRecv(a.x)$$

$b.y$ is the output whereas $a.x$ is the input. Note that the $SendRecv$ is executed on both processes (see Figure 3.3). According to the semantics of the blocking communication in MPI, the communication amounts to the following operation:

$$b.y \leftarrow a.x$$

By interpreting the semantics, the $SendRecv$ function is decomposed, yielding the following PGAS code:

$$b.y = a.x$$

According to the adjoint model, we end up with the corresponding adjoint statement:

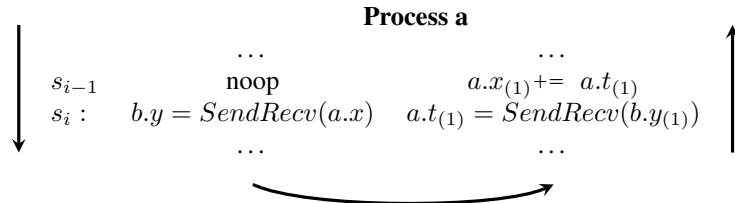
$$\begin{array}{rcl} a.x_{(1)} & += & b.y_{(1)} \\ b.y_{(1)} & = & 0 \end{array}$$

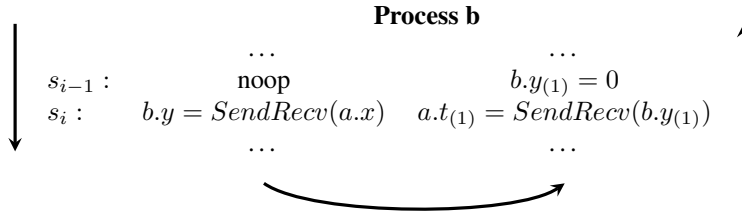
The *nullification* of adjoint variables (e.g. $b.y_{(1)} = 0$) was introduced in Section 2.4.1. Nullifying is only required if the variable b was previously used in the original code. The SAC assumes that each intermediate variable is only used once. This is of course not the case in practice where programming languages do not put such restrictions on variables. However, it is unclear whether such a restriction applies to the MPI buffers after they were adjointed using an AD tool. All the implementations in this work (see Chapter 5) use single use buffers as it assumed that the impact on the performance is negligible. Hence, nullifications are not considered in the pattern benchmarks. However, for completeness, they are included in the reverse communication patterns if an AD tool relies on correct adjoint nullifications.

The adjoint interprocess communication is modelled by an incremental assignment in the adjoint section. In the end, every interprocess communication has to be implemented in MPI. However, there exists no incremental assignment by point-to-point communication in MPI. Therefore the interprocess incremental assignment has to be split into an assignment and a local increment.

$$\begin{array}{rcl} a.t_{(1)} & = & b.y_{(1)} \\ a.x_{(1)} & += & a.t_{(1)} \\ b.y_{(1)} & = & 0 \end{array}$$

A send and receive $SendRecv$ construct is used for communication and this leads to the following single assignment code:





The adjoint communication described by this extended SAC for the sender a and the receiver b fulfills the claim of the theorem. The arrows hint at the execution order of the code. First the forward section is executed from top to bottom, followed by a bottom up execution of the reverse section. The arrows will be omitted in future extended SAC listings. $\square$

Adjoining the blocking point-to-point communication mainly involves circumventing the incremental communication not available in MPI. Besides this, it essentially consists of reversing the underlying data flow. An incremental point-to-point communication in MPI would facilitate the adjoint implementation and potentially improve efficiency.

3.3.1 Pattern Runtime

The implementation of the pattern is a straightforward implementation of a send/receive pair with the forward pattern and reverse pattern fulfilling the premisses of the forward section and reverse section as deduced in this section. As has been motivated in the proof, the nullification of the adjoint $b.y_{(1)} = 0$ is never implemented in the pattern benchmarks since it is considered to be tool specific.

```

1 void passive_pattern(double *x, int &n) {
2   if(rank==0)
3     MPI_Send(x,n,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
4   if(rank==1)
5     MPI_Recv(x,n,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
6 }
7
8 void adjoint_forward_pattern(double *x, int &n) {
9   if(rank==0)
10    MPI_Send(x,n,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
11   if(rank==1)
12    MPI_Recv(x,n,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
13 }
14
15 void adjoint_reverse_pattern(double *x, double *z, int &n) {
16   if(rank==1) {
17     MPI_Send(z,n,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
18   }
19   if(rank==0) {
20     MPI_Recv(z,n,MPI_DOUBLE,1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
21     for(int i=0;i<n;i++) x[i]+=z[i];
22   }
23 }

```

Compared to the passive pattern, the combined forward and reverse pattern amount to a doubling of the communication and an additional operation for the increment of the adjoint. Both a send and a receive have a complexity of $\mathcal{O}(n)$ with n being the message length. The increment is assumed to have

a complexity of $\mathcal{O}(n)$ too. Thus, the passive pattern has a runtime complexity of $\mathcal{O}(n)$ whereas the combined adjoint pattern has a runtime complexity of $\mathcal{O}(3n)$. Hence, a slowdown factor of $\delta_c = 3$ is to be expected. Runtime tests yield a slowdown factor δ_c ranging from 2.9 to 3.4 depending on the length of the message.

3.4 Nonblocking Point-to-Point Communication

The reversal of nonblocking communication has already been the subject of several publications [60, 62] covering the reversal of `Isend/Wait` and `Irecv/Wait` pairs. Nonblocking communication tries to address the issue of blocking communication which has to start and end within a single function call. If the communication takes a considerable amount of time, the entire program is waiting for this communication to finish. One workaround is to use a buffered send, where the entire send buffer is copied into a temporary buffer until the communication is done. This does not solve the issue with the receive and may introduce an overhead when dealing with large buffers.

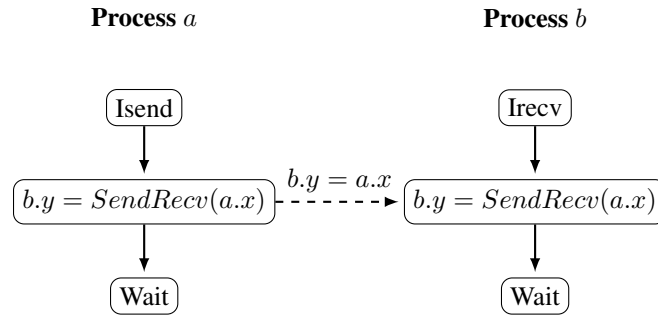


Figure 3.4: Schematics of a nonblocking MPI communication using the *SendRecv* construct of the SAC notation. A variable x on process a is sent to process b and saved in the variable y amounting to the PGAS assignment $b.y = a.x$.

The idea of nonblocking communication is to mark the start and the end of a communication by two function calls (see Figure 3.4). A communication is being defined and posted at the start, whereas the end only defines where the communication has to be accomplished in the code. The actual communication takes place somewhere in between these two calls. One way of using nonblocking communication in MPI is with `MPI_Isend`, `MPI_Irecv` and `MPI_Wait`. `MPI_Isend` and `MPI_Irecv` mark the start whereas `MPI_Wait` marks the end for both receive and send. The name `MPI_Wait` hints at a potential waiting time when a given communication is not finished when it is called. The program halts and waits for a remote process to mark the communication as done. This wait dependence introduces the potential of deadlocks due to wait dependences [28].

```

int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm, MPI_Request *request)
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int src,
int tag, MPI_Comm comm, MPI_Request *request)

```

with

- **buf**: address of send or receive buffer

- **count**: number of elements in the buffer
- **datatype**: data type of each buffer element
- **dest/src**: rank of destination or source
- **tag**: message tag
- **comm**: communicator
- **request**: communication request

The arguments are split into three groups. **buf**, **count** and **datatype** define what the shape of the communicated data is. **dest/src** and **request** are relevant for defining the remote and the period in which the communication has to take place. **tag** and **comm** are MPI specifics that do not have to be treated for the adjoint computation.

The difference between the blocking and the nonblocking point-to-point communication in terms of arguments is the **request**. If a send or receive is placed, a variable of type `MPI_Request` is committed. MPI initiates a communication and stores its status through an internal bookkeeping system using this request. If `MPI_Wait` is called later in the code and the communication is still unfinished, it becomes semantically equivalent to a blocking communication. The function only returns when the communication associated with the committed request is done. Between an `MPI_Isend/MPI_Irecv` and an `MPI_Wait` the communication is in an undefined state (neither started, nor ended) . This will be modelled with constraints on the actual communication.

Theorem 3.2. *Given a nonblocking send pair composed of a nonblocking send (`MPI_Isend`) and a wait (`MPI_Wait`) on process a and a nonblocking receive pair composed of a nonblocking receive (`MPI_Irecv`) and a wait (`MPI_Wait`) on process b . The adjoint call of a wait (`MPI_Wait`) on process b is a nonblocking send (`MPI_Isend`). The adjoint of a wait (`MPI_Wait`) on process a is a nonblocking receive (`MPI_Irecv`). The adjoint call of the nonblocking send (`MPI_Isend`) and the nonblocking receive (`MPI_Irecv`) on process a and b is a wait (`MPI_Wait`), followed by an increment of the adjoint buffer on process a and a nullification of the adjoint buffer on process b .*

Proof. Semantically, a nonblocking communication has the same signature in PGAS as the blocking communication.

$$s_i : b.y = \text{SendRecv}(a.x)$$

$b.y$ is the output whereas $a.x$ is the input. According to the semantics of the blocking communication in MPI, the communication amounts to the following operation:

$$b.y \leftarrow a.x$$

However, it is unknown when this communication is executed. It may happen anytime in between the send or receive (`MPI_Isend/MPI_Irecv`) initiation and the wait (`MPI_Wait`). Assuming the code has the following structure in extended SAC notation:

Process a		Process b	
	...		...
$s_i :$	<code>Isend()</code>	$s_i :$	<code>Irecv()</code>
	...		...
$s_k :$	$b.y = \text{SendRecv}(a.x)$	$s_k :$	$b.y = \text{SendRecv}(a.x)$
	...		...
$s_j :$	<code>Wait()</code>	$s_j :$	<code>Wait()</code>
	...		...

The *Isend/Irecv* and *Wait* are actually idle operations (noops). Their sole purpose is the constraint they convey to the *SendRecv*, namely $i < k < j$. It is guaranteed that neither $b.y$ nor $a.x$ are accessed in one of the statements in this interval. Therefore the *SendRecv* may be executed at any arbitrary position k between statement s_i and s_j . So what we are left with is a *SendRecv* with the additional locality constraints and the access constraints which state that $b.y$ and $a.x$ are not accessed in the given interval. A *lock* on a variable expresses that no read or write access is allowed by the user (see Section 2.6.2). Such a lock directly conveys to its adjoints. In the original code, MPI implicitly forbids read and write access to the receive buffer $lock(b.x)$, while no write access is allowed to the send buffer $lock_w(a.x)$.

Process a	Process b
...	...
$s_k : \quad b.y = \text{SendRecv}(a.x) \quad i < k < j, lock(b.y), lock_w(a.x)$	$s_k : \quad b.y = \text{SendRecv}(a.x)$
...	...

Applying our reversal rule for the point-to-point communication and for the locks results in the following adjoint SAC.

Process a	Process b
...	...
$s_i : \quad \text{noop} \quad a.x_{(1)} += a.t_{(1)}$	
...	...
$s_k : \quad b.y = \text{SendRecv}(a.x) \quad a.t_{(1)} = \text{SendRecv}(b.y_{(1)}) \quad i < k < j, lock_w(a.x), lock(a.x_{(1)})$	
...	...
...	...
	Process b
	$s_i : \quad \text{noop} \quad b.y_{(1)} = 0$
	...
	$s_k : \quad b.y = \text{SendRecv}(a.x) \quad a.t_{(1)} = \text{SendRecv}(b.y_{(1)}) \quad i < k < j, lock(b.y), lock_w(b.y_{(1)})$
	...

The lock on $a.x_{(1)}$ does not allow us to do the increment just before statement s_k in s_{k-1} as we do in the blocking case. The increment has to be postponed to the statement s_i which is where the *Isend* was placed. In this statement s_i , no operation was executed in forward section. By analogy, the same is true for the optional nullification of $b.y_{(1)}$. Hence, our theorem holds and describes a correct adjoint pattern for nonblocking communication. $\square$

3.4.1 Pattern Runtime

The communication pattern in itself is not different from the blocking point-to-point communication. The implementation would exactly match the one presented in Section 3.3.1 with an additional wait (`MPI_Wait`). Without context a benchmark of nonblocking communication is of no use here. Practical experience in the applications yield a slowdown factor δ_c of around 3 of single nonblocking communications. Note that this only holds if there are no `MPI_Waitall` calls described in the next section.

3.4.2 Antiwait

The method applied in this thesis only describes robust adjoint patterns that do not require additional user input. In [62] the idea of moving the reverse wait was introduced. This is in particular motivated by single nonblocking sends and receives in the forward section that are delimited by a waitall (`MPI_Waitall`). In that case, applying our logic, the reverse section will have a multitude of single `MPI_Irecv` and `MPI_Isend` calls all grouped together at the position of the waitall in the forward section. They are all matched by single waits that correspond to the nonblocking sends and receives in the forward section.

Moving the waits in the reverse section is equivalent to postponing the increment of an adjoint $x_{(1)}$ in the code. In general, this is allowed up to the statement where a value was assigned to x in the original code (e.g. $x = k$). At that point, the adjoint $x_{(1)}$ is needed to compute the adjoint $k_{(1)}$ in the reverse section. In combination with these single waits in an adjoint MPI pattern, there may be a potential performance gain by moving the waits and thus the increments. Such an optimization of moving the reverse waits and potentially recombining them into a waitall is not achievable without either user input or some data dependency analysis. This is not covered by this work.

3.4.3 Persistent Communication

Persistent communication is used in the case study of PETSc (Section 5.4). It has some particularities that need to be addressed. For the adjoint communication it actually amounts to a nonblocking communication. Persistent communication is similar in logic to the nonblocking pairs. This section is only included due to completeness with regard to the PETSc case study. The proof would exactly match that of nonblocking communication and is therefore reduced to a technical description. As stated by the MPI standard the purpose of persistent communication is:

Often a communication with the same argument list is repeatedly executed within the inner loop of a parallel computation. In such a situation, it may be possible to optimize the communication by binding the list of communication arguments to a persistent communication request once and, then, repeatedly using the request to initiate and complete messages.

Thus, an `MPI_Start` may be called several times upon the same initialization using `MPI_Send/Recv_init`. MPI predicts a potential reduction of the communication overhead due to the persistent request. The requests need to be deallocated explicitly using `MPI_Request_free` as opposed to an implicit deallocation at the `MPI_Wait` using nonblocking communication. That logic should be preserved in adjoint MPI. In particular, the multiple `MPI_Start` calls should be correctly and efficiently adjointed. We now go step by step through the reverse section for each routine and look at what has to be stored in the forward section.

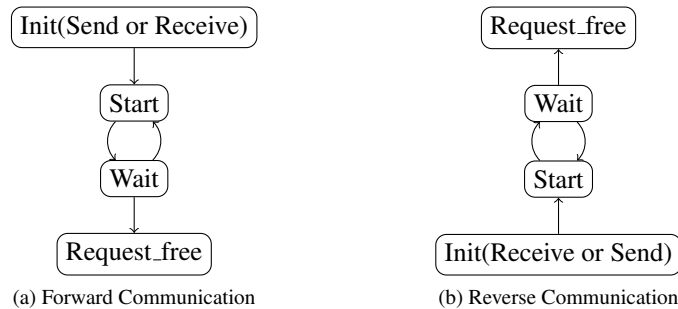


Figure 3.5: Adjoint communication of a Init, Start, Wait and Request_free

Request_free

The deallocation of the requests marks the end of the persistent communication instance. Therefore, in the adjoint section this marks the beginning of the adjoint persistent communication. The adjoint buffer is allocated here and either `MPI_Send_init` or `MPI_Recv_init` is called depending on the opcode (Send or Recv) that was saved during the forward section.

Wait

The wait marked the end of a particular communication where the buffer was either received or sent. That way the adjoint communication is started here with interchanged source and target. All this information was saved in the forward section, conveyed to the `MPI_Wait` through additional information stored in the request (see nonblocking communication) at the `MPI_Start`.

Start

`MPI_Start` marks the start of actual data communication. In the adjoint case this amounts to a Wait. It has to be made sure, that the adjoints have arrived at that point, since they may be read from now on.

Init

By symmetry, the `MPI_Send/Recv_init` marks the end of the the adjoint communication. The requests may be released with an `MPI_Request_free`.

3.5 Nondeterminism in MPI

The formalism so far ignores nondeterministic behaviour in MPI 1.1 calls. Nondeterministic patterns only exist with blocking and nonblocking point-to-point communication. There are three ways of introducing nondeterminism in MPI 1.1. The first one is by using wildcards (*) as the source in receives (`MPI_Recv` or `MPI_Irecv`). A process will then receive a message from any sending process that has issued a message to the destination that uses wildcards. The number of receiving and sending calls must always be equal during an MPI instance. However, the order of the messages may well vary, depending on which process initiates its send first. This nondeterminism in the forward section is not conveyed to the reverse section. After a message has been received, MPI allows to poll for the rank of the source using the status of type `MPI_Status`. While adjoining MPI, this instance has to be traced as if there were no wildcards. The reverse section is based on that particular instance of the forward section. Since instances of an MPI code are treated in the PGAS extended SAC, wildcards are not modeled. The same logic unfolds for the `WAIT_ANY` and `MPI_Test`. In both cases AMPI is able to extract the necessary information for tracing that particular instance of MPI communication through the status. Therefore, a nondeterministic formalism for these features is not needed.

This will change in the one-sided communication (see Section 3.8). Here, MPI does not provide any mechanism to trace the nondeterministic program execution. The reason for this is inherent to the one-sided communication. At the synchronization MPI does not return any information on the target that covers incoming or outgoing data. Data movements are totally opaque to the target. This gives the underlying network hardware freedom in optimizing communication.

3.6 Collective Communication

Collective MPI communication always involves all the processes that are included in the communicator, as opposed to point-to-point communication or one-sided communication where only two processes exchange data. In the MPI 2.2 standard every collective communication is also blocking. Every process continues its computation only after its communication is done. Note that other processes involved in the communication pattern may very well still communicate after one particular process already continues its computation. In Table 4.1 is a list of all the collective MPI calls covered in the MPI standard 2.2 (p.131) and those covered in this work. One major progress in MPI 3.0 is the introduction of non blocking collective communication.

This work should not be a repetitive reapplication of the same logic. Hence, the covered collective calls are restricted to the most important ones that pose a challenge to the adjoint logic. Following this reasoning the gather and scatter communications (`MPI_Gather`, `MPI_Gatherv`, `MPI_Scatter` and `MPI_Scatterv`) are not covered although they are supported by the adjoint MPI library.

We start with a rather simple collective: the broadcast (`MPI_Broadcast`). Although, in the forward section the data is only distributed, the reverse section introduces the incremental operation that has to be executed in addition to the communication. This is a common problem with adjoint communication patterns: the incremental adjoint model requires an additional operation. Next, the reduction (`MPI_Reduce`) is covered where we have to deal with the adjoint operations of the original operations in the forward section. The last collective operation is the Allreduce which has no master or root process.

3.6.1 Broadcast

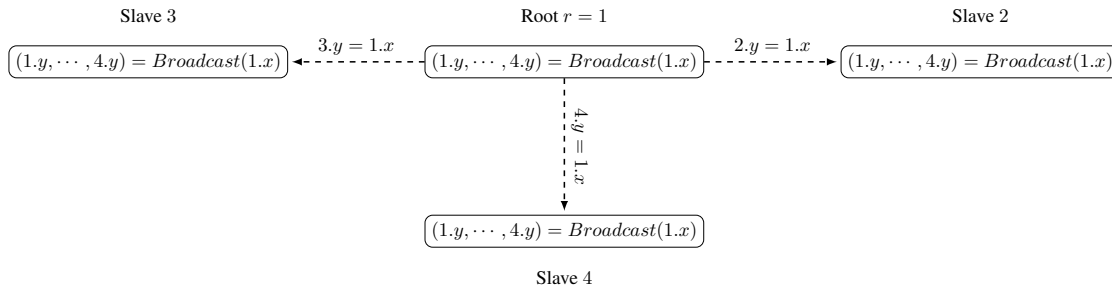


Figure 3.6: Schematics of a MPI reduction communication using the *Broadcast* construct in the SAC notation. A variable x on root $r = 1$ is broadcast to slave 2, 3 and 4 and saved in the variable y amounting to the PGAS assignment $i.y = 1.x$ where i is the rank of a slave.

The broadcast, as its name hints, serves as a broadcast where a process with rank `root` sends its data to all the other processes in the communicator `comm`. After the broadcast is done, the content of `buffer` on all processes matches the one of `buffer` on process `root`. The relevant arguments for adjoining are the buffer and the rank `rank`.

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int
root, MPI_Comm comm)
```

with

- **buffer**: address of buffer

- **count**: number of entries in buffer
- **datatype**: data type of elements of buffer
- **root**: rank of root process
- **comm**: communicator

Theorem 3.3. *The adjoint communication to a broadcast ($MPI_Broadcast$) is a reduction of the adjoints equivalent to MPI_Reduce with the sum operation MPI_SUM .*

Proof. The broadcast is modelled as a vector function where the number of outputs is equal to the n processes involved. The value $r.x$ of the root r is distributed unaltered across all the n processes. The root process r is considered as receiving its own input $r.x$ into $r.y$, with $1 \leq r \leq n$. Every process k receives its value of $k.y$ with

$$(1.y, \dots, k.y, \dots, n.y) = Broadcast(r.x) \quad .$$

The *Broadcast* function is semantically decomposable into n *SendRecv* function calls.

$$\begin{aligned} 1.y &= SendRecv(r.x) \quad , \\ &\dots \\ n.y &= SendRecv(r.x) \quad . \end{aligned}$$

The *Broadcast* is treated as a black box with no means of investigating how the broadcast was implemented eventually. The decomposition into single *SendRecv* calls yields the following PGAS code (see Section 3.3):

$$\begin{aligned} 1.y &= r.x \\ &\dots \\ n.y &= r.x \end{aligned}$$

Applying the adjoint model yields the following adjoint statements:

$$\begin{aligned} r.x_{(1)} &+= \sum_{j=1}^n j.y_{(1)} \\ j.y_{(1)} &= 0 \end{aligned}$$

Applying the adjoint model to the single *SendRecvs* yields the following extended SAC:

Process $j, j \neq r$			
	Forward section		Reverse section
$s_{i-1} :$	$\dots$		$\dots$
$s_i :$	$(1.y, \dots, k.y, \dots, n.y) = Broadcast(r.x)$		$j.y_{(1)} = 0$
	$\dots$		$r.t_{(1)} = SendRecv(j.y_{(1)})$
	$\dots$		$\dots$
Root r			
	Forward section		Reverse section
$s_{i-1} :$	$\dots$		$\dots$
	$\dots$		$r.x_{(1)} += r.y_{(1)}, r.y_{(1)} = 0$
			$\forall j \in \{1, \dots, n\}, j \neq r:$
$s_i :$	$(1.y, \dots, k.y, \dots, n.y) = Broadcast(r.x)$		$r.x_{(1)} += r.t_{(1)},$
	$\dots$		$r.t_{(1)} = SendRecv(j.y_{(1)})$
	$\dots$		$\dots$

In the reverse section the root r collects the adjoints $j.y_{(1)}$ of all the other involved processes j in a temporary buffer $r.t_{(1)}$. It then increments the local adjoint $r.x_{(1)}$ by all the incoming contributions in $r.t_{(1)}$. This pattern may be implemented as a sum reduction, although any pattern implementation that fulfills the description is a valid one (e.g. single sends and receives with explicit increment). It actually matches exactly the semantics of a reduction `MPI_Reduce` using an `MPI_SUM` operation (see Section 3.6.2). $\square$

Notice that the increment that follows from the adjoint model is being executed by MPI through the operation `MPI_SUM` and does not have to be added locally. In MPI, the broadcast value on the root r is unaltered. However, in our notation the receive buffer is separate from the send buffer on the root ($r.y$ and $r.x$), which is particularly important in the reverse section (see statement s_{i-1} on root r). How these semantics are implemented is left to the developer (e.g. using `MPI_IN_PLACE` for the reduction or not).

Pattern Runtime

The forward and reverse sections amount to the following pattern testing code.

```

1 void passive_pattern(double *sendbuf, double *recvbuf, int &n) {
2   MPI_Bcast (sendbuf, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
3 }
4
5 void adjoint_forward_pattern(double *sendbuf, double *recvbuf, int &n) {
6   MPI_Bcast (sendbuf, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
7 }
8
9 void adjoint_reverse_pattern(double *sendbuf, double *recvbuf, int &n) {
10  MPI_Reduce (sendbuf, recvbuf, n, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
11 }
```

The runtime cost of a broadcast and a reduction are heavily dependent on the hardware and the MPI implementation. If a binary tree is used as a topology for the communication, the broadcast and the reduction have a complexity of $n \log(p)$ and $2n \log(p)$, respectively. n is the communication size while p is the number of processes. This means that the amount of communicated data is doubled. However, the reduction incorporates an operation (`MPI_SUM`) on the communicated data. If the data is communicated using the binary tree, the sum operation is executed at each level of the binary tree, thus amounting to a total runtime cost of at most $2n \log(p)$. In summary, the adjoint pattern is in the same complexity class as the original pattern. However, this time the expected slowdown factor δ_c is higher than 2, depending on the CPU speed, due to the sum operation in the reduction. We have a complexity of $\mathcal{O}(n \log(p))$ for the forward passive pattern, $\mathcal{O}(n \log(p))$ for the forward pattern and $\mathcal{O}(2n \log(p))$ for the reverse pattern. Thus the expected slowdown factor δ_c is approximately 3. Different message sizes and numbers of processes (2 to 16) were tested. The slowdown ratio remained constant between 2.5 and 2.7.

3.6.2 Reduction

The MPI reduction is exceptional since it is the only MPI call besides the accumulation (see Section 3.8.4) that does not only perform a communication. It does an additional operation on the data elements of the buffer. The reasoning is that communication and computation are closely linked. When the involved operators are associative and optionally commutative, the order of execution is not necessarily fixed and may be chosen arbitrarily at runtime. In computer science operators are never really associative or commutative due to the machine precision, although this fact is neglected in this work.

A reduction in MPI may span different computational graphs on different platforms. Thus the same is true for the adjoint computation. However, there is no access to the data and communication structures

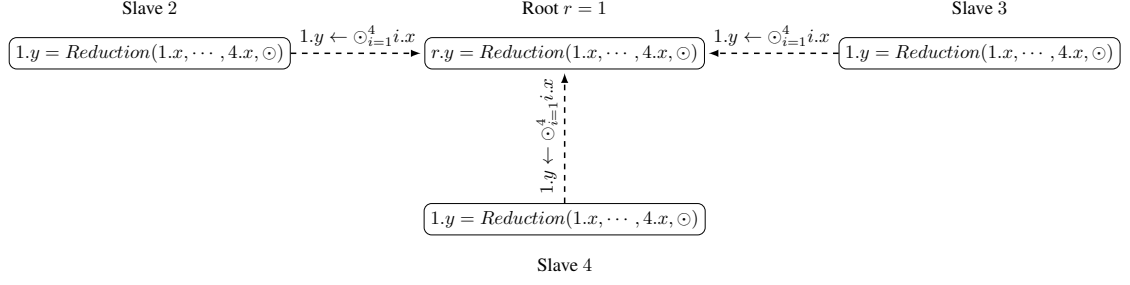


Figure 3.7: Schematics of a reduction MPI communication using the *Reduction* construct of the SAC notation. A variable x on root $r = 1$ is sent to slave 2, 3 and 4 and saved in the variable y amounting to the PGAS assignment $r.y \leftarrow \odot_{i=1}^n i.x$ where $\odot$ is one of the built-in MPI operations.

Operation $\odot$	Forward Section	Reverse Section
MPI_SUM	$y = \sum_{i=1}^n x_i$	$x_{(1)_i} += y_{(1)}$
MPI_MAX/MPI_MIN	$x_m = \max/\min(x_1, \dots, x_n)$	$x_{(1)_i} += x_{(1)_m}$
MPI_PROD	$y = \prod_{i=1}^n x_i$	$x_{(1)_i} += \prod_{j \neq i} x_j \cdot y_{(1)} = \frac{y}{x_i} \cdot y_{(1)}$

Table 3.3: Commutative MPI operations and their corresponding adjoint computation

behind a reduction through the MPI interface. To gain access to these structures, one would have to look into a specific MPI implementation. Note that these structures may even change through various versions of the same MPI implementation. As a generic approach we try to compute the adjoint operations of the basic MPI operations (see Table 3.3). The function `MPI_Reduction` has the following signature:

```
int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype
datatype, MPI_Op op, int root, MPI_Comm comm)
```

with

- **sendbuf**: address of send buffer
- **recvbuf**: address of receive buffer
- **count**: number of elements in send buffer
- **datatype**: data type of elements of send buffer
- **op**: reduction operation
- **root**: rank of root process
- **comm**: communicator

Depending on the operation `op` the SAC is different. Additionally, depending on the implementation, the reduction may just consist of the `root` process executing all the operations, or there might be a binary tree where the operations are spread as evenly as possible across all the processes. In the end it is a special

reduction tailored to the underlying cluster topology. In any case, access to the internal representation of the reduction is avoided, as it requires access and manipulation of the MPI library's source code. There is no way to dive into the substeps that lead to this result. The reduction is treated as a black box, with access to the underlying reduction operation (see Section 4.2.3 for more details on this subject). The relevant arguments in MPI are the send buffer `sendbuf`, the receive buffer `recvbuf`, the reduction operation `op` and finally the rank `root` where the result of the reduction is to be located. Therefore a reduction yields a custom intrinsic function *Reduction* that is different from the *SendRecv*:

$$r.y = \text{Reduction}(1.x, \dots, i.x, \dots, n.x, \odot) \quad ,$$

with $r, 1 \leq r \leq n$ being the root process, $1, \dots, n$ the n involved processes in the reduction and the reduction operation $\odot$. The semantics are defined as follows:

$$r.y \leftarrow \odot_{i=1}^n i.x$$

Using the chain rule we are able to derive the adjoint operation:

$$i.x_{(1)} \leftarrow i.x_{(1)} + \nabla(\odot_{i=1}^n i.x) \cdot r.y_{(1)}$$

The adjoint expression for a reduction operation depends on the operation that is being executed. We will now go through each one and write the PGAS code of the reductions of the basic floating point operations `MPI_SUM` and `MPI_PROD`.

MPI_SUM

Theorem 3.4. *The adjoint communication to a reduction (`MPI_Reduce`) with the sum operation `MPI_SUM` is an allreduction (`MPI_Allreduce`) with the sum operation `MPI_SUM`.*

Proof. In case of `MPI_SUM` the operation $\odot$ amounts to the algebraic sum $+$ applied to every $j.x$ with $1 \leq j \leq n$, where each variable is located on a different process:

$$r.y \leftarrow \sum_{j=1}^n j.x$$

The adjoint operation of a sum is again a sum in the incremental adjoint model:

$$\begin{aligned} \forall j \in \{1, \dots, n\} : j.x_{(1)} + &\leftarrow r.y_{(1)}, \\ r.y_{(1)} &= 0 \end{aligned}$$

The adjoint $r.y_{(1)}$ of the reduction's result is distributed or broadcast across all the involved processes $1, \dots, n$ and added to the local adjoints $j.x_{(1)}$.

For the SAC, the operation $+$ is irrelevant as the sum is seen as a property of the *Reduction* function. It has all the sum contributions as an argument while the sum operation $\sum$ that is executed is defined by the last argument, here $+$. All the processes call the *Reduction* function in the forward section with the same arguments. Eventually only the root has the result of $r.y$.

Process j

$$s_i : \quad r.y = \text{Reduction}(1.x, \dots, j.x, \dots, n.x, +)$$

However, for the reverse section we have to distinguish between the root and the other processes. The root is going to send its adjoint contribution $r.y_{(1)}$ to all the other processes modelled by a *SendRecv*. Therefore root has to call *SendRecv* $n - 1$ times and add the adjoint $r.y_{(1)}$ to its reduction contribution $r.x_{(1)}$. All the other processes receive the adjoint contribution $r.y_{(1)}$ and add it to the local adjoint $j.x_{(1)}$. Again, like for the point-to-point communication, the reception of the adjoints and the increment have to be split, since there is no incremental communication in MPI.

Process $j, j \neq r$			
	Forward section		Reverse section
	...		...
$s_{i-1} :$	...		$j.x_{(1)} += j.t_{(1)}$
$s_i :$	$r.y = \text{Reduction}(1.x, \dots, j.x, \dots, n.x, +)$		$j.t_{(1)} = \text{SendRecv}(r.y_{(1)})$
	...		...
Root r			
	Forward section		Reverse section
	...		...
$s_{i-1} :$	...		$r.y_{(1)} = 0$
			$r.x_{(1)} += r.y_{(1)},$
$s_i :$	$r.y = \text{Reduction}(1.x, \dots, j.x, \dots, n.x, +)$		$j.t_{(1)} = \text{SendRecv}(r.y_{(1)}),$
			$\forall j \in \{1, \dots, n\}, j \neq r$
	...		...

The solution amounts to an incremental broadcast implementation using the blocking MPI broadcast `MPI_Bcast`, which incidentally is equivalent to an MPI allreduction using an `MPI_SUM` operation (see Section 3.6.3). $\square$

Pattern Runtime The forward and reverse sections amount to the following pattern testing code.

```

1 void passive_pattern(double *sendbuf, double *recvbuf, int &n) {
2   MPI_Reduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
3 }
4
5 void adjoint_forward_pattern(double *sendbuf, double *recvbuf, int &n) {
6   MPI_Reduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
7 }
8
9 void adjoint_reverse_pattern(double *sendbuf, double *recvbuf, int &n) {
10  MPI_Bcast(sendbuf, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
11 }
```

As mentioned in Section 3.6.1, the broadcast has a complexity of $\mathcal{O}(n \log(p))$, with p being the number of processes and n the size of the message. The reduction has to additionally compute the operation which increases the runtime to two times that of the broadcast, $2n \log(p)$ (communication and operation). The complexities are as follows: $\mathcal{O}(2n \log(p))$ (communication and operation) for the passive pattern, $\mathcal{O}(2n \log(p))$ for the adjoint forward pattern and $\mathcal{O}(n \log(p))$ for the broadcast in the adjoint reverse pattern. Hence, the estimated slowdown factor δ_c of the adjoint pattern is $\frac{3}{2}$. Different message sizes and numbers of processes (2 to 16) were tested. The slowdown ratio varied between 1.4 (2 processes) and 2 (16 processes).

MPI_PROD

Theorem 3.5. *The adjoint communication to a reduction (MPI_Reduce) with the product operation MPI_PROD requires an all to all communication in the forward section. The reverse section is a broadcast followed by an incremental product operation amounting to an allreduction (MPI_Allreduce) together with a sum operation MPI_SUM .*

Proof. In case of MPI_PROD the operation $\odot$ amounts to the algebraic product $\cdot$.

$$r.y \leftarrow \prod_{j=1}^n i.x$$

The adjoint of a product $\prod$ is not as straightforward as the adjoint of a sum $\sum$ since it is a nonlinear operation. In particular, there are two ways to compute the adjoint, which are mathematically equivalent for $i.x \neq 0$. One variant is not valid for the entire domain and does not yield the same result numerically.

The first variant (Variant 1) is the general case fulfilling the theorem, resulting from the straightforward mathematical application of the adjoint model. To compute the adjoint of $j.x$ all the product contributions except $j.x$ are multiplied. This is the correct partial that then needs to be multiplied by $r.y_{(1)}$, followed by the increment of $j.x_{(1)}$.

$$j.x_{(1)}+ \leftarrow \prod_{k \neq j} k.x \cdot r.y_{(1)}, \forall j \in \{1, \dots, n\}$$

One directly notices that every process j must have access to all the remote components $k.x$ of the reduction. Using another variant (Variant 2), all this global access may be avoided and one can restrict itself primarily to local accesses. Dividing $r.y$ by $j.x$ yields the same partial needed to compute the adjoint. Obviously, this only holds if $j.x$ is not zero.

$$j.x_{(1)}+ \leftarrow \frac{r.y}{j.x} \cdot r.y_{(1)}, \forall j \in \{1, \dots, n\}, \quad j.x \neq 0$$

Numerical properties aside (see Section 3.6.2), when a value of a contribution is zero, the result is then zero. But the same is not true for the derivative and thus the partials. First the adjoint of the result $r.y_{(1)}$ is not necessarily zero. If one contribution $l.x$ is zero for one l , all the derivatives with respect to the other contributions are zero too. However, the derivative with respect to $l.x$ is nonzero. In that case, our simplification does not lead to the correct result. With two or more zero contributions it would still hold again, since then all partials are equal to zero. Since in practice this is a well defined exception that one can catch at runtime, the simplified computation is nonetheless covered because it leads to a considerably higher performance.

Similarly to the sum reduction the same *Reduction* function is only differing in the operation argument, now being $\cdot$.

Process j

$$s_i : \quad r.y = \text{Reduction}(\overset{\dots}{1.x, \dots, j.x, \dots, n.x, \cdot})$$

Two solutions of computing the adjoint will now be proposed with variant 1 being the proof of the theorem.

Variant 1 Variant 1 will lead to a considerable higher amount of communication than the passive communication. First, in order to compute the adjoint each process has to have access to all the contributions $j.x$, $1 \leq j \leq n$ of the reduction. This is modeled using single *SendRecv* functions, but this clearly amounts to an all to all communication pattern. Each process j has an array $j.x[]$ where it stores the contribution of the other processes. The contribution of process k is stored in $j.x[k]$. Note that this is the first time that the forward section already differs from the original computation with regard to the communication pattern. The communication complexity is increased by an order of magnitude. In the reverse section, the amount of communication decreases again. The root r broadcasts the adjoint $r.y_{(1)}$ to all the processes just as in the case of the sum $\sum$. This time each process has to compute the product of all the global contributions $j.x$ and multiply the result with $r.y_{(1)}$.

Process j , $j \neq r$

Forward section

$$\begin{aligned} s_i : & \quad r.y = \text{Reduction}(1.x, \dots, r.x, \dots, n.x, \cdot) \\ s_{i+1} : & \quad j.x[k] = \text{SendRecv}(k.x), k.x[j] = \text{SendRecv}(j.x), \forall k \neq j \end{aligned}$$

Reverse section

$$\begin{aligned} s_{i-1} : & \quad j.x_{(1)} += \prod_{k \neq j} j.x[k] \cdot j.t_{(1)} \\ s_i : & \quad j.t_{(1)} = \text{SendRecv}(r.y_{(1)}) \end{aligned}$$

Root r

Forward section

$$\begin{aligned} s_i : & \quad r.y = \text{Reduction}(1.x, \dots, r.x, \dots, n.x, \cdot) \\ s_{i+1} : & \quad r.x[k] = \text{SendRecv}(k.x), k.x[r] = \text{SendRecv}(r.x) \forall k \neq r \end{aligned}$$

Reverse section

$$\begin{aligned} s_{i-2} : & \quad r.y_{(1)} = 0 \\ s_{i-1} : & \quad r.x_{(1)} += \prod_{k \neq r} r.x[k] \cdot r.y_{(1)} \\ s_i : & \quad j.t_{(1)} = \text{SendRecv}(r.y_{(1)}), \forall j \in \{1, \dots, n\}, j \neq r \end{aligned}$$

□

Pattern Runtime (Variant 1) The forward and reverse sections amount to the following pattern testing code.

```

1 void passive_pattern(double *sendbuf, double *recvbuf, int &n) {
2   MPI_Reduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_PROD, 0, MPI_COMM_WORLD);
3 }
4
5 void adjoint_forward_pattern(double *sendbuf, double *recvbuf, int &n) {
6   MPI_Reduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_PROD, 0, MPI_COMM_WORLD);
7   MPI_Alltoall(sendbuf, n, MPI_DOUBLE, recvbuf, n, MPI_DOUBLE, MPI_COMM_WORLD);

```

```

8 }
9
10 void adjoint_reverse_pattern(double *sendbuf, double *recvbuf, int &n) {
11     MPI_Bcast(sendbuf, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
12 }

```

The all to all communication in the forward section elevates the total runtime complexity of $\mathcal{O}(n \log(p))$ of the reduction and allreduction to $\mathcal{O}(np)$. Hence, the slowdown factor δ_c is dependent on the number of processes. Relying on variant 1 for the adjoining of the product reduction changes the runtime behaviour dramatically. Measurements yield a slowdown factor δ_c of 3.5 for 2 processes up to 13 for 16 processes.

Variant 2 Variant 2 also changes the communication pattern of the forward section. In order to compute the partial using the simplified partial every process has to have access to result $r.y$. This is easily solved by introducing an *Allreduce* that is covered in the next section 3.6.3. *Allreduce* returns a vector which models the distribution of the result in $j.y$, $1 \leq j \leq n$ to all the processes $j.y$. In the reverse section the adjoint $r.y_{(1)}$ has to be broadcast from the root to all the other processes just as in Variant 1 and compute the partial by dividing $j.y$ by $j.x$. Using this variant, the complexity of the active code can potentially be kept at the same level as the passive code.

Process $j, j \neq r$

Forward section

$$s_i : \quad (1.y, \dots, j.y, \dots, n.x) = \overset{\dots}{Allreduction}(1.x, \dots, j.x, \dots, n.x, \cdot)$$

Reverse section

$$s_{i-1} : \quad j.x_{(1)} += \overset{\dots}{\frac{j.y}{j.x}} \cdot j.t_{(1)}, j.x \neq 0$$

$$s_i : \quad j.t_{(1)} = \overset{\dots}{SendRecv}(r.y_{(1)})$$

Root r

Forward section

$$s_i : \quad (1.y, \dots, r.y, \dots, n.x) = \overset{\dots}{Allreduction}(1.x, \dots, r.x, \dots, n.x, \cdot)$$

Reverse section

$$s_{i-2} : \quad r.y_{(1)} = 0$$

$$s_{i-1} : \quad r.x_{(1)} += \overset{\dots}{\frac{r.y}{r.x}} \cdot r.y_{(1)}$$

$$s_i : \quad j.t_{(1)} = \overset{\dots}{SendRecv}(r.y_{(1)}), \forall j \neq r$$

Variant 1 implies an all to all communication in the forward section. Hence, variant 2 is recommended in practice. See Section 3.6.2 for the numerical properties of variant 2.

Pattern Runtime (Variant 2) The forward and reverse sections amount to the following pattern testing code.

```

1 void passive_pattern(double *sendbuf, double *recvbuf, int &n) {

```

```

2  MPI_Reduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_PROD, 0, MPI_COMM_WORLD);
3  }
4
5  void adjoint_forward_pattern(double *sendbuf, double *recvbuf, int &n) {
6  MPI_Allreduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_PROD, MPI_COMM_WORLD);
7  }
8
9  void adjoint_reverse_pattern(double *sendbuf, double *recvbuf, int &n) {
10 MPI_Bcast(sendbuf, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
11 }

```

Variant 2 is much more efficient than variant 1. In particular, the complexity class of the adjoint pattern is the same as for the passive pattern. It amounts to $n \log(p)$, where p is the number of processes and n the size of the communication. Similarly to the sum reduction there is a broadcast in the reverse section. In the forward section we now have an additional allreduction for the distribution of the result. An allreduction is equivalent to a reduction (sum of communication and operation) and a broadcast (only communication), thus having a complexity of $\mathcal{O}(3n \log(p))$. The passive pattern has a complexity of $\mathcal{O}(2n \log(p))$, the adjoint forward pattern $\mathcal{O}(3n \log(p))$ and the adjoint reverse pattern $\mathcal{O}(n \log(p))$. Hence, the estimated slowdown factor δ_c is approximately 2.

Measurements with various message sizes n yield a slowdown factor δ_c of 2.2 for 2 processes up to 2.8 for 16 processes.

Numerical Stability of Product Reduction Operation MPI_PROD (Variant 2)

The product reduction operation is given by:

$$y = \prod_{i=1}^n x_i$$

The computation of the partial is

$$\frac{dy}{dx_i} = \frac{y}{x_i} \quad (3.1) \quad \text{or} \quad \frac{dy}{dx_i} = \prod_{i \neq j} x_j \quad (3.2)$$

It is known that for the multiplication $z = x \cdot y$ and division $z = \frac{x}{y}$ the relative error of z is equal to the sum of the relative errors of x and y : $\delta(z) = \delta(x) + \delta(y)$. It follows that:

$$\delta\left(\frac{y}{x_i}\right) - \delta\left(\prod_{j \neq i} x_j\right) = \left(\sum_{j=1}^n \delta(x_j) + \delta(x_i)\right) - \sum_{i \neq j} \delta(x_j) = \sum_{j \neq i} \delta(x_j) + 2 \cdot \delta(x_i) - \sum_{j \neq i} \delta(x_j) = 2 \cdot \delta(x_i)$$

For one, the amount of additional relative error introduced by using the method in Equation (3.2) instead of the straightforward computation in Equation (3.1) in the computation of $x_{(1)_i}$ is only dependent on the relative error of the value x_i . In the MPI context, this means that only the error on process i influences the error of its adjoint.

An additional disadvantage of the method in Equation (3.1) is the case where x_i is equal to 0. If another value x_j , $i \neq j$ is equal to 0, all adjoints are 0. However, if only this one value x_i is 0, all adjoints $x_{(1)_j}$ with $i \neq j$ will be 0 except $x_{(1)_i}$. In this case, the adjoint computation would need to fall back on the method of Equation (3.2). One can decide whether to use variant 1 or 2 at runtime. In order to determine the number of 0 valued arguments one process has to collect the number of 0 values for all processes and distribute the result. If this result is exactly one for one argument, variant 1 is used and variant 2 otherwise. This decision uses a reduction and a broadcast hence staying inside the communication complexity of variant 2, namely $\mathcal{O}(n \log p)$.

3.6.3 Allreduce

The MPI Allreduction is very similar to the ordinary reduction except that there is no process marked as root. The result of the reduction is available to every process involved. Hence, the signature of `MPI_Allreduce` is exactly the same as for `MPI_Reduce` except that there is no **root** argument. An `MPI_Allreduce` has the same effect as an `MPI_Reduce` followed by a broadcast of the result using `MPI_Bcast`. Of course, the added broadcast has an effect on the adjoint code.

```
int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

with

- **sendbuf**: address of send buffer
- **recvbuf**: address of receive buffer
- **count**: number of elements in send buffer
- **datatype**: data type of elements of send buffer
- **op**: reduce operation
- **comm**: communicator

Albeit that the allreduction is equivalent to a combined reduction and a broadcast, its internals are again entirely opaque to the user and is therefore treated as an intrinsic function. In addition, it is the first MPI function that has for each contribution of a process an output, since each process contributes to the reduction's result. This way, the intrinsic *Allreduction* is a vector function with an output vector of $n = \text{count}$ elements.

The relevant arguments are similar to the ordinary reduction, except that there is no need for a root process. So we end up with `sendbuf` as the input and `recvbuf` as the output, with `n=count` setting the number of inputs and outputs, while the operation is again defined by `op`. The signature of *Allreduction* is then as follows:

$$(1.y, \dots, k.y, \dots, n.y) = \text{Allreduction}(1.x, \dots, j.x, \dots, n.x, \odot).$$

with x and y as the inputs and outputs, $1, \dots, n$ the n involved processes in the reduction and $\odot$ the reduction operation. The semantics are dependent on the commutative operation $\odot$ and defined as follows:

$$(1.y, \dots, k.y, \dots, n.y) \leftarrow \odot_{i=1}^n i.x$$

Using the chain rule, the adjoint operation (see Section 3.6.2) can be derived. We will now take a look at the operations `MPI_SUM` and `MPI_PROD`.

MPI_SUM

Theorem 3.6. *The adjoint communication to an allreduction (`MPI_Allreduce`) with the operation `MPI_SUM` is an allreduction (`MPI_Alldreduce`) with the sum operation `MPI_SUM`.*

Proof. In case of `MPI_SUM` the operation $\odot$ amounts to the algebraic sum $+$ with n outputs, where all the outputs are equal to the result of the algebraic sum. Hence, the abstraction for the SAC differs from the general representation only by the operation argument marked with a '+'.

Process j

$$s_i : \quad (1.y, \dots, k.y, \dots, n.y) = \overset{\dots}{Allreduction}(1.x, \dots, j.x, \dots, n.x, +)$$

The result of the sum is mapped on the n equal results $k.x$:

$$k.y \leftarrow \sum_{j=1}^n j.x, \quad \forall k \in \{1, \dots, n\}.$$

Applying the chain rule, the adjoint operation amounts to:

$$j.x_{(1)} \leftarrow \sum_{k=1}^n k.y_{(1)}, \quad \forall j \in \{1, \dots, n\}.$$

It is evident that the adjoint operation is exactly the same as the original operation of the sum allreduction. Hence, assuming that `MPI_Allreduce` is efficient, it is the easiest MPI communication with regard to its adjoint. The *Allreduction* in the forward section becomes an *Allreduction* in the reverse section:

Process j **Forward section**

$$s_i : \quad (1.y, \dots, k.y, \dots, n.y) = \overset{\dots}{Allreduction}(1.x, \dots, j.x, \dots, n.x, +)$$

Reverse section

$$s_{i-1} : \quad j.y_{(1)} = 0$$

$$s_i : \quad (1.x_{(1)}, \dots, k.x_{(1)}, \dots, n.x_{(1)}) = \overset{\dots}{Allreduction}(1.y_{(1)}, \dots, j.y_{(1)}, \dots, n.y_{(1)}, +)$$

□

Pattern Runtime The forward and reverse sections amount to the following pattern testing code.

```

1 void passive_pattern(double *sendbuf, double *recvbuf, int &n) {
2   MPI_Allreduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
3 }
4
5 void adjoint_forward_pattern(double *sendbuf, double *recvbuf, int &n) {
6   MPI_Allreduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
7 }
8
9 void adjoint_reverse_pattern(double *sendbuf, double *recvbuf, int &n) {
10  MPI_Allreduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
11 }
```

With the runtime cost of an allreduction being $n \log(p)$, the expected slowdown factor δ_c is straightforward. With two allreductions in the adjoint pattern the estimated slowdown factor δ_c is 2. Measurements with various message sizes n show a slowdown factor of around 2 to 2.2 independent of the number of processes.

MPI_PROD

Theorem 3.7. *The adjoint communication to an allreduction ($MPI_Allreduce$) with the product operation MPI_PROD requires an all to all communication in the forward section. The reverse section is a broadcast followed by an incremental product operation.*

Proof. With the MPI product operation MPI_PROD things become more difficult in case of an allreduction. Similar to the sum, we end up with a vectorized product $\prod$ where the results $k.y$ are distributed across all the processes

$$k.y \leftarrow \prod_{j=1}^n j.x, \forall k \in \{1, \dots, n\}.$$

Again, there are two ways of adjoining the product. Based to adjoint operations of the reduction (Section 3.6.2) the two variants of the adjoint product operation are derived, while only variant 1 fulfills the theorem.

Variant 1

$$j.x_{(1)+} \leftarrow \left(\prod_{k \neq j} k.x \right) \cdot \left(\sum_{l=1}^n l.y_{(1)} \right), \forall j \in \{1, \dots, n\}$$

Variant 2

$$j.x_{(1)+} \leftarrow \frac{j.y}{j.x} \cdot \sum_{l=1}^n l.y_{(1)}, \forall j \in \{1, \dots, n\}$$

The same restrictions apply as in case of the Variant 2 reduction. One single input must not be equal to zero and the same affirmations with regard to numerical stability apply (3.6.2). Where the allreduction marked a reduced complexity for the adjoint code with regard to the operation MPI_SUM , the adjoint of an MPI_PROD operation is much harder compared to the reduction.

Variant 1 In this variant all the inputs of the *Allreduction* have again to be sent to all the processes. It is a collective all to all in statement s_{i+1} of the forward section just as in the reduction.

Process j **Forward section**

$$\begin{aligned} s_i : & \quad \dots \\ & \quad (1.y, \dots, j.y, \dots, n.y) = Allreduction(1.x, \dots, j.x, \dots, n.x, \cdot) \\ s_{i+1} : & \quad j.x[k] = SendRecv(k.x), k.x[j] = SendRecv(j.x), \forall k \neq j \end{aligned}$$

Reverse section

$$\begin{aligned} s_{i-2} : & \quad \dots \\ & \quad j.y_{(1)} = 0 \\ s_{i-1} : & \quad j.x_{(1)+} = \prod_{k \neq j} j.x[k] \cdot j.sum_{(1)} \\ s_i : & \quad (1.sum_{(1)}, \dots, j.sum_{(1)}, \dots, n.sum_{(1)}) = Allreduction(1.y_{(1)}, \dots, j.y_{(1)}, \dots, n.y_{(1)}, +) \\ & \quad \dots \end{aligned}$$

The combined adjoint contributions of all the outputs $j.y_{(1)}$ have to be summed up and send to all the processes. This again amounts to an allreduction marked in statement s_i in the reverse section. Each process then computes the local adjoint $j.x_{(1)}$ in statement s_{i-1} . $\square$

Pattern Runtime The forward and reverse sections amount to the following pattern testing code.

```

1 void passive_pattern(double *sendbuf, double *recvbuf, int &n) {
2   MPI_Allreduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_PROD, MPI_COMM_WORLD);
3 }
4
5 void adjoint_forward_pattern(double *sendbuf, double *recvbuf, int &n) {
6   MPI_Allreduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_PROD, MPI_COMM_WORLD);
7   MPI_Alltoall(sendbuf, n, MPI_DOUBLE, recvbuf, n, MPI_DOUBLE, MPI_COMM_WORLD);
8 }
9
10 void adjoint_reverse_pattern(double *sendbuf, double *recvbuf, int &n) {
11   MPI_Allreduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
12 }
```

Similar to the product reduction (Variant 1), the allreduction introduces an all to all communication in the forward section. This increases the runtime complexity from $\mathcal{O}(n \log(p))$ to $\mathcal{O}(n \cdot p)$ and the slowdown factor δ_c is not a constant anymore, while it increases with the number of processes p . It increases from 2.8 with 2 processes to 6.7 with 16 processes.

Variant 2 Variant 2 gets rid of the all to all communication and we end up with the allreduction in the forward section and a sum allreduction in the reverse section where the combined adjoint contributions $j.sum_{(1)}$ of the result adjoints $j.y_{(1)}$ are sent to every process. Each process then computes the local adjoint by dividing the result $j.y$ by the local contribution $j.x$ and then multiplying with the summed up adjoint $j.sum_{(1)}$.

Process j

Forward section

$$s_i : \quad (1.y, \dots, j.y, \dots, n.y) = \text{Allreduction}(1.x, \dots, j.x, \dots, n.x, \cdot)$$

Reverse section

$$\begin{aligned}
s_{i-2} : \quad & j.y_{(1)} = 0 \\
s_{i-1} : \quad & j.x_{(1)} += \frac{j.y}{j.x} \cdot j.sum_{(1)} \\
s_i : \quad & (1.sum_{(1)}, \dots, j.sum_{(1)}, \dots, n.sum_{(1)}) = \text{Allreduction}(1.y_{(1)}, \dots, j.y_{(1)}, \dots, n.y_{(1)}, +)
\end{aligned}$$

Variant 1 implies an all to all communication in the forward section. Hence, variant 2 is recommended in practice. See Section 3.6.2 for the numerical properties of variant 2.

Pattern Runtime The forward and reverse sections amount to the following pattern testing code.

```

1 void passive_pattern(double *sendbuf, double *recvbuf, int &n) {
2   MPI_Allreduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_PROD, MPI_COMM_WORLD);
```

```

3 }
4
5 void adjoint_forward_pattern(double *sendbuf, double *recvbuf, int &n) {
6   MPI_Allreduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_PROD, MPI_COMM_WORLD);
7 }
8
9 void adjoint_reverse_pattern(double *sendbuf, double *recvbuf, int &n) {
10  MPI_Allreduce(sendbuf, recvbuf, n, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
11 }

```

We end up with the same pattern and timings as for the sum allreduction.

3.7 Barrier

All MPI communication calls are implicitly or explicitly synchronized, the one-sided passive synchronization being the only exception to this rule. In that case `MPI_Barrier` could theoretically be used to synchronize window locks (`MPI_Win_lock`). However, we discarded an automatic solution to the passive target synchronization in AMPI (see Section 3.8.5). Otherwise, barriers may be relevant in communications using wildcards, `WAIT_ANY` or `TEST_ANY`. One therefore distinguishes between relevant and irrelevant barriers [61].

Irrelevant barriers are used for debugging purposes in order to keep the execution of processes in a well-behaved order. Another usage of irrelevant barrier is in preliminary timings, where the time is measured just after a barrier call. However, this is not encouraged as the barrier itself may greatly alter the timing itself. Sometimes a barrier is functionally irrelevant but may increase the runtime performance. By synchronizing the computational tasks, the independent communications may avoid a network congestion.

Relevant barriers in the forward section are actually becoming irrelevant in the reverse section. Every non-deterministic behaviour inserted through wildcards, `WAIT_ANY` and `TEST_ANY`, is removed in the reverse section, because one instance of the forward execution is traced. Thus, only one instance is executed in the reverse section and the relevant barriers may be ignored in AMPI.

A well-tuned MPI code with specific functionally irrelevant barriers may not relate performance-wise to its corresponding adjoint code. For AMPI one then has several options both automatic and manual for functionally irrelevant barriers:

1. execute the barriers only in the forward section,
2. trace and execute the barriers in the forward section and execute them at the same spot in the reverse section,
3. or move the barriers in the reverse section manually.

Option 1 and 2 must be supported by the AMPI library, whereas option 3 is up to manual coding by the developer.

3.8 One-Sided Communication

One-sided communication has the same notion of initiating communication as the nonblocking send/receive. There is an interval of statements delimited by some synchronization methods, defining a so-called *access epoch*. The fundamental difference is that the communication is only being initiated by one of the two involved processes. It is a point-to-point communication with only one active process, whereas the

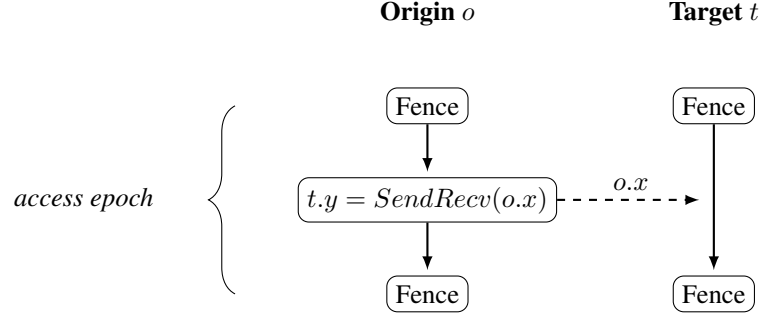


Figure 3.8: Schematics of a one-sided MPI communication using the *SendRecv* construct of the SAC notation. A variable x on the origin o is sent to the target t and saved in the variable y .

other remains passive for the message exchange. This imitates a feature of shared-memory systems with the *origin* process having direct access to the remote or *target* memory. To synchronize the target and origin, MPI introduces three synchronization methods :

1. Collective active target synchronization with fences (MPI_Win_fence)
2. Selective active target synchronization with window exposure mechanism (MPI_Win_post, MPI_Win_complete, MPI_Win_start and MPI_Win_wait)
3. Passive target synchronization with locks (MPI_Win_lock and MPI_Win_unlock)

The analysis of one-sided communication will focus on the collective target synchronization. Similar to the collective communication, this involves all the MPI processes that have created the window in the synchronization process. The selective synchronization allows users to select a subset of processes involved in the synchronization. This leads to a better scaling for example in halo exchange communication where only neighbouring processes orchestrate a synchronization. However, inside the subset of processes the same logic applies for the adjoint communication as in a collective setting. The involved processes are from a communication pattern perspective a special case of collective synchronization. The third method using strictly passive targets is a special case. In Section 3.8.5 it will be discussed why passive target synchronization is not achievable without user input.

Put (x)	$t.x = \text{Sendrecv}(o.x)$
Get (x)	$o.x = \text{Sendrecv}(t.x)$
Accumulate ($x, \text{MPI_PROD}$)	$t.y *= \text{Sendrecv}(o.x)$

Table 3.4: Equivalent SAC of MPI_Put, MPI_Get and MPI_Accumulate, with origin o and target t

Based on the nonblocking communication a one-sided MPI code is expressed as a PGAS extended single assignment code (SAC) (see Figure 3.8). The communication itself is represented by a *SendRecv*; there is a sink (output) and a source (input) with data being exchanged between the two. The difference lies entirely in the initiation of the communication. Semantically, it is only executed on the origin. Hence, this time the intrinsic function is only present in the code on the origin leaving no trace of the communication in the code of the target. The data on the target is directly manipulated by the *SendRecv* on the origin. The target's window is directly manipulated by the origin. There are three one-sided communications (see Table 3.4): MPI_Get, MPI_Put and MPI_Accumulate.

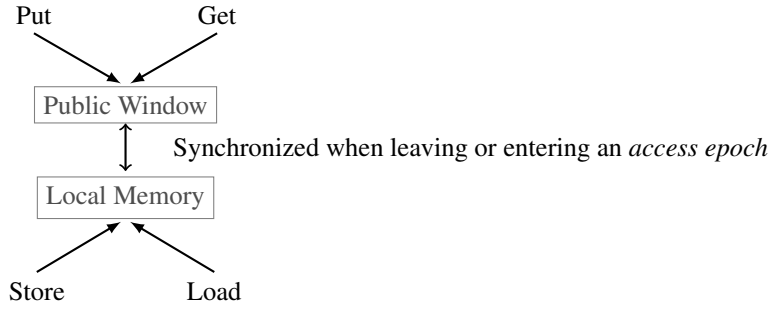


Figure 3.9: Window model in MPI - Public window is exposed to other processes through RMA operations while the local memory is still accessible through local load and stores.

The semantics in MPI distinguish between the local memory, that is accessed by local stores and loads, and the public windows that are accessed through the RMA (Remote Memory Access) operations (see Figure 3.9). Any update of the public window by RMA operations is not visible in local memory and vice versa. Thus, any update of either the local memory or the public window results in an undefined state of that memory location. Given such a state, that memory location cannot be read from or written to until a synchronization method is called. Accessing a memory location in an undefined state is called a not *well-behaved* code. No access to a memory location implies that it is neither read from nor written to. Using the lock notation introduced in Section 2.6.2 it follows that any update (equivalent to a write access) of a variable at statement s_i implies a read and write lock of

$$s_i : \text{update}(x) \implies \forall s_j, j > i : \text{lock}(x) \quad .$$

Notice that it makes no difference whether the update takes place in the public window or local memory. Using the same logic as explained in Section 2.6.2 the write access translates into a read and write access of the adjoint $x_{(1)}$ and an according $\text{lock}_w(x_{(1)})$ in all statements s_j with $j < i$:

$$\forall s_j, j > i : \text{lock}(x) \implies \forall s_j, j < i : \text{lock}_w(x_{(1)}) \quad .$$

For a read operation however, there is no implied *lock* constraint for the accessed memory location in MPI, since the public window and local memory are still in the same state. Therefore the MPI standard allows multiple load accesses on the same memory location [40, p. 363]. According to the incremental adjoint model, this results in multiple updates of the adjoint $x_{(1)}$ in the adjoint section. Each read access to a memory location in the forward section translates into an update for both the local memory and the public window in the reverse section. Synchronization then becomes a major issue. In practice, this leads to an increase in runtime. Each time a fence is hit in the reverse section, all the adjoints in the public window and the local memory need to be synchronized before being summed up. With no access to the MPI internals the summation has to be implemented sequentially after the synchronization. To circumvent this issue a restriction is introduced when dealing with adjoints and one-sided communication.

Access Restriction Axiom. *Concurrent public window and local memory access (read or write) is not allowed inside an access epoch.*

This axiom is a consequence of the MPI standard and the very nature of adjoint code. MPI disallows concurrent write/reads and write/writes exactly to address the synchronization overhead. In the reverse section of an adjoint code, every read has become a write. So if the same order of efficiency should be achieved in the adjoint code, these synchronizations have to be avoided with the very same reasoning MPI

uses. Since the generic way of resolving concurrency in the reverse section is very costly, it is advised to restructure the code in a way to avoid read/read access. The adjoint extended SAC in the `MPI_Get` and `MPI_Put` will be derived and discussed with and without the latter premiss in order to demonstrate its cost benefit. The accumulation `MPI_Accumulate` allows concurrent access to the target. It will be shown that passive target synchronization is only possible if the access restriction axiom holds. This axiom should be conveyed to users as a clear guideline for developing adjoinable one-sided MPI code.

It should be again emphasized that this work is restricted to the semantic level of MPI. An implementation of adjoint MPI that reimplements MPI could definitely benefit from direct hardware access and potentially represent a workaround to the synchronization issue. However, this would defeat our goal of portability.

The next section will introduce the MPI synchronization called fence `MPI_Win_fence`. We will then go through the three one-sided communication calls in MPI: `MPI_Get`, `MPI_Put` and `MPI_Accumulate`. The case of `MPI_Get` should serve as a more detailed introduction to the subject.

3.8.1 Fence

```
int MPI_Win_fence(int *assert, MPI_Win win)
```

with

- **assert:** program assertion
- **win:** window object

One of three ways for defining an access epoch in MPI is the collective synchronization using `MPI_Win_fence`, which limits access in between two calls. Every one-sided communication taking place in between two fences is guaranteed to happen no earlier than the first fence and no later than the second fence. Every process that created the involved window has to call the fence, even if there is no communication initiated by that process. The semantics of `MPI_Win_fence` are simple, yet contain all the necessary restrictions for the adjoint computation that would also arise with the selective target synchronization.

3.8.2 Get

```
int MPI_Get(void *origin_addr, int origin_count, MPI_Datatype
  origin_datatype, int target_rank, MPI_Aint target_disp, int
  target_count, MPI_Datatype target_datatype, MPI_Win win)
```

with

- **origin_addr** address of the buffer in which to receive the data
- **origin_count** number of entries in origin buffer (nonnegative integer)
- **origin_datatype** datatype of each entry in origin buffer (handle)
- **target_rank** rank of target (nonnegative integer)
- **target_disp** displacement from window start to the beginning of the target buffer (nonnegative integer)
- **target_count** number of entries in target buffer (nonnegative integer)

- **target_datatype** datatype of each entry in target buffer (handle)
- **win** window object used for communication (handle)

The `MPI_Get` illustrates that the one-sided communication calls have a rather lengthy signature. The target and the origin data are defined in two distinct ways. The origin buffer is defined much in the same way as in case of a point-to-point communication (buffer, number of elements and MPI datatype). Similarly, the datatype and number of elements are not treated while the origin buffer in PGAS is depicted as $o.x$ with o being the origin rank and x the origin buffer. The target buffer is defined using the collective window `win` and a given offset of `target_disp`. This is broken down to the known PGAS notation where $t.y$ describes the buffer y at target process t . In PGAS a code that uses an `MPI_Get` has the following structure:

	Origin o		Target t
$s_i :$	<code>Fence()</code>	$s_i :$	<code>Fence()</code>
	$\dots$		$\dots$
$s_k :$	<code>$o.y = \text{Get}(t.x)$</code>	$s_j :$	<code>Fence()</code>
	$\dots$		
$s_j :$	<code>Fence()</code>		

Theorem 3.8. *The adjoint operation of a get (`MPI_Get`) is an accumulation (`MPI_Accumulate`) with a sum operation (`MPI_SUM`).*

Proof. The abstraction of a get in our extended SAC notation is very similar to the nonblocking communication. If statements s_i and s_j are fences, all the statements s_k in between $i < k < j$ are inside an access epoch. The `MPI_Get` marks the beginning of a potential receiving one-sided communication much like an `MPI_Irecv` in the nonblocking case. The communication may take place at any statement s_l with $k < l < j$. During this period the MPI standard states that no access to the origin buffer $o.y$ is allowed. This leads to a lock on that buffer. On the target side, the communication may take place at any time during the access epoch. However, as the communication is one-sided, the target is not involved in this process. The fences essentially define the period $i < k < j$ where the read target buffer $t.x$ is not allowed to be written to. Additionally, if the access restriction axiom is enforced, the read access is also restricted on the target. This leads to a write lock on $t.x$.

	Origin o	
$s_i :$	<code>Fence</code>	
	$\dots$	
$s_k :$	<code>$o.y = \text{SendRecv}(t.x)$</code>	$\forall s_l : k < l < j, \text{lock}(o.y)$
	$\dots$	
$s_j :$	<code>Fence</code>	
	Target t	
$s_i :$	<code>Fence</code>	
	$\dots$	$\forall s_k : i < k < j, \text{lock}_w(t.x)$
$s_j :$	<code>Fence</code>	

The same `SendRecv` function used for nonblocking communication may also be used for the one-sided communication. Except that it is only executed by the origin. It has an apparent effect on the target, but the target remains passive; the memory access is opaque to the executed code. To derive the adjoint statements the rules for the extended SAC are applied. The `SendRecv` is adjoined according to its semantics (see Section 3.3), but there is one key difference. In the blocking and nonblocking case the

adjoint increment and the *SendRecv* had to be separated into two statements. This is not necessary here, since one-sided communication allows communication and increment of the target variable $t.x_{(1)}$ in one statement by calling `MPI_Accumulate`. Only in the context of one-sided communication, there exists an intrinsic incremental *SendRecv* function.

Origin o

	Forward section	Reverse section	
$s_{i-2} :$		Fence	
$s_{i-1} :$		$o.y_{(1)} = 0$	
$s_i :$	Fence	Fence	
	...	...	
$s_k :$	$o.y = \text{SendRecv}(t.x)$	$t.z += \text{SendRecv}(o.z)$	$\forall s_l : k < l < j, \text{lock}(o.y)$ $\forall s_l : i < l < j, \text{lock}_w(o.y_{(1)})$
	...	...	
$s_j :$	Fence	Fence	
$s_{j+1} :$		$o.z = o.y_{(1)}$	
$s_{j+2} :$		Fence	

Target t

	Forward section	Reverse section	
$s_{i-2} :$		Fence	
$s_{i-1} :$		$t.x_{(1)} += t.z$	
$s_i :$	Fence	Fence	
	...	...	$i < k < j, \text{lock}_w(t.x), \text{lock}(t.x_{(1)})$
$s_j :$	Fence	Fence	
$s_{j+1} :$		$t.z = 0$	
$s_{j+2} :$		Fence	

The lock on $o.y$ on statements $s_l : k < l < j$ translates into a lock of $o.y_{(1)}$ on statements $s_l : i < l < j$ because the incremental *SendRecv* in the reverse section ranges from s_j to s_i as opposed to the *SendRecv* in the forward section which ranges only from s_k to s_j . The reason is that the *SendRecv* in the reverse section is initiated in the fence and can only be limited by another fence. There is no way of limiting its range inside the access epoch. The nullification has to be postponed until after the access epoch, because the lock on $o.y_{(1)}$ blocks write access. Therefore another access epoch has to be introduced solely for the purpose of nullifying the adjoints on the origins O .

□

Access restriction axiom Enabling concurrent RMA and local read access introduces two additional caveats regarding read accesses on the origin o and the target t . Under the following circumstances, read accesses are allowed in the forward section inside an access epoch:

1. Read $t.x$ in statements $s_k : i < k < j$,
2. Read $o.y$ in statements $s_l : i < l < k$.

The 1st is easily solvable by introducing a separate buffer $t.z$ in the reverse section of the target t . The 2nd forces us to introduce a separate buffer $o.z$ for the publicly available window and the local memory. If there are concurrent local and remote reads of a variable, there are concurrent writes for the adjoints. By splitting the buffer into two, the incoming remote adjoints and the local increments are written into two different locations and are added up during a synchronization at the next fence in statement s_i . Note

that in that case the incremental += *SendRecv* is not necessary and may be replaced by an assignment = *SendRecv*.

Enabling concurrent reads introduces additional fences and synchronizations. In particular, the target t does not know whether $t.x_{(1)} \text{+=} t.z$ has to be executed, because it does not know whether the origin initiated a communication in the forward section. The only solution is the conservative approach of updating $t.x_{(1)}$ every time a fence is called, thus creating a huge overhead. Enacting the access restriction removes the need of temporary buffers $t.z$ and $o.z$, the additional fences and the overhead issue.

Pattern Runtime

```

1 void passive_pattern(double *x, int &n) {
2   if(rank==0) {
3     MPI_Win_fence(0,win);
4     MPI_Get(x,n,MPI_DOUBLE,1,0,n,MPI_DOUBLE, win);
5     MPI_Win_fence(0,win);
6   }
7   if(rank==1) {
8     MPI_Win_fence(0,win);
9     MPI_Win_fence(0,win);
10  }
11 }
12
13 void adjoint_forward_pattern(double *x, int &n) {
14   if(rank==0) {
15     MPI_Win_fence(0,win);
16     MPI_Get(x,n,MPI_DOUBLE,1,0,n,MPI_DOUBLE, win);
17     MPI_Win_fence(0,win);
18   }
19   if(rank==1) {
20     MPI_Win_fence(0,win);
21     MPI_Win_fence(0,win);
22   }
23 }
24
25 void adjoint_reverse_pattern(double *x, double *z, int &n) {
26   if(rank==0) {
27     MPI_Win_fence(0,win);
28     for(int i=0;i<n;i++) z[i]=x[i];
29     MPI_Win_fence(0,win);
30     MPI_Put(z,n,MPI_DOUBLE,1,0,n,MPI_DOUBLE, win);
31     MPI_Win_fence(0,win);
32     MPI_Win_fence(0,win);
33   }
34   if(rank==1) {
35     MPI_Win_fence(0,win);
36     for(int i=0;i<n;i++) x[i]+=z[i]; // adjoint increment
37     MPI_Win_fence(0,win);
38     MPI_Win_fence(0,win);
39     for(int i=0;i<n;i++) z[i]=0; // nullification
40     MPI_Win_fence(0,win);
41   }
42 }

```

For testing purposes an adjoint pattern of a Get is emulated. We violate the access restriction and introduce a temporary buffer z . The increment of x by z has a complexity of $\mathcal{O}(n)$. One-sided communications have also a linear runtime complexity of $\mathcal{O}(n)$. The synchronization in the `MPI_Win_fence` is dependent on the number of processes p . However, we were unable to measure any impact by increasing the number of processes to 64. All in all we have a linear complexity for the passive pattern and the adjoint pattern. In the passive pattern we have the Get as the only linear operation, whereas in the adjoint pattern we have a maximum of 3 linear operations: one Get, one Put and 1 write operation on the source. This amounts to an estimated slowdown factor of 3.

Measurements yield a constant slowdown factor δ_c of exactly 5.0. Keep in mind that with one-sided communication the cluster topology and the network hardware is even more important than for MPI 1.0 communications. This slowdown factor could very well vary a lot from the predicted 5.0, due to the Get being far less or far more efficient than local writes or due to the increased impact of the fences.

3.8.3 Put

```
int MPI_Put(void *origin_addr, int origin_count, MPI_Datatype
  origin_datatype, int target_rank, MPI_Aint target_disp, int
  target_count, MPI_Datatype target_datatype, MPI_Win win)
```

with

- **origin_addr** address of the origin buffer containing the data to be sent
- **origin_count** number of entries in origin buffer (nonnegative integer)
- **origin_datatype** datatype of each entry in origin buffer (handle)
- **target_rank** rank of target (nonnegative integer)
- **target_disp** displacement from window start to the beginning of the target buffer (nonnegative integer)
- **target_count** number of entries in target buffer (nonnegative integer)
- **target_datatype** datatype of each entry in target buffer (handle)
- **win** window object used for communication (handle)

An `MPI_Put` is very similar to the `MPI_Get`. Instead of pulling data from the target, the data is being pushed onto the target. The selection of relevant arguments is identical to the `MPI_Get`; they break down to the target buffer $t.y$ and the origin buffer $o.x$.

Origin o		Target t	
	...		...
$s_i :$	$Fence()$	$s_i :$	$Fence()$
	...		...
$s_k :$	$t.y = Put(o.x)$		...
	...		...
$s_j :$	$Fence()$	$s_j :$	$Fence()$
	...		...

Theorem 3.9. *The adjoint operation of a put (`MPI_Put`) is a Get (`MPI_Get`) followed by an increment in the synchronization. MPI-3.0: A fetch and accumulate is available (`MPI_Get_Accumulate`) [41].*

Proof. The abstraction in extended SAC notation is very similar to the `MPI_Get` with the $o.x$ being the input to `SendRecv` and $t.y$ the output.

	Origin o
$s_i :$	Fence
	$\dots$
$s_k :$	$t.y = \text{SendRecv}(o.x) \quad \forall s_l : k < l < j, \text{lock}_w(o.x)$
	$\dots$
$s_j :$	Fence
	Target t
$s_i :$	Fence
	$\dots$
$s_j :$	Fence $\forall s_k : i < k < j, \text{lock}(t.y)$

However, differences appear in the adjoint code due to the opposite direction of the data flow. First, the locks are switched. The restrictions MPI imposes are a write lock $\text{lock}_w(o.x)$ on the origin o together with a lock $\text{lock}(t.y)$ on the target t .

	Origin o	
	Forward section	Reverse section
$s_{i-3} :$		Fence
$s_{i-2} :$		$o.x_{(1)} += o.z$
$s_{i-1} :$		$t.y_{(1)} = \text{SendRecv}(o.0) \quad // \text{origin nullifies, target is strictly passive}$
$s_i :$	Fence	Fence
	$\dots$	$\dots$
$s_k :$	$t.y = \text{SendRecv}(o.x)$	$o.z = \text{SendRecv}(t.y_{(1)})$
		$\forall s_l : k < l < j, \text{lock}_w(o.x)$ $\forall s_k : i < k < j, \text{lock}(o.x_{(1)})$
	$\dots$	$\dots$
$s_j :$	Fence	Fence
	Target t	
	Forward section	Reverse section
$s_i :$	Fence	Fence
	$\dots$	$\dots$
$s_j :$	Fence	Fence
		$i < k < j, \text{lock}(t.y), \text{lock}_w(t.y_{(1)})$

As opposed to the `MPI_Get` there is no incremental `SendRecv` that increments the target value ($t.y_{(1)}$) onto the origin value ($o.x_{(1)}$). The MPI standard specifically disallows this (see `MPI_Accumulate` in [40]). Hence, the temporary variable z is introduced that holds the incoming adjoint before being added up to $o.x_{(1)}$ at statement s_{i-2} . This forced synchronization marks not a tremendous performance hit as for the get `MPI_Get`, because the action is triggered by the origin in the forward section and has to be executed by the active origin in the reverse section. Since the origin is active and not passive it knows which elements have to be updated. Due to the target being strictly passive, only the origin is able to set the target adjoint $t.y_{(1)}$ to 0 through an additional communication ($t.y_{(1)} = \text{SendRecv}(o.0)$). The write lock $\text{lock}_w(o.x)$ becomes a full lock $\text{lock}(o.x_{(1)})$ in the reverse section, while the lock on $t.y$ $\text{lock}(t.y)$ becomes a write lock on $t.x$ $\text{lock}_w(t.x_{(1)})$. MPI-3.0 remedies this forced synchronization by introducing `MPI_Get_Accumulate`, where the target value $t.y$ may be added to the origin value $o.x$. The nullification of the adjoint $t.y_{(1)}$ has to be done by the origin o alongside the increment of $o.x_{(1)}$ in statement s_{i-1} . $\square$

Access restriction axiom Read access in the forward section is allowed on the target in statements $s_h : i < h < k$. A temporary variable z is needed which has already been introduced in the context of the non incremental *SendRecv*. As opposed to the `MPI_Get` the performance hit is far lower, because the origin knows which elements need an adjoint update at the synchronization.

Pattern Runtime

The implementation of the adjoint Put pattern is very similar to the Get in Section 3.8.2. The only difference is that each Get is replaced by a Put and vice versa and there is only one synchronization in the adjoint pattern. So we end up with one Put in the passive pattern and 3 linear operations in the adjoint pattern: one Put, one Get and one synchronization. The expected slowdown factor δ_c is therefore 3. Measurements show a slowdown slightly above 3 of about 3.4.

3.8.4 Accumulate

```
int MPI_Accumulate(void *origin_addr, int origin_count, MPI_Datatype
    origin_datatype, int target_rank, MPI_Aint target_disp, int
    target_count, MPI_Datatype target_datatype, MPI_Op op, MPI_Win win)
```

with

- **origin_addr** Address of the buffer in which to receive the data
- **origin_count** number of entries in origin buffer (nonnegative integer)
- **origin_datatype** datatype of each entry in origin buffer (handle)
- **target_rank** rank of target (nonnegative integer)
- **target_disp** displacement from window start to the beginning of the target buffer (nonnegative integer)
- **target_count** number of entries in target buffer (nonnegative integer)
- **target_datatype** datatype of each entry in target buffer (handle)
- **op** built-in reduction operation (handle)
- **win** window object used for communication (handle)

The accumulation `MPI_Accumulate` is the one-sided equivalent to the reduction `MPI_Reduce`. The signature is very similar to the `MPI_Get` and `MPI_Put` with the only difference being the additional reduction operation argument `op`. However, the allowed operations for the accumulation are restricted to the MPI built-in operations where the numerical relevant ones, involving floating points, are `MPI_SUM` and `MPI_PROD`. Moreover, the data types too are restricted to the built-in types. Based on our experiences with `MPI_Get` and `MPI_Put` we will try to tackle the `MPI_Accumulate` while dealing first with the sum operation `MPI_SUM` and then with the product `MPI_PROD`. Similar to the reduction (Section 3.6.2) the same intrinsic function *Reduction* is used:

$$t.y = Reduction(o.x, \odot).$$

However, there are two differences. The reduction is always called one-sided by the origin o and there is only one data argument $o.x$. MPI allows multiple data access on a location on the target. During an

access epoch several origins $o \in O$, O being the set of all origins, may use the same data element $t.y$ as the target. This is the only exception in the context of one-sided communication where a memory location may be accessed more than once during an access epoch. The structure of an accumulation is equivalent to the following pseudo-code:

	Origin $o \in O$
$s_i :$	$Fence()$
	$\dots$
$s_k :$	$t.y = Reduction(o.x, \odot)$
	$\dots$
$s_j :$	$Fence()$
	Target t
$s_i :$	$Fence()$
	$\dots$
$s_j :$	$Fence()$

The one-sided semantics of an accumulation give us the following extended SAC:

	Origin $o \in O$	
s_i	Fence	Fence
	$\dots$	
$s_k :$	$t.y = Reduction(o.x, \odot)$	$\forall s_l : k < l < j, lock_w(o.x)$
	$\dots$	
s_j	Fence	Fence
	Target t	
s_i	Fence	Fence
	$\dots$	$\forall s_k : i < k < j, lock(t.y)$
s_j	Fence	Fence

The MPI semantics state that the implied reduction may take place anytime in between the statement s_k and s_j for the origin o and between s_i and s_j on the target. Thus, a lock has to be placed on $o.x$ from the s_k onwards and a full lock on the target t . The variable $o.x$ may only be written before the actual accumulation call and read at any time inside the access epoch.

MPI_SUM

Theorem 3.10. *The adjoint operation of an Accumulation ($MPI_Accumulate$) with a sum (MPI_SUM) is a Get (MPI_Get) followed by an increment in the synchronization. *MPI-3.0: A fetch and accumulate is available ($MPI_Get_Accumulate$) [41].**

Proof. The semantics of the sum operation MPI_SUM have been covered in detail when dealing with a reduction (see Section 3.6.2). Together with the operation semantics the reduction amounts to the following SAC:

Origin $o \in O$

Forward section		Reverse section	
$s_{i-2} :$		Fence	
$s_{i-1} :$		$o.x_{(1)} += o.z$	
	...	...	
$s_k :$	$t.y = Reduction(o.x, +)$	$o.z = SendRecv(t.y_{(1)})$	$\forall s_l : k < l < j, lock_w(o.x)$ $\forall s_k : i < k < j, lock(o.x_{(1)})$
	...	...	
$s_j :$	Fence	Fence	

Target t

Forward section		Reverse section	
$s_i :$	Fence	Fence	
	...	...	$i < k < j, lock(t.y), lock_w(t.y_{(1)})$
$s_j :$	Fence	Fence	

The reverse section of the accumulation with the sum operation is nearly identical to the one of an MPI_Put. Again, there is no accumulation with the origin value being incremented. Therefore, the temporary variable z is introduced that is summed up with $o.x_{(1)} += o.z$ at the synchronization. As for the MPI_Put, MPI-3.0 avoids this synchronization by introducing MPI_Get_Accumulate, where the target value $t.y$ is added to the origin value $o.x$. The only difference to put is that no nullification is needed because the initial value of $t.y$ was not overwritten. The incoming adjoint $t.y_{(1)}$ is also the correct adjoint of the initial $t.y$ because of the sum being a linear operation. $\square$

```

1 void passive_pattern(double *x, int &n) {
2   if(rank==0) {
3     MPI_Win_fence(0, win);
4     MPI_Accumulate(x, n, MPI_DOUBLE, 1, 0, n, MPI_DOUBLE, MPI_SUM, win);
5     MPI_Win_fence(0, win);
6   }
7   if(rank==1) {
8     MPI_Win_fence(0, win);
9     MPI_Win_fence(0, win);
10  }
11 }
12
13 void adjoint_forward_pattern(double *x, int &n) {
14   if(rank==0) {
15     MPI_Win_fence(0, win);
16     MPI_Accumulate(x, n, MPI_DOUBLE, 1, 0, n, MPI_DOUBLE, MPI_SUM, win);
17     MPI_Win_fence(0, win);
18   }
19   if(rank==1) {
20     MPI_Win_fence(0, win);
21     MPI_Win_fence(0, win);
22   }
23 }
24
25 void adjoint_reverse_pattern(double *x, double *z, int &n) {
26   if(rank==0) {

```

```

27     MPI_Win_fence(0, win);
28     MPI_Get(z, n, MPI_DOUBLE, 1, 0, n, MPI_DOUBLE, win);
29     MPI_Win_fence(0, win);
30     for(int i=0; i<n; i++) x[i]+=z[i];
31     MPI_Win_fence(0, win);
32 }
33 if(rank==1) {
34     MPI_Win_fence(0, win);
35     MPI_Win_fence(0, win);
36     MPI_Win_fence(0, win);
37 }
38 }

```

The implementation of the adjoint pattern is again very similar to the Put in Section 3.8.3 and to the Get in Section 3.8.2. For testing purposes an adjoint pattern of a Get is emulated. The accumulation has a linear complexity of $\mathcal{O}(n)$ where n is the length of the message. However, the constant factor is a little higher since the message is communicated and an operation is executed on the data. All in all we have one linear operation in the passive pattern and 3 in the adjoint pattern with one accumulation, one Get and one increment. Thus, the expected slowdown factor δ_c is 3. Measurements show a slowdown factor of 3.3 – 3.5.

MPI_PROD Having dealt with the collective product reduction in Section 3.6.2, the reverse section here is more complicated than the one of a sum operation. There were two variants of the adjoint product operation `MPI_PROD`. The same applies here at the one-sided communication. As a reminder the two adjoint statements of the product reduction are listed:

$$\text{The adjoint of } y = \prod_{i=1}^n x_i \quad \text{amounts to} \quad x_{(1)_i} += \prod_{j \neq i} x_j \cdot y_{(1)} = \frac{y}{x_i} \cdot y_{(1)} \quad ,$$

where n is the number of process that execute an accumulate operation on $t.y$. As for the reduction there are two choices on how to compute the adjoints.

Theorem 3.11. *The adjoint communication of an accumulation (`MPI_Accumulate`) with a product (`MPI_PROD`) is impossible (Variant 1). A shortcut solution is provided (Variant 2) amounting to an accumulation (`MPI_Accumulate`) with a replace operation (`MPI_REPLACE`).*

Proof. Variant 1 is the numerically correct adjoint according to the chain rule.

Variant 1: $x_{(1)_i} += \prod_{j \neq i} x_j \cdot y_{(1)}$

Origin $o \in O$ **Forward section**

$s_i :$ Fence
 $\dots$
 $s_k :$ $t.y = \text{Reduction}(o.x, \cdot)$ $\forall s_l : k < l < j, \text{lock}_w(o.x)$
 $s_{k+1} :$ $o.x[m] = \text{SendRecv}(m.x), m.x[o] = \text{SendRecv}(o.x)$ $\forall m \in O$
 $\dots$
 $s_j :$ Fence

Reverse section

$s_{i-2} :$ Fence
 $s_{i-1} :$ $o.x_{(1)} += o.z \cdot \prod_{m \neq o}^n o.x[m]$
 $s_i :$ Fence
 $\dots$
 $s_k :$ $o.z = \text{SendRecv}(t.y_{(1)})$ $\forall s_k : i < k < j, \text{lock}(o.x_{(1)})$
 $\dots$
 $s_j :$ Fence

Target t **Forward section Reverse section**

$s_{i-2} :$		Fence	
$s_i :$	Fence	Fence	
$s_k :$			$i < k < j, \text{lock}(t.y), \text{lock}_w(t.y_{(1)})$
$s_j :$	Fence	Fence	

Based on the adjoint product reduction (Section 3.6.2), each process o has to pull the values $m.x$, for all $m \in O$. However, this implies that each process o knows the entire set O , which is not the case. Up until now there is no reasonable solution to this problem. Supposing that there is some way to know m , we are able to compute the adjoint in s_{i-1} of the reverse section by pulling the adjoint first from the target into a temporary variable $o.z$ in s_k . The next problem is that all the origins have to compute the adjoint $t.y_{(1)}$ of the initial value of $t.y$. Again the computation thereof requires the knowledge about the shape of the set O . Hence, we conclude that this variant is not solvable with one-sided communication and that the adjoint product accumulation is thus not self-enclosed in one-sided MPI.

Variant 2: $x_{(1)_i} += \frac{y}{x_i} \cdot y_{(1)}$

So we end up with the only feasible solution being the shortcut variant, which has some stability issues discussed in Section 3.6.2. Here it is avoided to pull all the values from all the processes, because only the local value $o.y$ is needed. But again, every process o has no knowledge about the number of processes in the set O .

Origins $o \in O$ **Forward section**

$s_{i-2} :$ Fence
 $o.y = \text{SendRecv}(t.y)$ // get value of $t.y$ used to compute the adjoint $t.y_{(1)}$
 $s_i :$ Fence
 $\dots$
 $s_k :$ $t.y = \text{Reduction}(o.x, \cdot)$ $i < k < j, \text{lock}_w(o.x)$
 $\dots$
 $s_j :$ Fence
 $o.res = \text{SendRecv}(t.y)$
 $s_{j+2} :$ Fence

Reverse section

$s_{i-3} :$ Fence
 $t.y_{(1)} = \text{Reduction}\left(\frac{o.res}{o.y} \cdot o.z, \text{replace}\right)$
 $o.x_{(1)} = \frac{o.res}{o.x} \cdot o.z$
 $s_i :$ Fence
 $\dots$
 $s_k :$ $o.z = \text{SendRecv}(t.y_{(1)})$ $i < k < j, \text{lock}(o.x)$
 $\dots$
 $s_{j+2} :$ Fence

On the origin only the final value of $t.y$ is needed. However, this is only equal to the final value if the following fence is done. Therefore, the result is pulled into $o.res$ in s_{j+1} of an additionally created access epoch. On process o the initial value of $t.y$ is also pulled, because some process other than t has to compute the adjoint. Remember that process t is completely passive inside the access epoch. Since there is no notion of the processes in the set O , they cannot agree on which one has to compute adjoints; all processes in O have to compute the adjoint. Who writes the adjoint back to t in s_{i-1} ? Writing to one location concurrently is not allowed according to the standard. Fortunately, there must have been similar problems, where the designation of a task to a specific process was impossible. MPI allows to call accumulate with the operation `MPI_REPLACE`. This operation does not exist for the collective reduction. It essentially replaces the value on the target t and it is undetermined which process's value prevails. But since every process computes the exact same adjoint, this makes no difference here. The program keeps being deterministic.

Target t **Forward section Reverse section**

$s_{i-1} :$	Fence	Fence	
$s_i :$	Fence	Fence	
$s_k :$	$\dots$	$\dots$	$i < k < j, \text{lock}(t.y), \text{lock}_w(t.y_{(1)})$
$s_j :$	Fence	Fence	
$s_{j+1} :$	Fence		

Of course, the target remains passive in the forward section as well as in the reverse section. Only the additional synchronizations have to be inserted. $\square$

```

1 void passive_pattern(double *x, double *y, int &n) {
2   if(rank==0) {

```

```

3     MPI_Win_fence(0, win);
4     MPI_Accumulate(x, n, MPI_DOUBLE, 1, 0, n, MPI_DOUBLE, MPI_PROD, win);
5     MPI_Win_fence(0, win);
6 }
7 if(rank==1) {
8     MPI_Win_fence(0, win);
9     MPI_Win_fence(0, win);
10 }
11 }
12
13 void adjoint_forward_pattern(double *x, double *y, double *res, int &n) {
14     if(rank==0) {
15         MPI_Win_fence(0, win);
16         MPI_Get(y, n, MPI_DOUBLE, 1, 0, n, MPI_DOUBLE, win);
17         MPI_Win_fence(0, win);
18         MPI_Accumulate(x, n, MPI_DOUBLE, 1, 0, n, MPI_DOUBLE, MPI_PROD, win);
19         MPI_Win_fence(0, win);
20         MPI_Get(res, n, MPI_DOUBLE, 1, 0, n, MPI_DOUBLE, win);
21         MPI_Win_fence(0, win);
22     }
23     if(rank==1) {
24         MPI_Win_fence(0, win);
25         MPI_Win_fence(0, win);
26         MPI_Win_fence(0, win);
27         MPI_Win_fence(0, win);
28     }
29 }
30
31 void adjoint_reverse_pattern(double *x, double *y, double *z, int &n) {
32     if(rank==0) {
33         MPI_Win_fence(0, win);
34         MPI_Get(z, n, MPI_DOUBLE, 1, 0, n, MPI_DOUBLE, win);
35         MPI_Win_fence(0, win);
36         for(int i=0; i<n; i++) x[i]=(z[i]/x[i])*z[i];
37         MPI_Accumulate(x, n, MPI_DOUBLE, 1, 0, n, MPI_DOUBLE, MPI_REPLACE, win);
38         MPI_Win_fence(0, win);
39     }
40     if(rank==1) {
41         MPI_Win_fence(0, win);
42         MPI_Win_fence(0, win);
43         MPI_Win_fence(0, win);
44     }
45 }

```

The implementation of the adjoint pattern is again very similar to the put in Section 3.8.3 and to the get in Section 3.8.2.

In the passive pattern we have the accumulation as the only operation with linear complexity $\mathcal{O}(n)$. In the adjoint pattern there are two get and one accumulation in the forward section. In the reverse section there is one get, one accumulation and one update of the adjoint. All in all, there are 6 operations with linear complexity as opposed to 1 in the passive pattern. The expected slowdown δ_c should be around 6. In practice, measurements showed it to be around 7.5 – 8.8.

3.8.5 Passive Target Synchronization

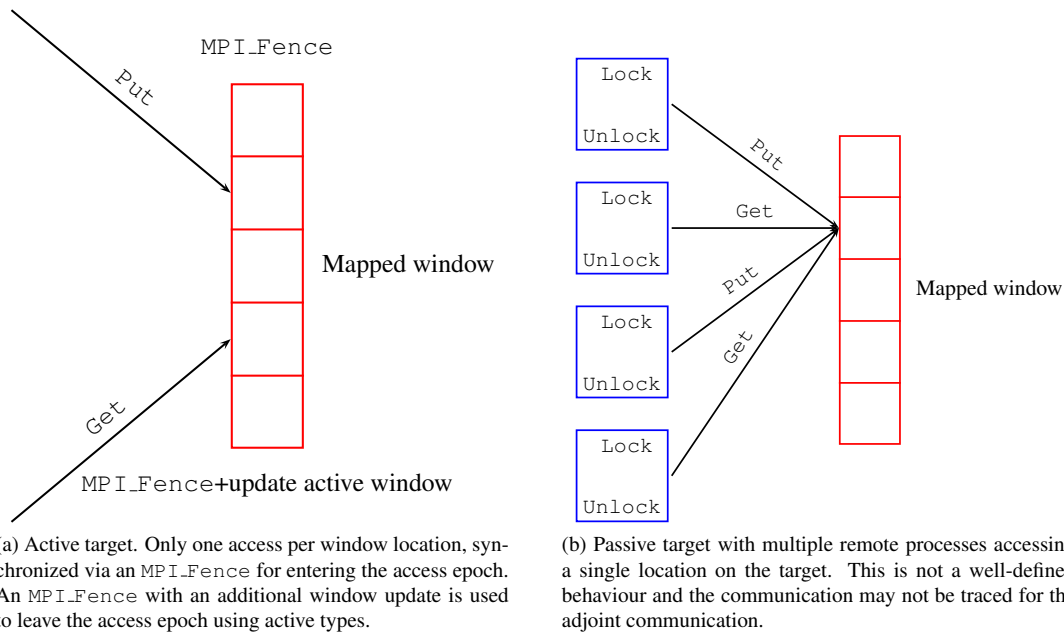


Figure 3.10: Active versus passive target

The passive target access introduces the notion of exposing a window to other origin processes. These origin processes then orchestrate the one-sided access through so called *locks* implemented by `MPI_Win_lock` and `MPI_Win_unlock`. The target's window is locked and released by remote processes. Hence, passive target communication allows a memory location to be write-read and write-write accessed more than once during an exposed period (see Figure 3.10). However, neither the target nor the origin are able to trace these data accesses, let alone whether an element was read or written at all. Such a code has no well-defined outcome and therefore it is not treated as generically adjoinable. A deeper algorithmic insight of the underlying numerical problem is required. The code has a similar structure to:

Target	Origin
	1 <code>MPI_Win_create(win)</code>
1 <code>MPI_Win_create(win)</code>	2 <code>MPI_Win_lock(win)</code>
2 . . .	3 . . .
3 <code>MPI_Win_free(win)</code>	4 <code>MPI_Win_unlock(win)</code>
	5 <code>MPI_Win_free(win)</code>

On the target side there are no means of tracing the order of lock accesses. However, enforcing the access restriction axiom changes the situation dramatically. Disallowing concurrent local and RMA reads on the target have now only to be asserted between a `lock` and `unlock` access on the target side. MPI has no built-in support for such an assertion, so the burden is entirely on the user side to implement this. If this is guaranteed to be the case, the passive target is handled exactly the same way as the active target illustrated in Section 3.8.2 and Section 3.8.3. The access epochs delimited by `Fence` on the origins are changed to `Lock` and `Unlock` periods. The target side is left unchanged. In summary, adjoints of passive target MPI code is only possible if the user enforces the access restriction axiom.

3.9 Guidelines for an Adjoinable MPI Code

This section summarizes this chapter by extracting relevant practical guidelines for developers of numerical simulation codes. The guidelines are listed in order of priority as follows:

- Adjoint memory wall
- Two-sided communication (MPI standard 1.0)
- One-sided communication (MPI standard 2.0)

First, the issue with decreasing memory availability on single cluster nodes has been discussed in Section 3.1.2. Before implementing code one should decide whether an adjoint code should run in parallel. Adjoint code generated by an AD tool may perform well using single processor machines, while scaling very badly when run in parallel. If the adjoint memory wall risks to become an issue, one should look into continuous adjoints as discrete adjoints become infeasible.

Second, if the adjoint memory wall is not an issue, one should start writing MPI code using two-sided communication. Two-sided communication has been proven to be robustly adjoinable (see Chapter 5). Only the product reduction introduces a challenge by requiring a decision at runtime on how it should be adjoint (see Section 3.6.2).

Third, given a use case for one-sided communication in the original code, one should seriously reconsider implementing one-sided MPI if adjoint code thereof should be generated by an AD tool. One-sided MPI code is not generically adjoinable. In order to keep one-sided communication generically adjoinable and expect reasonable performance, one has to enforce the access restriction axiom (see Section 3.8). It introduces a constraint on the original code by restricting concurrent read and write access in the public window and the local memory inside an access epoch. Moreover, communications like for example the accumulate with a product operation require serious workarounds in order to generate correct adjoints (Section 3.8.4), where the original communication pattern is altered considerably. This may have drawbacks performance-wise. Additionally, passive target synchronization should only be used with manual user input. It is not possible to adjoint it generically. An in depth understanding of AD should be expected for developing adjoinable passive target code, thus reducing the maintainability of the original code. (Section 3.8.5).

Chapter 4

Adjoint MPI Library

There are two generic adjoint MPI libraries at the moment. The first one was developed by the author from 2010 to 2014 for prototyping purposes called “Adjoint MPI library” or simply AMPI. It was eventually integrated into several in house AD tools that will all be described in Chapter 5 (`dco/c++` in Section 5.4, `dcc` in Section 4.3.2 and CompAD in Section 5.3.3). The other library started in early 2013 as a joint effort between the Argonne National Laboratory (US), INRIA (France) and the STCE at the RWTH Aachen University (Germany) with the aim to develop a common AMPI library called “Adjoinable MPI”. It has more control over the internal structure of the library with for example support of zero-copy (see Section 2.3.3) for tools with contiguous memory layout. From the ground up it was designed to work with ADOL-C, `dco/c++`, and Tapenade. However, it breaks the MPI function signatures in favor of more granular performance tweaks and settings for the adjoint communication.

In this chapter the development of the adjoint MPI library (AMPI) based on the results of the previous chapter will be discussed. One-sided communication is left out for the time being, since there was no real use case and no hardware with specific support that achieved any measurable runtime benefits. A first implementation based on the results of this work has been made by the author in the Adjoinable MPI library and is available in the repository (see Section A.2).

The adjoint MPI library is a straightforward implementation of the patterns derived in Chapter 3. However, some design choices had to be made and those implementation specifics will be discussed in this chapter. First, the usability requirements are elaborated in Section 4.1. The library should be portable, generic, efficient and self-contained. The portable data layout design is illustrated in 4.2.1. In Section 4.2.2 the nonblocking communication handling is presented. A first try of tracing the reduction operations is illustrated in Section 4.2.3. And last the implementation of higher order adjoint communication is shown in Section 4.3.

4.1 Usability

Usability is about how the user perceives the developed implementation. Actually, there are two users with regard to the AMPI library. The first one is the AD tool developer and the second one is the end user who applies AD to some code using an AD tool. There are four design requirements that have an impact on either the AD tool developer or the end user.

Portability (tool developer, end user) The MPI standard itself is the de facto standard for parallel programming. Although its implementations may be highly specialized and hardware specific, there is hardly any cluster with no MPI library available. It is clear that the same requirement applies to the adjoint

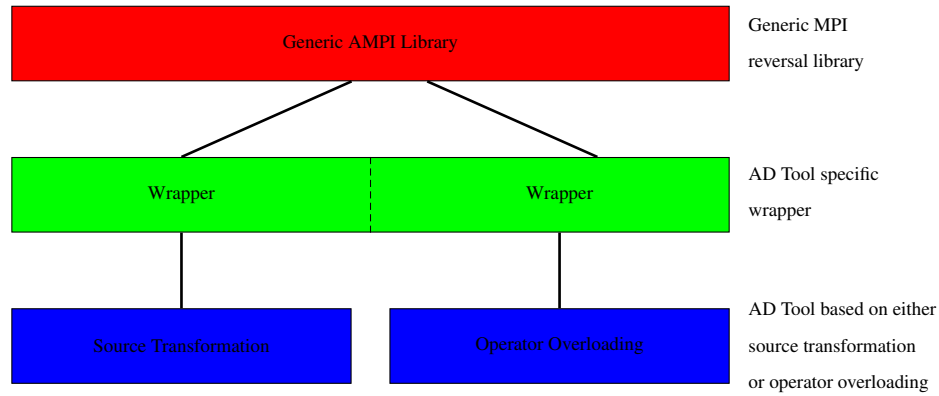


Figure 4.1: Coupling the AMPI library with an arbitrary AD tool. The wrapper has to be implemented jointly by the AD tool developers and AMPI developers.

MPI library. Any system that supports MPI should be able to run code that relies on the adjoint MPI library. The MPI standard itself is defined as a C or Fortran interface. C is chosen as the implementation language for our library. Moreover, it is clear that any system that supports MPI should also support the AMPI library. At the communication level AMPI will then be linked against this particular MPI library. Finally, minimum requirements for running adjoint MPI code is a valid ANSI C compiler and an MPI library.

Generic (tool developer) The adjoint MPI library should not be bound to any particular AD tool. The interface of the library should be very generic, as small as possible and without any tool specific code. The only tool specific code will reside in the tool specific wrapper (see Figure 4.1). Inside the wrapper, any internal adjoint MPI logic and structure should be opaque and hidden to the user.

Efficiency (end user) Chapter 3 derived the structure and constraints that any adjoint MPI code has to fulfill. However, the implementation of these constraints may potentially have a huge effect on the performance of the adjoint code. Up until today, there is no formal way for deriving the most efficient code with regard to MPI, let alone for adjoint code. In this chapter implementations will be proposed that fulfill the constraints of Chapter 3. As the efficiency of MPI implementations tend to be very hardware specific it is hard to reach a general consensus on how the most efficient MPI code should be. Hence, we propose the most generic implementations that may nonetheless yield bad performance on a particular hardware. However, in all the use case validations in Chapter 5 no such degradation of performance could be observed.

Self-contained (tool developer, end user) The adjoint MPI library should be closed within MPI. One reason is the aforementioned portability. Increasing the number of dependencies would only decrease the portability. Moreover, the library should still be usable in the years to follow. By only relying on the MPI standard, we adhere to one of the oldest parallelization standards in a research field that is moving at a very high pace.

Besides our requirements, MPI itself puts forward some restrictions. First, MPI is inherently a runtime library. Most execution properties are determined at runtime like for example a process' rank which determines the data flow of a parallel run. Differentiating MPI using source transformation is therefore only achievable by going to great lengths in order to parse the source code of an MPI implementation.

MPI Call	Covered
Point-to-Point	
MPI_Send, MPI_Recv	✓
MPI_Bsend, MPI_Rsend, MPI_Ssend	-
MPI_Isend, MPI_Irecv, MPI_Wait	✓
MPI_Sendrecv_replace	✓
MPI_Start	✓
Collective	
MPI_Barrier	✓
MPI_Bcast	✓
MPI_Gather, MPI_Scatter	✓
MPI_Gatherv, MPI_Scatterv	✓
MPI_Allgather, MPI_Allgatherv	-
MPI_Alltoall, MPI_Alltoallw	-
MPI_Reduce, MPI_Allreduce	✓
MPI_Reduce_Scatter	-
MPI_Scan, MPI_Exscan	-
MPI_Start, MPI_Recv_init, MPI_Send_init	✓

Table 4.1: Commonly used MPI calls and those supported by the Adjoint MPI library

The consequences are that the adjoint MPI library consists of an MPI tape similar to an AD overloading tool. It has to store and restore the runtime arguments to MPI. Note that the library itself is perfectly usable in an AD source transformation tool through a special wrapper. See Section 4.3 where a source transformation tool is used for generating higher adjoints of MPI enabled code. Second, MPI is defined using the programming languages C and Fortran. As such we chose to write our library in C with an accompanying Fortran interface.

4.2 Adjoint MPI Implementation

Most adjoint MPI communication patterns are a straightforward implementation deduced from the abstract constraints in Chapter 3. Before starting an implementation, one has first to design an interface that fulfills the requirements of portability with regard to any AD tool. This has to cover both Fortran and C languages as stated by the MPI standard. Next question is what type of link between value and adjoint all AD tools can agree on. We want an abstraction that does not rely for example on the data type. How can AMPI handle active data without requiring knowledge about its layout? Second are practical specifics of MPI. In Chapter 3, MPI requests and windows were abstract types. These abstract types do not match between different MPI implementation. How does AMPI become independent of any MPI implementation?

This section first covers the interface and introduces a portable data reversal mechanism for adjoints. It then deals with MPI communication patterns that require special treatment with respect to practical issues. It essentially deals with implementation issues that are generic in nature.

4.2.1 Portable Data Reversal Mechanism

The AMPI library should be capable of handling arbitrary data buffers and communications bound to the AD tool currently used. These data buffers need to be identified. The different layouts of the active datatypes and the difference between contiguous and interleaved buffers were laid out in Section 2.3.

Portability forces us to cover all layouts. Internally, AMPI communicates passive variables of type `double`. Contiguous types would then only need to map the active buffers or adjoint buffers of type `double` onto the communication buffers. This only relies on the buffer address and does not require element-wise access to the buffer elements. However, interleaving types like for example in `dco/c++` requires AMPI to handle the buffers element-wise since it has no notion of an active type's structure. It needs to extract or insert the value and the adjoint from the active type and tape, respectively. Therefore, four access methods have to be provided by the AD tool interface, namely `get_value(...)`, `set_value(...)`, `get_adjoint(...)` and `set_adjoint(...)`. Additionally, AMPI has a pointer to the active buffer, it knows the element it needs to extract the value from. AMPI has to know which value belongs to what adjoint. This is possible through a unique identifier that links a value to its adjoint. The method that handles the unique ID is the fifth interface method called `get_idx(...)`.

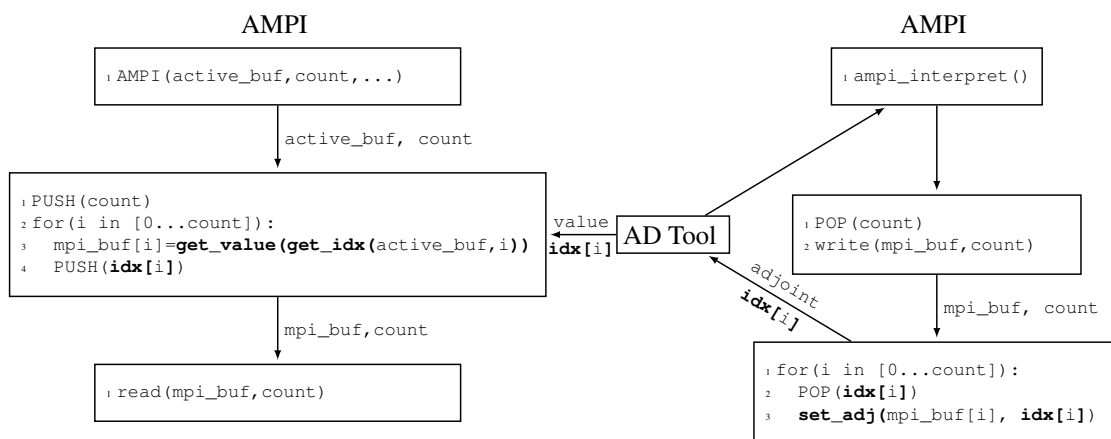


Figure 4.2: The sender of the statement $b.y = \text{SendRecv}(a.x)$: In the forward section the AMPI interface calls the AMPI library. For each active variable AMPI retrieves an index `idx` through the AD tool interface and its value. `count` is stored in the AMPI tape. In the reverse section the AD tool initiates an AMPI interpretation through `ampi_interpret`. AMPI restores the counter `count` and saves the adjoints in the AD tool tape through the identifier `idx` together with the received values.

Figure 4.2 and Figure 4.3 illustrate the generic interface between an arbitrary AD tool and the generic layout of the AMPI library. The interface covers read and write operations depending on whether MPI reads or writes data in the forward section. In the reverse section, the reads become writes and vice versa.

The sole element both tools have to agree on is the type of the identifier `idx`. Both in stack and trace based tools, this identifier is often a counter that increases in the forward section and decreases in the reverse section. Another approach would be a pointer that links the value and the adjoint. In any case, this link has to exist for non linear operations. Otherwise it is not possible to restore information in the reverse section that was stored in the forward section. In the AMPI library the type of the identifier is set through a `define`. The interface routines `get_value`, `set_value`, `get_adjoint` and `set_adjoint` are not dependent on AD tool specific definitions or declarations. This avoids any circular dependence between the AD tool and the AMPI library.

Although there is a considerable amount of function calls (e.g. one for each buffer element), the performance impact is usually negligible. The interface functions consist only of a few lines of code and thus they are implicitly inlined with compiler optimizations enabled.

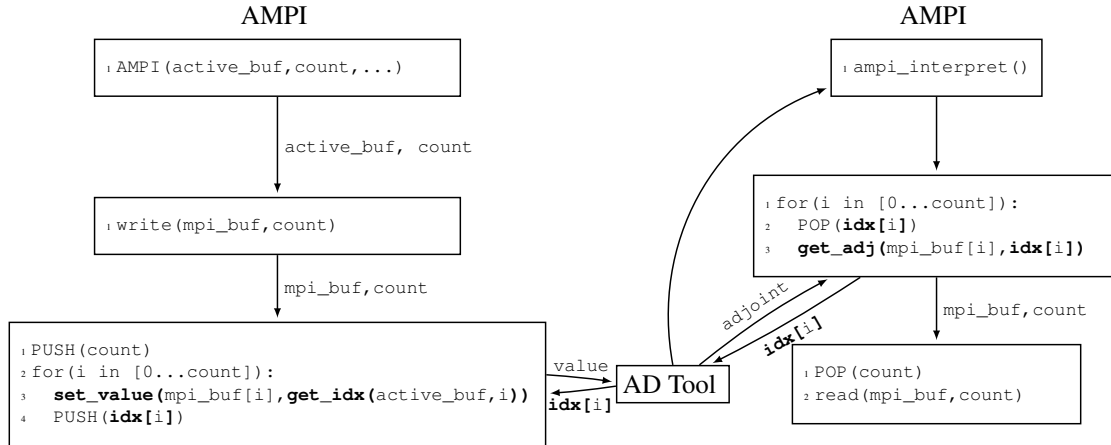


Figure 4.3: The receiver of the statement $b.y = \text{SendRecv}(a.x)$: In the forward section the AMPI interface calls the AMPI library. For each active variable AMPI retrieves an index `idx` and stores the received value through the AD tool interface. `count` is stored in the AMPI tape. In the reverse section the AD tool initiates an AMPI interpretation through `ampi_interpret`. AMPI restores the counter `count` and extracts the adjoints out of the AD tool through the identifier `idx` and dispatches them through MPI.

4.2.2 Nonblocking Point-to-Point Communication

Nonblocking communications define a period over which a communication may take place. It is initiated by a `MPI_Isend` or `MPI_Irecv` and closed by a `MPI_Wait`. There is an extensive semantic analysis in Section 3.4. This deferred data access has consequences for the `AMPI_Isend`, `AMPI_Irecv` and `AMPI_Wait`) and requires special handling in its implementation.

Figure 4.4 gives an overview of the forward section of nonblocking communications through the generic interface presented in Section 4.2.1. The three stages of the interface are essentially split into two (`AMPI_Isend` / `AMPI_Irecv` and `AMPI_Wait`). One for the initiation and one for the closure. In an `AMPI_Isend` the active types are copied to the communication buffer `mpi_buf` and an `MPI_Isend` is initiated. Inside the `AMPI_Wait`, it is matched by an `MPI_Wait`. The reception of data during the nonblocking period of a receive is initiated in `AMPI_Irecv` and has to be copied back into the active types in `AMPI_Wait`. However, `AMPI_Wait` has no active type in its signature, leaving the request as its only link to the active buffer of the communication. Adding the active buffer (`active_buf`) and buffer size (`count`) to the signature of `AMPI_Wait` would introduce an incompatibility between the AMPI and MPI interface. Moreover, the active buffer, albeit allocated, may not be available in the context of the code, thus requiring a restructuring of the code for AMPI. The only feasible solution is the introduction of a bookkeeping system. This may be implemented, for example, through a new active request type (`AMPI_Request`) that encompasses the original request as well as the additional information. The AMPI library uses a hash table described in Appendix B. In the reverse section, a similar situation unfolds to convey the adjoint buffer the `Isend` or `Irecv` to the `Wait`. This time however, the additional information can be stored on the AMPI tape and there is no dependence on additional information in the request. It is known whether an `AMPI_Wait` entry represents a receiving oder sending communication and we do know their tape location.

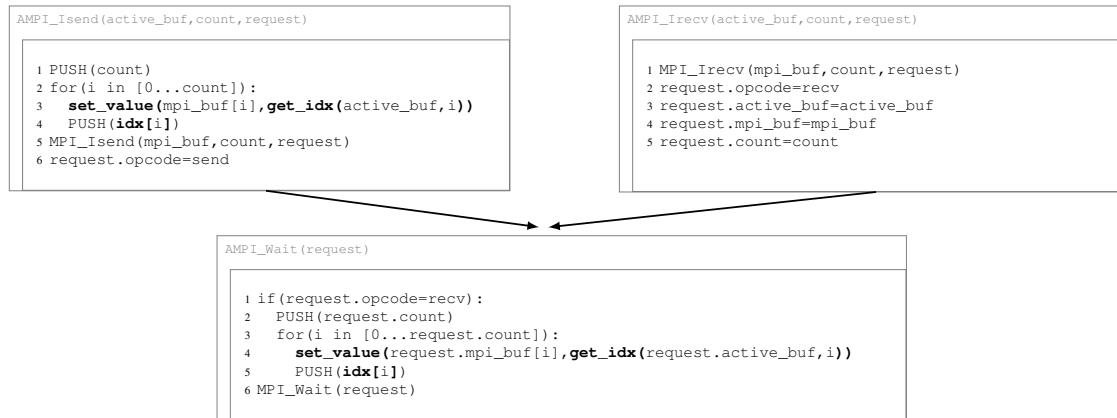


Figure 4.4: Nonblocking Peer-to-Peer communication: `MPI_Irecv` and `MPI_Isend` are the deferred `mpi_write` and `mpi_read` operations. Due to `MPI_Wait` having no active arguments, the information required by AMPI has to be conveyed through additional information in the `request`.

4.2.3 Collective Reduction Communication

The reduction and the accumulation mark a very unique feature in MPI. They are both a communication which execute one particular arithmetic operation on the buffer. There are predefined operations for the basic types. Of particular interest are the `MPI_SUM`, `MPI_PROD`, `MPI_MAX` and `MPI_MIN` operations in combination with `MPI_FLOAT` and `MPI_DOUBLE`. All the operations in MPI have to be associative while the predefined operations are additionally commutative. This allows MPI to change the order of execution and optimize the communication pattern with regard to the underlying hardware. Moreover, the communication and computation are merged and executed concurrently. However, all of this is opaque to the user of MPI. Only inside the underlying MPI library one is able to access the computational graph of such an execution.

Reduction

The semantics and the adjoint constraints for an adjoint computation of a reduction have been laid out in Section 3.6.2. One possible way of tracing the computational graph could be a derived type that conveys the rank throughout the reduction.

Listing 4.1: "Tracing the reduction using derived types and user defined operations"

```

1 #include <mpi.h>
2 #include <iostream>
3 #include <stdlib.h>
4
5 using namespace std;
6
7 int rank=0;
8
9 struct tup { double v; int r; };
10
11 void prod(void *invec, void *inoutvec, int *len, MPI_Datatype *
    datatype) {

```

```

12  tup *dinvec=(tup*) invec;
13  tup *dinoutvec=(tup*) inoutvec;
14  for(int i=0;i<*len;i++) {
15      dinoutvec[i].v*=dinvec[i].v;
16      if(dinvec[i].r!=rank)
17          cout << "sent from " << dinvec[i].r << " to " << rank << endl;
18  }
19 }
20 int main(int argc, char *argv[]) {
21     MPI_Op MY_MPI_PROD;
22     MPI_Init(&argc,&argv);
23     MPI_Datatype MY_MPI_DOUBLE;
24     MPI_Type_contiguous(1, MPI_DOUBLE_INT, &MY_MPI_DOUBLE);
25     MPI_Type_commit(&MY_MPI_DOUBLE);
26     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
27     tup *sbuf=new tup[10];
28     for(int i=0;i<10;i++) {
29         sbuf[i].v=(double) 1/((double) i+1);
30         sbuf[i].r=rank;
31     }
32     tup *rbuf=new tup[10];
33     MPI_Op_create(prod, 1, &MY_MPI_PROD);
34     MPI_Reduce(sbuf,rbuf,10,MY_MPI_DOUBLE,MY_MPI_PROD,0,MPI_COMM_WORLD);
35     MPI_Finalize();
36 }

```

An incomplete implementation illustrates the idea in Listing 4.1. A new derived type `tup` implementing a tuple of an `MPI_DOUBLE` and `MPI_INT` is committed. The integer serves as a storage of the origin rank for a variable. Instead of copying the active types to a temporary buffer of type `MPI_DOUBLE`, they are copied into these tuples. In the user defined product operation `prod` the multiplication is executed. If the rank of the incoming variable `invec` is different from the current rank, the communications source and destination is output. The problem here is, that the original processes, in particular the leaf nodes in case of a binary tree topography, do only send the variables in `invec`. They do not apply the operation `prod` on `invec`. Hence, it cannot be traced on the origins that the data was indeed sent. One could circumvent this problem through a more sophisticated MPI implementation using wildcard communication. However, this defies the purpose of a generic reduction by assuming any cluster topology. In summary, there is no way to trace and reverse the communications inside the reduction in the confines of MPI. We also considered this constraint in the semantic analysis (Section 3.6.2). Access to the internal communication pattern of the reduction could be added to the MPI standard.

4.3 Second-Order and Higher Adjoint Communication

Although the tangent-linear model is not the topic of this research, it is closely related to the adjoint model in the context of higher derivative code. The reapplication of the adjoint model does not yield the same runtime savings as the first-order adjoint model with respect to the number of outputs m . In order to accumulate the full Hessian, we chose one of the second-order adjoint models presented in [45] which all have a runtime complexity of $\mathcal{O}(nm) \cdot \text{cost}(F)$. For technical reasons, to generate second-order and higher adjoint code, this work resorts to the reapplication of the tangent-linear model on the

first application of the adjoint model called *forward over reverse mode*. Through the reapplication of the tangent-linear model, an arbitrary order of differentiation is attainable. This is later motivated by the fact that we want to use the adjoint MPI library in the higher adjoint mode. For reverse over reverse or other higher adjoint modes, a specialized adjoint MPI library would have to be reimplemented. Therefore, for simplicity, these use cases are avoided.

Higher-order derivatives are of crucial importance in the field of numerical optimization. Newton's method is used to find the roots of a function F iteratively with:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \nabla F(\mathbf{x})^{-1} \cdot F(\mathbf{x})$$

Minimizing or maximizing F is equivalent to finding the roots of ∇F (see example in Section 2.2.1):

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \nabla^2 F(\mathbf{x})^{-1} \cdot \nabla F(\mathbf{x})$$

Thus, optimization algorithms commonly rely on second-order information. Although algorithms like Gauss-Newton approximate second-order derivatives through first-order information they never achieve the same accuracy. Truncation and rounding errors are amplified when using finite-differences, while the second-order tangent linear model has a runtime complexity of $\mathcal{O}(n^2) \cdot \text{cost}(F)$ for the computation of the Hessian. Thus, the importance of the higher adjoint models is equivalent to the first-order adjoint model. Higher derivatives are often required due to algorithm encapsulation. Suppose a library implements second-order Newton's method for a minimization problem. This library may be called inside another second-order Newton code again. The AD tool would then have to cope with fourth order derivative code.

The tangent-linear code of an MPI implementation is rather straightforward and has been done as a side project of this thesis. The idea is to either communicate each order of tangents as a separate communication (source transformation) or introduce MPI derived data types and user defined operations that encapsulate the value and all the tangents into a new type (operator overloading).

First, the general implications of the tangent-linear model on MPI are presented in Section 4.3.1. An application of the second-order model called *forward over reverse* is demonstrated in Section 4.3.2 using the source transformation tool `docc`. At the end the generation of higher adjoint codes using AMPI is described in Section 4.3.3.

4.3.1 Tangent-Linear Communication

	Process a	Process b
$s_k :$	$b.x = \text{SendRecv}(a.x)$	$b.x = \text{SendRecv}(a.x)$
$s_k :$	$b.x = \text{SendRecv}(a.x)$ $b.x_{(1)} = \text{SendRecv}(a.x_{(1)})$	$b.x = \text{SendRecv}(a.x)$ $b.x_{(1)} = \text{SendRecv}(a.x_{(1)})$
$s_k :$	$(b.x, x_{(1)}) = \text{SendRecv}(a.x, a.x_{(1)})$	$(b.x, b.x_{(1)}) = \text{SendRecv}(a.x, a.x_{(1)})$

Figure 4.5: Duplication versus concatenation of *SendRecv* calls in tangent-linear code illustrated by a blocking send and receive

The tangent-linear model (see Section 2.2.3) does not reverse the data flow. The tangents are propagated in the same direction and at the same location as the values. In PGAS notation the communications are again modelled as assignments and are transferred through *SendRecv* calls. This means that the tangents are transferred in conjunction with the values. This conjunction may be achieved in two ways

illustrated in Figure 4.5. The first one is called *shadowing* where the communication calls *SendRecv* are duplicated. The values are communicated followed by a separate call with the tangents. In the second one the type of the data is changed into tuples of the original value type. Every communication is now doubled in size while the values and tangents are transferred in *concatenation* with one call to *SendRecv*.

Exactly these two alternatives also exist in practice. Concatenation is the method of choice if the original type is overloaded with a tangent-linear type essentially implementing the aforementioned tuples due to the association by address. Variables in codes generated by source transformation are mostly duplicated due to the association by name, one for the value and one for the tangent. Here, shadowing is the preferred method.

Next we want to look at the implications for MPI, especially if we want to use adjoint MPI coupled with a tangent-linear MPI implementation in order to compute higher adjoints. MPI has two features which pave the way for a concatenated type, namely derived data types and user defined operations. Based on the overloaded type a derived data type is created (see Listing 4.2 and Listing 4.3; a more generic implementation is available on the CD).

Listing 4.2: Tangent-linear derived type consisting of two `MPI_DOUBLE`s; value and tangent

```
1 MPI_Datatype T1S_TYPE;
2 MPI_Type_contiguous(1, MPI_DOUBLE_DOUBLE, &T1S_TYPE);
3 MPI_Type_commit(&T1S_TYPE);
```

Listing 4.3: User defined product

```
1 void AMPI_Prod_function(void *invec, void *inoutvec, int *len,
   MPI_Datatype *datatype) {
2   dco::tls::type *tls_invec=(dco::tls::type *) invec;
3   dco::tls::type *tls_inoutvec=(dco::tls::type *) inoutvec;
4   for(int i=0;i<*len;i++) {
5     tls_inoutvec[i]*=tls_invec[i];
6   }
7 }
```

This is the straightforward way of implementing a tangent-linear MPI (TMPI) library. The overloaded types are communicated and inside the user defined operations the overloaded operators are called. However, suppose the original type is already an active type where the logic of the AMPI library should be applied to. There is no way of calling the AMPI library, since the tangent-linear communication happens transparently. The other alternative is to shadow the tangent-linear communications by initiating one MPI communication for the value and each tangent component. Note that the value component is potentially already an active adjoint type. Although this is in particular used by source transformation tools, since they usually double the number of variables at each order of differentiation, it resolves the issue of finding an entry point to the AMPI library. The size of each communication is equal to the original buffer size. The AMPI library has to be called for the value and each tangent component. The number of AMPI calls that are shadowed by one TLM AMPI call is 2^d , where d is the order of differentiation.

4.3.2 Second-Order Code with `dcc`

`dcc` is a source transformation AD tool that generates derivative code in C out of an input code written in a specific subset of C/C++ [59]. It relies on `flex` and `bison` to generate an abstract syntax tree of the input code. The AD rules are applied to the internal representation according to the selected differentiation model. The differentiated code is then generated as C source code. The output language belongs to the same subset of C/C++ as the input code, allowing us to reapply `dcc` to generate a higher

$$F \left(\overset{\downarrow}{x}, \overset{\downarrow}{y} \right) \xrightarrow{dcc} F_{(1)} \left(\overset{\downarrow}{x}, \overset{\downarrow}{x}_{(1)}, \overset{\downarrow}{y}, \overset{\downarrow}{y}_{(1)} \right) \xrightarrow{dcc} F_{(1)}^{(2)} \left(\overset{\downarrow}{x}, \overset{\downarrow}{x}^{(2)}, \overset{\downarrow}{x}_{(1)}, \overset{\downarrow}{x}_{(1)}^{(2)}, \overset{\downarrow}{y}, \overset{\downarrow}{y}^{(2)}, \overset{\downarrow}{y}_{(1)}, \overset{\downarrow}{y}_{(1)}^{(2)} \right), \text{ where}$$

$$\begin{aligned} x_{(1)}^{(2)} &= x_{(1)}^{(2)} + x^{(2)T} \cdot \nabla^2 f(x) \cdot y_{(1)} + \nabla f(x)^T \cdot y_{(1)}^{(2)} & y^{(2)} &= \nabla f(x) \cdot x^{(2)} \\ x_{(1)} &= x_{(1)} + \nabla f(x)^T \cdot y_{(1)} & y &= f(x) \end{aligned}$$

Figure 4.6: Second-order adjoint derivative model generated by `dcc`. Upper arrows mark the inputs of a program whereas base arrows mark the outputs.

order differentiated code. The j th-order derivative code is generated by reapplication of `dcc` to the $(j - 1)$ th-order derivative code. `dcc` implements both the tangent-linear and adjoint modes of AD while preserving the reapplication feature.

The second-order forward over reverse model is presented in Figure 4.6. `dcc` applies the tangent-linear model presented in Section 2.2.3 to the incremental adjoint model in Section 2.2.4. Without the loss of generality we assume f to be a scalar function and assume the number of inputs n and the number of outputs m to be equal to one. For a more detailed coverage of higher order adjoints please refer to [45]. Hence, as scalars, the transposed of $x^{(2)T}$ is equal to $x^{(2)}$ and so on. For second derivatives every variable y , $y_{(1)}$, x , and $x_{(1)}$ of the adjoint code is augmented with an additional directional derivative component $y^{(2)}$, $y_{(1)}^{(2)}$, $x^{(2)}$, and $x_{(1)}^{(2)}$, respectively. In order to compute second-order derivatives, here the Hessian $\nabla^2 f(x)$, the variables have to be initialized correctly. With $y_{(1)}^{(2)}$ set to 0, $x_{(1)}^{(2)}$ set to 0, $x^{(2)}$ set to 1 and $y_{(1)}$ set to 1, we are able to harvest the Hessian in $y_{(1)}^{(2)}$. For completeness the adjoints $y_{(1)}^{(2)}$ and $y_{(1)}$ are set to 0 by `dcc` at end, so they may be regarded as outputs. Notice that if the seeding is done correctly, this code additionally computes first order derivatives that may be either harvested in $x_{(1)}$ or $y^{(2)}$.

Listing 4.4: Sample C/C++ code differentiable using `dcc`

```
1 void f(double *x, double cost) {
2     ...
3     c=c*x[myid];
4     MPI_Reduce(cost,c,root);
5 }
```

Listing 4.4 is a C/C++ example code that is differentiable using `dcc`. A function `f` computes a local product `c*x[myid]` and then reduces the result to the global variable `cost` using MPI.

Listing 4.5: Adjoint code according to the adjoint model using `dcc`

```
1 void a1_f(int m, double* x, double* a1_x, double cost, double
2     a1_cost) {
3     m=1; // forward run
4     ...
5     push(c); c=c*x[myid];
6     a1_MPI_Reduce(m, cost, a1_cost, c, a1_c, root);
7
8     m=2; // reverse run
9     a1_MPI_Reduce(m, cost, a1_cost, c, a1_c, root);
```

```

9  pop(c);
10  a1_x[myid]+=c*a1_c; a1_c+=x[myid]*a1_c;
11  ...
12 }

```

In the adjoint code (see Listing 4.5) every variable that is of differentiable floating point type is augmented by its adjoint. Before a variable is overwritten it is pushed onto a stack (e.g. `push(c)`). The focus is the function signatures, since this influences how MPI is differentiated. `dcc` differentiates arbitrary function calls. Each differentiable argument is augmented by its tangent or adjoint while the function name is prepended by the differentiation mode prefix (here `a1_`). There is one additional integer parameter `m` which will be explained in more detail in the next Section 4.3.2. It is used to distinguish between the forward and reverse section, $m = 1$ marks the forward section while $m = 2$ is the reverse section. Only the function calls are differentiated. The wrapper needs to provide the functions that match the signatures of the differentiated MPI function calls.

Listing 4.6: Second-order adjoint code according to the forward over reverse model using `dcc`

```

1 void t2_a1_f(int m, double* x, double *t2_x, double* a1_x, double*
    t2_a1_x, double cost, double t2_cost, double a1_cost, double
    t2_a1_cost) {
2   m=1; // forward run
3   ...
4   push(c); c=c*x[myid];
5   t2_c=c*t2_x[myid]+t2_c*x[myid];
6   t2_a1_MPI_Reduce(m, cost, t2_cost, a1_cost, t2_a1_cost, c, t2_c,
    a1_c,t2_a1_c,root);
7
8   m=2; // reverse run
9   t2_a1_MPI_Reduce(m, cost, t2_cost, a1_cost, t2_a1_cost, c, t2_c,
    a1_c,t2_a1_c,root);
10  pop(c);
11  a1_x[myid]+=c*a1_c;
12  t2_a1_x[myid]+=t2_c*a1_c+c*t2_a1_c;
13  a1_c+=x[myid]*a1_c;
14  t2_a1_c+=t2_x[myid]*a1_c+x[myid]*t2_a1_c;
15  ...
16 }

```

Listing 4.6 is the application of the tangent-linear model on the adjoint code of Listing 4.5 according to the forward over reverse model in Figure 4.6. Every variable and adjoint is augmented by its tangent in the code and in the function arguments. The function `t2_a1_MPI_Reduce` is called once with $m = 1$ in the forward section and once with $m = 2$ in the reverse section. The next section deals with how the AMPI library should be eventually called inside wrapper functions.

AMPI calls and joint/split reversal

`dcc` uses *joint call reversal* [63] shown in Figure 4.7b as its data-flow reversal model by introducing three modes $m=1, 2, 3$. It is a trade-off between computational and memory complexity of the generated adjoint code. Let `foo` be the root routine. The three modes are defined as follows:

- $m=1$: forward and reverse run
- $m=2$: store arguments + original routine call

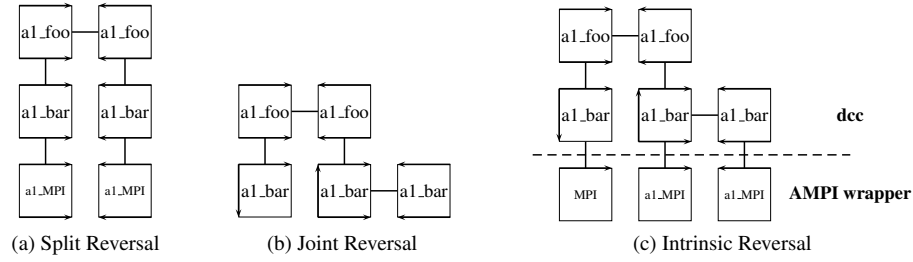


Figure 4.7: Reversal models

- $m=3$: restore arguments + forward and reverse run

Only the root routine `a1_foo` is called with $m=1$. It starts its forward run (double right arrow) and calls `a1_bar` with $m=2$. `a1_bar` then stores its arguments (down arrow) and runs the undifferentiated original routine `bar` (single right arrow). `a1_foo` now starts its reverse run (double left arrow). To do this, it needs the adjoints of the reversed routine `a1_bar`. This is done by calling `a1_bar` with $m=3$. `a1_bar` restores its arguments (up arrow) and performs its forward run (double right arrow) followed by the reverse run (double left arrow). We end up with all the adjoints when `a1_foo` finishes its reverse run.

Lets assume the routine `bar` has an MPI call in joint reversal. *Intrinsic reversal* is then used for the called `AMPI_Reduce`. Just like any intrinsic function AMPI routines are differentiated through provision of their differentiated counterpart by the AMPI library and not by `dcc`. Thus the forward and the reverse run can be separated, since there are internal structures that link these two calls if necessary. In Figure 4.7c we see that with joint reversal, the adjoint buffers are the same for both `a1_MPI` calls since they are called by the same instance of the routine `a1_bar` as opposed to split reversal in Figure 4.7a. Obviously this comes at a communication cost, since the original MPI routine has to be called during the undifferentiated call ($m=2$) of `a1_bar`.

The prime benefit of our approach of adjoining MPI communication is the avoidance of modification of `dcc`'s source transformation algorithms. `dcc` handles all MPI calls similar to any arbitrary user-written subroutine by calling adjoint versions in modes $m=1, 2, 3$. Dealing with memory management issues is deferred entirely to the wrapper.

AMPI Wrapper for `dcc`

The `AMPI_dcc` wrapper library is the interface between AMPI and the adjoint code generated by `dcc`. Its sole purpose is to ensure that values and adjoints are mapped properly from and to the AMPI routine calls. Hence, the relevant part of the AMPI calling signatures is the handling of values and adjoints. The *forward routines* need to be called with the original values as arguments in the forward run, whereas the *backward routines* need to be called with the adjoints as arguments in the reverse run. The logic of transforming the MPI communications is entirely handled by the AMPI library. Inside the `AMPI_Reduce_b` the reverse operation of the `MPI_Reduce` together with a sum operation `MPI_SUM` is executed. This is done transparently and amounts to an `MPI_Broadcast` where the adjoints are sent to other processes (see Section 3.6.2).

We want to move a step further and compute second derivatives by reapplying `dcc` in tangent-linear mode to the adjoint code. According to the second-order model in Figure 4.6, the Reduce will be differentiated again yielding

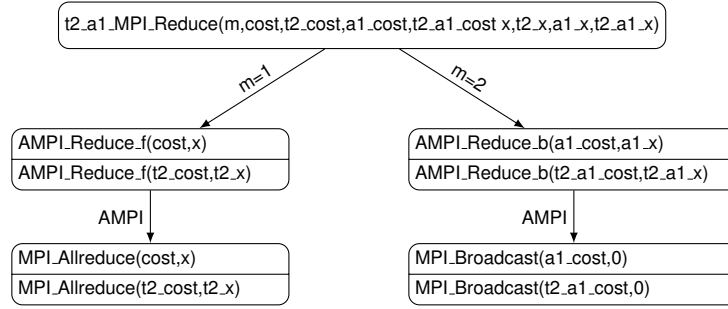


Figure 4.8: t2_a1_MPI_Reduce

```

t2_a1_MPI_Reduce(m, cost, t2_cost, a1_cost, t2_a1_cost, x, t2_x, a1_x,
t2_a1_x).

```

The function prefix `t2_a1` stands for the second-order adjoint code obtained by tangent-linear (`t2_`) over adjoint (`a1_`) mode. The second-order model in Figure 4.6 implies that every value and every adjoint get an additional tangent-linear component (highlighted) prefixed with `t2_`. Hence there are twice as many variables in the signature of `t2_a1_MPI_Reduce` than in the first-order adjoint routine `a1_MPI_Reduce`. Since the underlying AMPI library does not have specific second-order routines, we need to split the communication into two AMPI calls as shown in Figure 4.8. This amounts to a doubling of the MPI communication. As in first-order adjoint mode `t2_a1_MPI_Reduce` is called in the forward run and in the backward run. In the forward run ($m=1$) `t2_a1_MPI_Reduce` calls the AMPI library twice: Once to reduce the value of `x` to `cost` and once more to reduce the value of the tangent-linear component of `t2_x` to `t2_cost`. The resulting MPI calls are two Allreduction calls. As in first-order adjoint mode, the Allreductions are necessary, since each process needs the values of `cost` and `t2_cost` to compute the adjoints in the reverse run. In the reverse run ($m=2$), the adjoints are propagated in reverse order of the original code. Therefore the reverse AMPI Reduce routine `AMPI_Reduce_b` is called. Again this is done twice since the tangent-linear components (`t2_a1_cost`, `t2_a1_x`) of both adjoints and the adjoints (`a1_cost`, `a1_x`) themselves need to be set. The actual resulting MPI communications are the two Broadcasts of `a1_cost` and `t2_a1_cost`. `a1_x` and `t2_a1_x` are computed based on the adjoint of the reduction operation.

The procedure of linking each second-order adjoint MPI routine with the corresponding AMPI library routines has been implemented and tested for blocking send and receive, non-blocking send and receive, reduction and broadcast.

4.3.3 Higher Adjoints

The generation of higher adjoints is straightforward. `dcc` is reapplied to the second-order code, thus creating `t3_t2_a1_MPI` routines. Inside these routines the second-order routines `t2_a1_MPI` are called twice. By analogy, a wrapper has to be written exactly the same way as described in the previous section. This manual generation of the wrappers could be easily automated using a script.

Using overloading (e.g. `dco/c++`), the approach is very similar, although not as straightforward. As has been laid out before, AMPI requires shadowing as the method of choice for transferring the tangents. Thus, a wrapper has to be written for each order of differentiation which extracts the $(j - 1)$ -th order active types out of the j -th order of active types down to the adjoint type, which is eventually transferred using the generic AMPI library. This is a real trade-off that the generic AMPI interface imposes on the user. However, assuming that higher adjoints are a rather special requirement, the benefit is a really simple interface.

Chapter 5

Applications

This chapter presents all the applications that the adjoint MPI library (AMPI) was tested with. It covers both AD tools as well as simulation codes. The goal is not only to verify that the approach works, but to demonstrate the consequences of this method for applications. We take the user's as well as the developer's perspective.

The first example in Section 5.2 is the adjoint computation of the heat equation. It is a small yet complete example that demonstrates the integration of AMPI inside a code in order to differentiate non-blocking communication. The next section (Section 5.3.1) covers the AD-enabled NAG Fortran Compiler that emerged out of the CompAD project [49, 57] and its application to Sisyphe/Telemac. The difficulty here lies in the Fortran language where a valid interface has to be defined and the numerical effects of AMPI on such an extensive code base. The last application is PETSc (Section 5.4), a generic framework of parallel and portable solvers. It often constitutes the core of numerical codes and serves here as a use case of fully discrete versus continuous adjoints on parallel systems with regard to AMPI. All three case studies are motivated by a parameter optimization problem described in Section 2.2.1.

5.1 Hardware Setup

Name	System Type	Nodes available	RAM (GB)	CPU
Heisenberg	Fujitsu Workstation	1	128	Intel Xeon E5-2630@2.30GHz
RWTH Compute Cluster	Bullx Blade B500	252	96	Intel Westmere X5675@3.07

Table 5.1: Hardware used for benchmarks and runtime tests of AMPI

The hardware used in the three case studies is either the institute cluster *Heisenberg* or the RWTH Compute Cluster at the RWTH IT Center (ITC). Heisenberg is a special hardware setup tailored towards the computation of adjoints. It has a relatively high amount of RAM (128 GB). Its outstanding feature is a RAID controller that has 8 SSD drives connected to it. This allows a high throughput that, if used sequentially, reaches the same level as RAM. It consists of two CPUs or nodes that have each their own memory bus to the RAM, thus allowing only two MPI processes for benchmarking purposes. The Xeon E5-2630 is based on the Sandy Bridge chipset and is capable of 8 double precision floating point operations per cycle. The ratio of flops per memory λ was introduced in Section 3.1.2 as a measurement of how fit a system is for the computation of adjoints. The system is composed of two CPUs. Hence, the theoretical maximum of λ is equal to $\frac{128 \text{ GB}}{2.3 \text{ GHz} \cdot 8 \text{ FLOP} \cdot 2 \text{ cores}} = 3.4 \left[\frac{\text{GB} \cdot \text{s}}{\text{FLOP}} \right]$. This system is used to benchmark the slowdown of AMPI enabled code compared to sequential code. Writing the tape on the

RAID using two processes would create a bottleneck that is not subject of this work. So we are left with the available RAM of 128 GB shared by at most two processes.

The RWTH Compute Cluster has several configurations out of which we selected the one with the highest ratio λ . The Intel Westmere X5675 runs at 3.07 GHz with 4 double precision floating point operations per cycle and two CPUs on one socket. Hence, the theoretical maximum of λ is equal to $\frac{96 \text{ GB}}{3.07 \text{ GHz} \cdot 4 \text{ FLOP} \cdot 2 \text{ cores}} = 3.9 \left[\frac{\text{GB} \cdot \text{s}}{\text{FLOP}} \right]$. Although its λ value is similar than for Heisenberg, it was much harder to exploit the system's performance in practice. Its batch system, used to submit jobs, is not designed for adjoint computations. If no checkpoints are used, adjoint computations use a lot of memory with a rather short runtime. The requested memory per process had to be considerably reduced in order for the job to start at all. The situation is worsened by the fact that the use cases in this work do not exploit multithreading. Occupying the entire memory of one socket by requesting 96 GB results in requesting (nodes per socket) $\cdot$ (cores per node) $= 2 \cdot 6 = 12$ cores, when in fact only one core is active. Eventually, this illustrated the issues of requiring a high value of λ for generating adjoints on current cluster systems.

The RWTH Compute Cluster was only used for PETSc and not for Sisyph. One reason is the inherent code structure of Sisyph that makes the values of a simulation dependent on the number of processes p . With different values of p the code produces different values, especially if the problem size is small with respect to the number of processes p ; which was the case when computing adjoints. Next, the runtime behaviour of Sisyph with a high number of processes (> 4) resulted in an actual slowdown. This may again be related to the unrealistic small problem size of our example. Such setups were left out, because it is unrealistic to conclude any meaningful runtime behaviour with regard to AMPI.

5.2 Overview: Heat Equation

A simple data assimilation problem is considered involving the one-dimensional heat equation. A bar of a given length is heated on one side for some time until the temperature distribution reaches an equilibrium. The simulated temperature distribution is compared with available measurements at a number of discrete points. The initial temperature distribution within the bar is to be estimated such that the discrepancy between simulated and measured values is minimized. This case study aims at demonstrating the easy integration of AMPI into an AD enabled code. Here, the focus is on the AMPI interface and the necessary code adaptations. In addition the numerical properties of this problem are only discussed superficially, since it is a purely academic example.

The following optimization problem should be solved $\min J$, where

$$J = \sum_{i=0}^{n_x} \left(T(1, x_i) - \tilde{T}_i \right)^2$$

such that

$$T_t = c \cdot T_{xx} \quad \text{for } 0 \leq x, t \leq 1$$

and where

$$x_i = \frac{i}{n_x} \quad \text{for } i = 0, \dots, n_x$$

$$T = T(t, x) : \mathbb{R}^2 \rightarrow \mathbb{R}$$

$$\tilde{T} \in \mathbb{R}^{n_x} \quad (\text{observations})$$

$$T^0 = T(0, x) = f(x) \quad \text{for } 0 < x < 1 \quad (\text{initial condition})$$

$$T(t, 0) = \alpha \quad \text{for } 0 \leq t \leq 1 \quad (\text{left boundary condition})$$

$$T(t, 1) = \beta \quad \text{for } 0 \leq t \leq 1 \quad (\text{right boundary condition}).$$

Discretization in space is done by centered finite differences with step size δx . Explicit Euler is used for time integration with time step

$$\delta t \leq \frac{(\delta x)^2}{2 \cdot c}$$

to ensure stability [26] and to get

$$\frac{T_j^{k+1} - T_j^k}{\delta t} = c \cdot \frac{T_{j+1}^k - 2 \cdot T_j^k + T_{j-1}^k}{(\delta x)^2}$$

and hence

$$\begin{aligned} T_j^{k+1} &= T_j^k + c \cdot \frac{\delta t}{(\delta x)^2} \cdot (T_{j+1}^k - 2 \cdot T_j^k + T_{j-1}^k) \\ &= T_j^k + c \cdot \frac{n_x^2}{n_t} \cdot (T_{j+1}^k - 2 \cdot T_j^k + T_{j-1}^k), \end{aligned}$$

where $n_x = (\delta x)^{-1}$ and $n_t = (\delta t)^{-1}$.

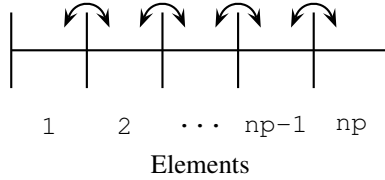


Figure 5.1: Halo exchange

The cost function is implemented in C++ as shown in Listing 5.1. For parallelization the bar is decomposed into np elements (lines 3-5). Each process computes one time step on its element (lines 8-10), followed by an halo exchange with the neighboring elements (lines 13-34). This simple setup is illustrated in Figure 5.1. The individual contributions `mpicost` to the overall costs are finally reduced to `cost` (lines 35-40). The halo exchange and the reduction of the cost or some residual is a very generic parallel setting in numerical science. For the Halo exchange point-to-point communication is used while the computation of the cost is achieved through a collective reduction.

```

1 active cost(int& nx, int& nt, active& delta_t, active& c, active*
    temp, active* temp_obs) {
2   active buf[4]; active mpi_cost = 0; MPI_Request request[4];
    MPI_Status status[4];
3   int mpi_j=(id * (nx/np))+1;
4   int mpi_nx=((id+1) * (nx/np))+1;
5   MPI_Comm_rank(MPI_COMM_WORLD,&id);
6   if(id == np -1)
7     mpi_nx--;
8   for(int i=0 ; i <= nt ; i++){
9     for(int j=mpi_j ; j<mpi_nx ; j++) {
10      temp[j] = temp[j]*c*nx*delta_t*(temp[j+1]-2*temp[j]+temp[j-1])
          ;
11    }
12    // recieve from right & send to right
13    if(id != np - 1){

```

```

14     buf[2]=temp[mpi_nx-1];
15     AMPI_Isend(&buf[2],1,MPI_DOUBLE,id+1,0,MPI_COMM_WORLD,&request
16               [2]);
17     AMPI_Irecv(&buf[1],1,MPI_DOUBLE,id+1,0,MPI_COMM_WORLD,&request
18               [1]);
19     }
20     // recieve from left & send to left
21     if(id != 0){
22         buf[0]=temp[mpi_j];
23         AMPI_Isend(&buf[0],1,MPI_DOUBLE,id-1,0,MPI_COMM_WORLD,&request
24               [0]);
25         AMPI_Irecv(&buf[3],1,MPI_DOUBLE,id-1,0,MPI_COMM_WORLD,&request
26               [3]);
27     }
28     if(id != np-1) {
29         AMPI_Wait(&request[1],&status[1]);
30         AMPI_Wait(&request[2],&status[2]);
31         temp[mpi_nx] = buf[1];
32     }
33     if(id != 0) {
34         AMPI_Wait(&request[0],&status[0]);
35         AMPI_Wait(&request[3],&status[3]);
36         temp[mpi_j-1] = buf[3];
37     }
38     }
39     cost=0;
40     for(int j=0 ; j<mpi_nx ; j++) {
41         mpi_cost = mpi_cost +(temp[j] - temp_obs[j])* (temp[j]-temp_obs[
42             j]);
43     }
44     AMPI_Reduce(&mpi_cost, &cost, 1, MPI_DOUBLE, MPI_SUM, 0,
45               MPI_COMM_WORLD);
46     return cost;
47 }

```

Listing 5.1: The cost function implementation of the optimization problem based on the heat equation

In its default configuration the AMPI library needs very little adaptations to the original code. For a differentiated version by operator overloading, two adjustments must be made. All variables of type `double` must have their type changed to the AD tool specific datatype `active`, and the names of all `MPI_*` communication routines must be renamed to `AMPI_*` (`MPI_Isend`, `MPI_Irecv` and `MPI_Wait`). The MPI routines not related to communication are left untouched (`MPI_Comm_rank`). MPI specific variables too do not need a type change (`MPI_Status` and `MPI_Request`). The request `MPI_Request` is used for an internal bookkeeping described in Section 4.2.2. A custom communicator (default: `MPI_COMM_WORLD`) can optionally be used and saved in the AMPI tape. The MPI datatype for the MPI calls is set to `MPI_DOUBLE` in the default configuration. For all AMPI communications with a different MPI type, the call is directly passed to the MPI library. This type can be set to a custom type (e.g. `AMPI_DOUBLE`) for distinguishing active and passive floating point variables. The built-in MPI reduction operations (here `MPI_SUM`) are all known to MPI.

The AMPI library directory has to be set via the environment variable `AMPI_DIR` (here: `~/hg/`

AdjointMPI). The path `AMPI_DIR/include` has to be added to the include path and the header `ampi_tape.hpp` has to be included. The AMPI library has to be linked via `-lampi -LAMPI_DIR/src`. We consider the case $c = 10^3$, $n_x = 200$, $n_t = 1000$, $\alpha = 2$, $\beta = 0$, and $f(x_i) = 2 - \frac{i}{100}$ for $i =$

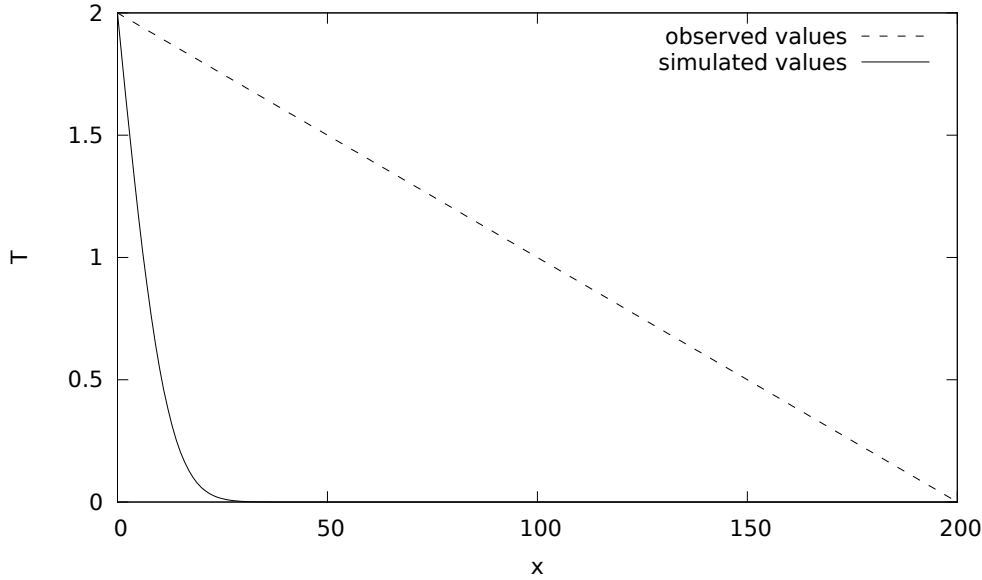


Figure 5.2: Discrepancy between original simulation of heat distribution starting from $T^0 = 0$ and observed values set to $f(x_i) = 2 - \frac{i}{100}$.

$0, \dots, n_x$. The substantial discrepancy between the originally simulated and the observed temperature distributions in Figure 5.2 at $n_t = 1000$ timesteps is reduced by applying the steepest descent method (see Figure 5.3). Starting from an initial `cost` of 227 for $T^0 = 0$, the algorithm performs 401 iterations (taking ~ 70 seconds on our PC in the sequential case and ~ 35 seconds when using 4 processes) to decrease the norm of the gradient to a value less than 10^{-3} . This was the first code that was automatically differentiated using a prototype of the AMPI library. It demonstrates in Listing 5.1 the AMPI interface that is used for the Halo exchange and the cost reduction. The same code structure is used in the next two case studies which analyze the potential slowdown created by AMPI.

5.3 Use Case: NAG Fortran Compiler and Sisyphe

The last section described the general approach of coupling AMPI with an MPI enabled C/C++ code. Another programming language commonly used in numerical science is Fortran. It has its origin in 1957 as one of the first imperative programming languages. The name FORMula TRANslation hints at its primal use case: the numerical science. The language has steadily evolved and is still widely used as of today. Whether this is due to legacy code or new projects is open for debate. The MPI standard defines equivalent C and Fortran interfaces. There are a few shortcomings in Fortran due to the different type handling [40, p. 480], but there is currently no sign that Fortran will fade in the near future. Hence, providing an AMPI solution in Fortran is crucial for its acceptance in the numerical science community.

This next sections present the coupling of AMPI with the AD enabled NAG Fortran compiler in Section 5.3.1 and Section 5.3.2. This is followed by a case study in Section 5.3.3 where AMPI is integrated into the sediment transport simulation *Sisyphe*.

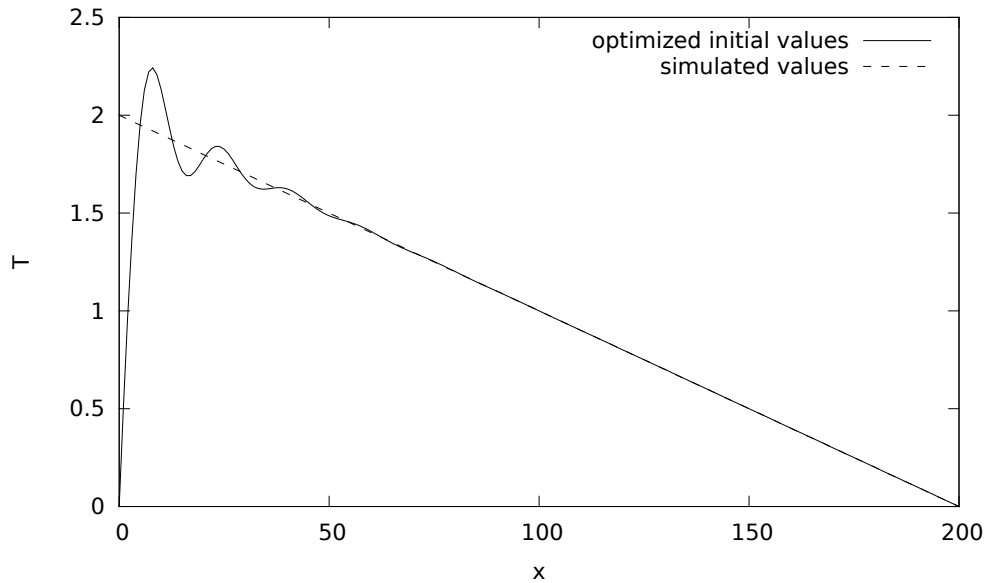


Figure 5.3: Simulation of heat distribution with optimized initial values

5.3.1 AD-enabled NAG Fortran Compiler (CompAD)

The NAG Fortran Compiler is known as a robust and reliable tool for compiling Fortran code. Its strict standard compliance guarantees that code which is successfully compiled by the NAG compiler is accepted by any other Fortran compiler. By adding AD to its feature list, the NAG Compiler is one of the first Fortran compilers that comes with AD capabilities out of the box and makes it a powerful tool in numerical simulation sciences.

CompAD [49] started as a joint effort by the Numerical Algorithms Group (NAG), Oxford, and the University of Hertfordshire, Hatfield, UK, to incorporate AD capabilities into the NAG Fortran compiler. Later stages (CompAD II and CompAD III) [57] involved among others our research team member Jan Riehme who gave the support for CompAD in this work. CompAD's emphasis was on usability and robustness. Its transparent integration into a compiler and its ease of use make this tool one of the most complete AD tools available. Although coupled with an AD compiler, CompAD does not resort to source transformation. It incorporates an overloading AD solution, the CompAD tape. The compiler applies the AD related code adaptations automatically and deals in a generic way with drivers, checkpoints and IO through the CompAD interface.

In order to avoid a reimplementaion in Fortran, we opted for a Fortran interface of the AMPI C library. The active forward and reverse MPI routines are implemented by our basic C library as presented in Chapter 4.2. We are going through the different steps needed for the adjoint tape module integration. The goal is to create a generic Fortran interface not only for CompAD but for any arbitrary Fortran AD tool.

5.3.2 AMPI and CompAD

The CompAD adjoint tape module consists of the following source files:

- `compad_module_adj_tape_common.F90`: types, constants

- `compad_module_adj_tape_opcodes.F90`: opcodes
- `compad_module_adj_tape.F90`: tape code

These files implement the CompAD library without adjoint MPI. The following files are only related to AMPI:

1. `compad_module_adj_tape_ampi.F90`: AMPI tool interface
2. `ampi_fortran_wrapper.c`: Fortran interface in C
3. `ampi_fortran_interface.F90`: Fortran to C interface
4. `compad_module_adj_tape_mpi.F90`: AMPI communication routines interface (higher dimensions, MPI types, ...) matching the MPI interface

The CompAD tape traces every operation and stores their opcode according to AD by operator overloading (see Section 2.3). AMPI communications are treated the same way as operations while relying on the AMPI interface presented in Chapter 4. Whenever an AMPI communication is called, the AMPI library creates a tape entry on the CompAD tape together with AMPI dummies which store the values of data transferred through AMPI. In the reverse section CompAD calls the AMPI library whenever a tape entry with the AMPI opcode is hit. This trace and reversal process requires two adaptations to the core CompAD tape:

1. CompAD tape interpreter extension for the AMPI opcode
2. C struct interface for the Fortran CompAD type

```

1 TYPE, BIND(C) :: compad_type
2   SEQUENCE
3   REAL(C_DOUBLE)      :: v   = 0.D0
4   INTEGER(TAPE_ISO_IKND) :: j   = 0
5 END TYPE compad_type
```

Listing 5.2: CompAD type using `iso_c_binding`

Extending the CompAD tape interpreter is a mere addition of one opcode with an external definition of the AMPI tape interpretation interface `ampi_interpret_tape()`. However, creating an interface of the C struct to the Fortran type through the Fortran `iso_c_binding` requires the Fortran type to be strictly sequential (see Listing 5.2). This has the drawback of forcing every derived type that includes a variable of type `CompAD_TYPE` to be strictly sequential. Although no detrimental effect has been observed so far, it forces code amendments onto the user. An AMPI library written entirely in Fortran would remove this requirement.

We now move over to the code that only deals with the AMPI library. Figure 5.4 gives an overview of how the AMPI Fortran interface is designed. It consists of two parts. First there is the Fortran interface to the generic C interface introduced in Section 4.2.1.

```

1 SUBROUTINE AMPI_GET_VAL (compad_elem, i, v) BIND(C, name=  
    ampi_get_val_fort')
2   TYPE (CompAD_TYPE), DIMENSION (*)      ::  
    compad_elem
3   INTEGER (C_INT)                          :: i
4   REAL (C_DOUBLE)                          :: v
5
```

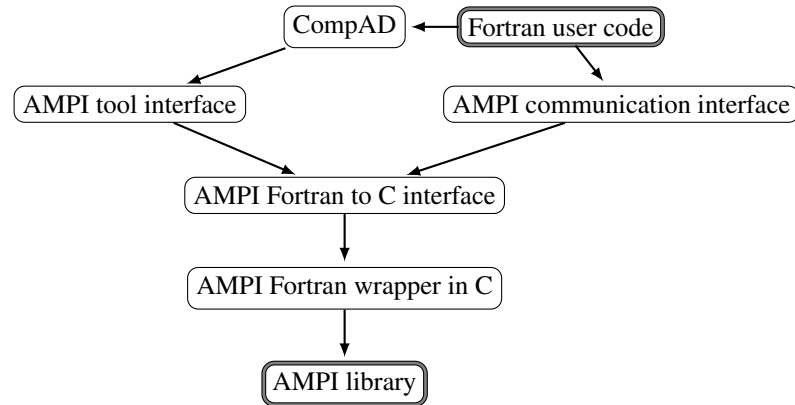


Figure 5.4: An overview of the AMPI Fortran implementation in CompAD through the dependency graph. The original Fortran user source code relies on CompAD as well as on an AMPI interface that is the equivalent of the MPI interface.

```

6   v = compad_elem(i+1)%v
7   END SUBROUTINE AMPI_GET_VAL

```

Listing 5.3: Fortran AMPI tool interface in `compad_module_adj_tape_ampi.F90`

An example of the Fortran interface routine `AMPI_GET_VAL` is shown in Listing 5.3. It is bound to the C function `ampi_get_val_fort` which itself implements the external defined function `ampi_get_val`. This additional hop is required because of some pointer particularities. All in all, it is a straightforward Fortran to C interface implementation which is also available on the accompanying CD.

```

1 void ampi_isend_fort(compad_type *buf, int *count, int *datatype,
   int *dest, int *tag, int *comm, int *request, int *ierr)
2 {
3   AMPI_Isend(buf, *count, MPI_DOUBLE, *dest, *tag, MPI_COMM_WORLD,
   request_map[*request])
4 }

```

Listing 5.4: Fortran to C wrapper in `ampi_fortran_wrapper.c`

Another part is a reimplement of the Fortran AMPI interface based on the MPI Fortran interface. The Fortran and C MPI interface differ in some aspects, although the AMPI library is only available in C. On the C side the C AMPI interface has to be converted and a C wrapper has to be built that matches the Fortran AMPI interface. This is done in Listing 5.4. All Fortran routines have no return value. They return the error code through the last argument. Note that in the case of the nonblocking send, the requests of type integer have to be managed. They are mapped on C MPI requests `request_map` of type `MPI_Request` via a hash table based lookup table. Other interface differences between Fortran and C had to be solved here.

```

1 SUBROUTINE AMPI_ISEND_C (buf, count, datatype, dest, tag, comm,
   request, ierr) BIND(c, name='ampi_isend_fort')
2 USE iso_c_binding
3 USE compad_module_adj_tape_common
4 TYPE (CompAD_TYPE), DIMENSION (*) :: buf

```

```

5  INTEGER(c_int)                                :: count, datatype, dest, tag,
      comm, request, ierr
6  END SUBROUTINE AMPI_ISEND_C

```

Listing 5.5: Fortran to C interface in `ampi_fortran_interface.F90`

The next interface in `ampi_fortran_interface.F90` deals exclusively with the link of Fortran and C. It is a straightforward projection of the C functions in Listing 5.4 using the `iso_c_binding` module.

```

1  INTERFACE AMPI_ISEND
2      MODULE PROCEDURE AMPI_ISEND_0d, AMPI_ISEND_1d
3  END INTERFACE
4  SUBROUTINE AMPI_ISEND_0d(buf, count, datatype, dest, tag, &
5      comm, request, ierr)
6      TYPE(CompAD_TYPE)    :: buf
7      TYPE(CompAD_TYPE), DIMENSION(1) :: buf_
8      buf(1)=buf
9      ...
10     CALL AMPI_ISEND_C(buf, count, datatype, dest, tag, comm, request, ierr)
11     ...
12 END SUBROUTINE AMPI_ISEND_0d
13
14 SUBROUTINE AMPI_ISEND_1d(buf, count, datatype, dest, tag, &
15     comm, request, ierr)
16     TYPE(CompAD_TYPE), DIMENSION(:) :: buf
17     ...
18     CALL AMPI_ISEND_C(buf, count, datatype, dest, tag, comm, request, ierr)
19     ...
20 END SUBROUTINE AMPI_ISEND_1d

```

Listing 5.6: Fortran AMPI interface in `compad_module_adj_tape_mpi.F90`

And last, a Fortran language specific interface for MPI is needed in Fortran (see Listing 5.6) for the array dimensions zero and one of the MPI communication buffers.

5.3.3 Sisyphe

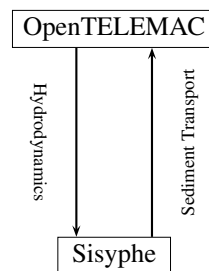


Figure 5.5: Feedback loop of OpenTELEMAC and Sisyphe

OpenTELEMAC [27] is an open source finite elements hydrodynamics simulation framework developed by Électricité de France (EDF, France), Bundesanstalt für Wasserbau (BAW, Germany) and others. Part of the framework is Sisyphe, a sediment transport simulation relying on other components

for hydrodynamic information (e.g. TELEMAC-2D). Additionally, the sediment transport influences the hydrodynamics computation of OpenTELEMAC. Both tools are in a feedback interaction loop. At each computational step of OpenTELEMAC, Sisyphé is called to compute the sediment transport. Sisyphé itself has callbacks to OpenTELEMAC which change the bottom topography of the given flow problem (see Figure 5.5). Currently, Sisyphé is used to simulate sediment transport in rivers like for example the Rhine. The goal is to predict the depth of the river in order to guarantee a continuous ship movement on the river. However, ultimately engineers want to influence the sediment transport by targeted interventions into the flow of the river, e.g. by constructing walls or changing the riverbed. These one time interventions aim at avoiding an aggradation of the river's shipping route by a constant sediment accumulation [38].

In this use case the capabilities of the AMPI enabled CompAD module of the NAG Fortran Compiler are demonstrated. Moreover, guidelines for MPI user codes should be established that make the activation of the MPI communication seamless. Sisyphé and OpenTELEMAC are parallelized using MPI and potentially run on large-scale cluster systems. A decomposition tool (*Partel*) relying on METIS is used to decompose the input file into single input files for each process. After the simulation has finished, each process will create a result file. These files are merged by the post processing tool *Gretel*.

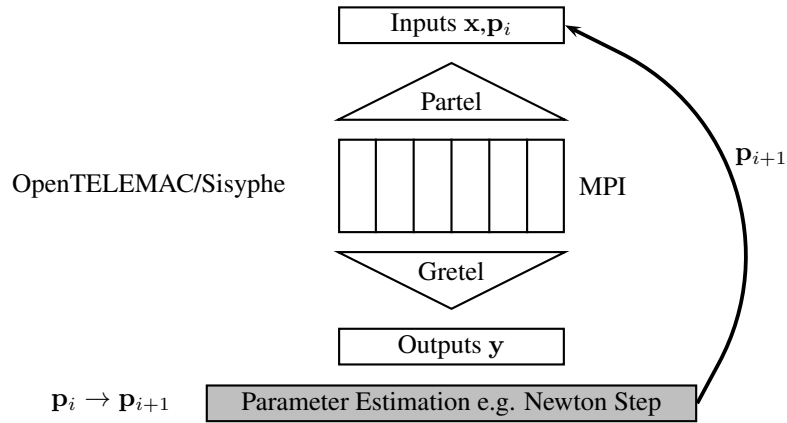
As for any simulation, correct parameters are crucial to achieve robust and valid results (see Section 2.2.1). These parameters are often based on measurements. Adjoints are motivated by an inverse parameter estimation problem with for example the following objective function to be minimized:

$$F_{cost} = \sum_{i=1}^n (y_i - \tilde{y}_i)^2 \quad ,$$

where y are the computed and $\tilde{y}$ the measured values. $y = f(x, p)$ is dependent on some inputs x and parameters p unknown to the observer. For some fixed values x , F_{cost} is minimized by adapting p using for instance Newton's method with the following newton step:

$$p_{i+1} = p_i - (\nabla_p^2 F_{cost}(p, x))^{-1} \cdot \nabla_p F_{cost}(p, x)$$

where $\nabla_p F_{cost}$ and $\nabla_p^2 F_{cost}$ are the Jacobian and the Hessian, respectively for some fixed values x and with respect to p .



It becomes clear that such a parameter estimation requires correct and efficient derivative information. Our goal is to provide this derivative information by applying the CompAD enabled NAG Fortran compiler together with AMPI to Sisyphé. Note that in the context of parameter estimation, the relevant output is the result of the objective function F_{cost} generated out of the outputs of Gretel.

Parallelization Encapsulation

Both Sisyphe and OpenTELEMAC use the same parallel interface. These are located in a separate source tree. Parallelization encapsulation has relieved our effort to adjoin MPI communication tremendously. The initial idea is that all the MPI calls are wrapped in order to provide high level communication patterns. These high level communication calls may then be optimized once at very specific code sections. The side effect of such an implementation is that the MPI calls only need to be activated here. Easy debugging and performance analysis made this code a prime use case for the first use of AMPI in a large simulation code. In Section 5.4 it is described how to deal with no parallel encapsulation as PETSc does not provide one. This is a result of MPI having a very stable interface that has practically never fundamentally changed since its beginnings. Moreover, there is no alternative to MPI in the context of message passing. This unfortunately encourages developers to not encapsulate MPI.

Domain Decomposition

Sisyphe relies on a 2D sediment transport model based on a two dimensional grid. The computation on this grid is achieved through a split that distributes the number of computations evenly. This split creates a non uniform border that consists of the frontier nodes. These nodes make up the Halo zone that is present on all the processes that touch these border nodes. Note that a node may be present on more than one process if that node is at a cross section. The strategy for the domain decomposition does not affect the integration of AMPI in any way. The Halo zones are exchanged by all the involved processes through point-to-point communication in between of two time steps. As has been laid out in 3.1.1, activating MPI communication doubles the number of messages.

However, Sisyphe has some particularities regarding the values of the Halo zone nodes. The Halo exchange is implemented in `./bief/bief_v6p0/SOURCES/paraco.f`. In a common Halo implementation each node's value is calculated by one, and only one, process. Thus, the size of the stencil determines the size of the halo zone. This makes the computation numerically independent from the number of processes. The result are exact same values that may only differ if the MPI communication yields numerical errors. For some unknown reason, Sisyphe has chosen a different path. Here, each process that has a node in its Halo zone computes a new value for that node. This yields different values for the same node on the different processes. The nodes are exchanged and each process then computes a new global value. Sisyphe provides different methods of computing the global value like for example the maximum. In our case, Sisyphe computes the mean value. As opposed to the mean value, the maximum is not even differentiable in the first place. Nonetheless, using the mean gives us different mean values and thus numerically different outputs for varying number of processes. The motivation behind this design choice is not clear to us. It leads however to misconceptions about the validity of the adjoint values, since under some circumstances they seemed more sensitive to this numerical effect than the values.

Driver/Callbacks

In Sisyphe a simulation setup is defined by:

- description in a `.cas` file,
- geometry in a `.geo` file,
- the boundary conditions in a `.conlim` file
- and a Fortran file defining the sediment transport model.

Usually, a differentiated program is subdivided into active code, where the AD enabled types are used, and passive code where no differentiated code is executed. The program starts in passive mode

and switches to the active mode whenever necessary. The code segment in the passive part that calls the active code and generates adjoints by initiating the interpretation is called the *driver*. Introducing this code structure into still passive code may lead to code duplication. If a function is called both in the active and the passive section, it has to be implemented using both types. To avoid code restructuring and because the overhead of the active types is often limited, Sisyphé is compiled using only active types. The *tape* is the data structure used to store the operations and values in the forward section in order to compute the adjoints in the reverse section. With the tape turned off, only the forward section is executed, which is equivalent to the original (*passive*) version of the code. Sisyphé is started with the tape tracing off. The distinction between passive and active code sections is done by turning the tape on or off and not via a driver. When the AD section is entered, the tape is switched on and the independents are set. At the end, before the interpretation, the tape is switched off again and the dependents are set. Later the interpretation is executed. These three spots, tape switch on, tape switch off and tape interpretation are different places in the code. Where these actions take place, an externally defined function call was inserted. The implementation of these callback functions is put in the Fortran file that implements the sediment transport model.

The subroutine `CONDIM_SISYPHE` implements the sediment transport file. This has to be defined, otherwise the linking fails. For AD there are the following three routines. First, `AD_SISYPHE_MAIN_INIT` which is called in the initialization routines of Sisyphé. The CompAD module is initialized here. Before this call takes place, no active operations may be executed. In `AD_SISYPHE_INITIALISATION_END`, the independents are set and the tracing is turned on. Last, the interpretation is started at `AD_SISYPHE_END`, which marks the end of a Sisyphé execution. If we want to compute adjoints after each time step there is a placeholder `AD_SISYPHE_TIMESTEP_END`.

Setup

Our simulation setup models a narrow water channel with a bottom sand topography (see Figure 5.6). The sand has a constant grain size for the entire sediment. In the channel, a sediment elevation is placed with the shape of a wave. The water flows at a constant velocity through the channel. This flow transports the sediment downwards along the stream. We want to model the effect of the grain size on the bottom topography evolution and thus require the computation of sensitivities. The independent is the grain size whereas the bottom evolution is the dependent. The grain size is a scalar value since it applies to all the sediment while the bottom evolution is a vector field with a value for each grid node. With one input and multiple outputs, the tangent-linear model is well suited and has been implemented in a prior project by Jan Riehme. Our goal is a feasibility study of computing gradients using the adjoint model. At the time of this writing there was no well-defined use case for adjoints. However one can easily imagine that different grain sizes for different zones of the bottom topography would be beneficial to the adjoint differentiation model, as the number of inputs steadily increases with the number of zones. However, at the time of this writing, the underlying simulation code did not allow different grain sizes for different topological zones.

5.3.4 Results

The gradient was first computed on the institute cluster *Heisenberg* (see Section 5.1). The slowdown created by the CompAD tape and in combination with AMPI is shown in Figure 5.7 and Figure 5.8. The problem sizes are bound to the available memory and to the minimal problem size. The minimal problem size of the passive computation was set at 1000 timesteps which only yields a runtime of 0.64s in the sequential run and 0.43s in the parallel run with 2 processes. Going below this threshold does not produce any reasonable runtime results. The available RAM sets the maximum problem size to 8000 timesteps. The slowdown created by the CompAD tape alone is around 36 whereas in combination with

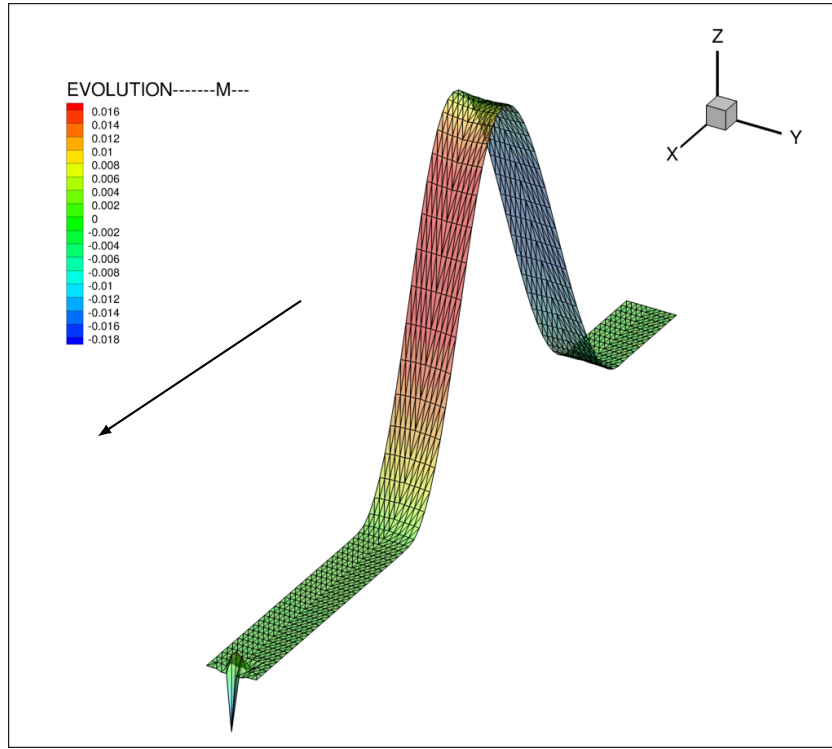


Figure 5.6: Sediment dune after 50 time steps: Change in bottom topography (Evolution)

AMPI this value drops to around 30. Thus, as predicted in Section 3.1.1, AMPI has no detrimental effect on the runtime behavior.

In addition to the gradient a steepest descent algorithm was implemented to motivate the adjoint model. The parameter p that has to be estimated is the grain size. For a given grain size p_m the bottom evolution after 144 timesteps is computed and this final bottom topography is considered as the measured values. The grain size p_m is then perturbed and the perturbed value p_0 is then used as the starting value for a parameter fitting problem. The steepest descent algorithm is then used to optimize the value of p and to see whether it converges to p_m .

The steepest descent algorithm was implemented by the author as a python script that extracts the values of the gradient and inserts the new grain size p_{i+1} into the input file. Note that this optimization step has only one input (grain size) and one output (cost functional). The grain size in Sisyphe is a scalar value and not a scalar field; all grains have the same size. Unfortunately, we had no access to a code that used a zonal distribution of the grain size. Figure 5.9 provides the convergence of a steepest descent run where the measured values were computed with the grain size set to 40. The grain size was then perturbed to 20. Similar tests were conducted yielding the same convergence behavior.

AMPI yields the expected runtime behaviour in Sisyphe. The steepest descent implementation proves the feasibility of a parameter optimization problem in Sisyphe that relies on the parallel computation using AMPI. Future use cases should be investigated that a backed by real world problems.

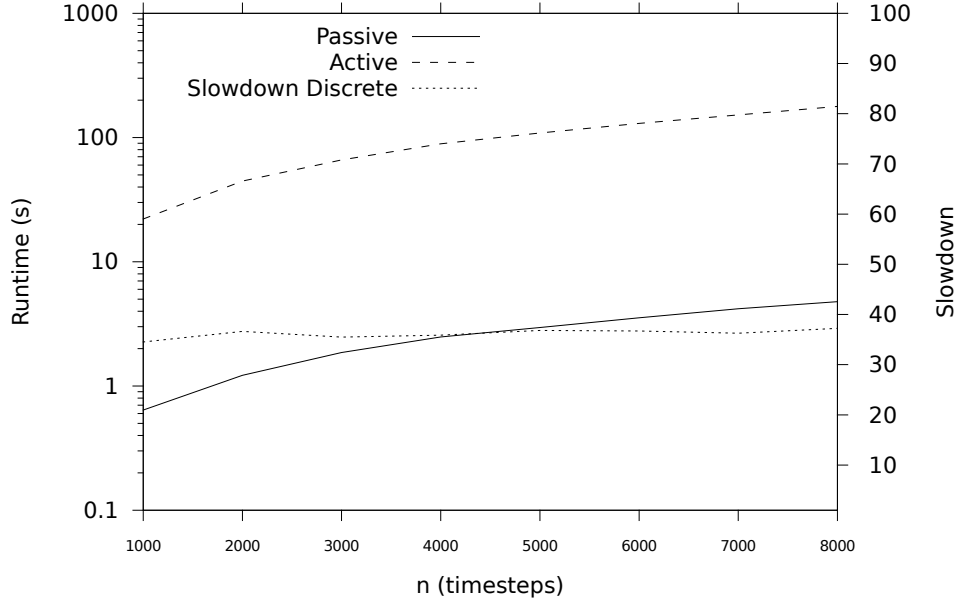


Figure 5.7: Benchmarks on Heisenberg of CompAD and AMPI: 1 process

5.4 Use Case: `dco/c++` and PETSc

At STCE an in-house developed differentiation tool by overloading called `dco/c++` [35] (derivative code by overloading) is developed. It is available upon request to Prof. Naumann at STCE and is developed in close partnership with NAG Ltd. where it is available as a commercial product. It is written in C++ and it is heavily templated, relying on a template engine for a *statement level preaccumulation* of the overloaded datatype. As described in generic programming [1], the adjoint model is applied at the statement level just as for example matrix and vector operations can be unrolled through metaprogramming at compile time. At each assignment the partials of the right-hand side are preaccumulated. The Portable, Extensible Toolkit for Scientific Computation (PETSc) [4] is one of the most widely used frameworks for solving linear and nonlinear equations. Its primary use is in the field of numerical simulations on large cluster systems. It relies heavily on MPI as its parallelization method.

In this section we first illustrate the interface necessary to integrate AMPI into `dco/c++` [36]. Next, an overloaded version of PETSc is presented that relies on the first-order adjoint datatype of `dco/c++`. Last, various benchmarks are presented that illustrate the slowdown, memory footprint and scalability of the AMPI library. In particular, we want to compare the runtime performance and memory consumption of discrete adjoints and continuous adjoints in order to confirm the claim in Section 3.1.1 and Section 3.1.2. According to this, scalability should not be an issue while discrete adjoints are not feasible on current cluster systems due to the adjoint memory wall (see Section 3.1.2). Therefore, AMPI is applied on code where the nonlinear solver is adjoint continuously instead of being differentiated discretely by using AD tools.

5.4.1 Integration into `dco/c++`

The AMPI interface for `dco/c++` is very similar to the one presented for the CompAD tape (see Section 5.3.2). However, no language specific interfaces for Fortran are necessary. Hence, we end up with only one header file that implements the generic interface presented in Section 4.2.1.

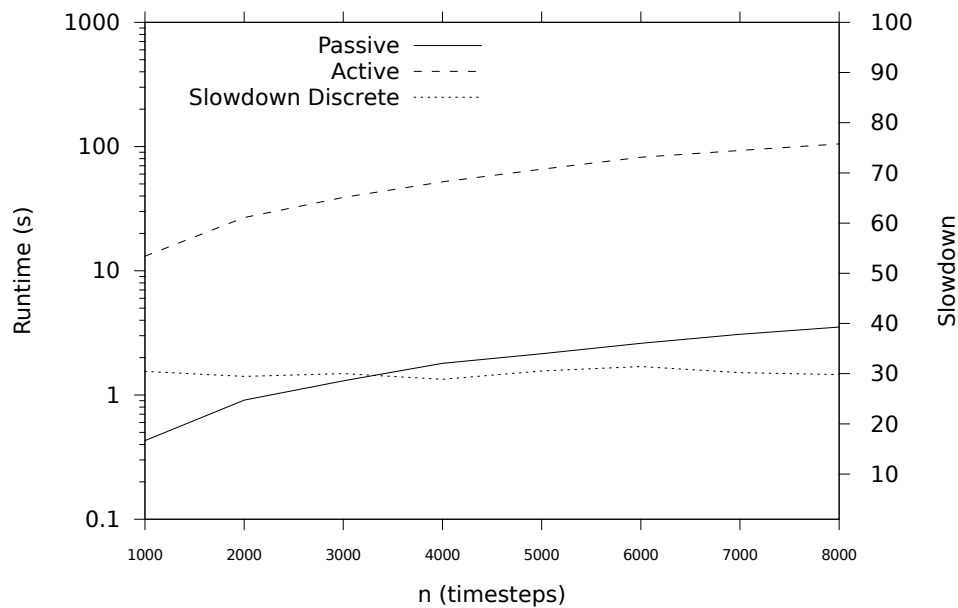


Figure 5.8: Benchmarks on Heisenberg of CompAD and AMPI: 2 processes

```

1 #include "dco.hpp"
2 #include "ampi_tape.hpp"
3
4 using namespace dco::als;
5 extern "C" {
6     void ampi_get_val(void *buf, int *i, double *x) {
7         *x=static_cast<type*>(buf)[*i]._value();
8     }
9     void ampi_set_val(void* buf, int *i, double *v) {
10         type &dummy= static_cast<type*>(buf)[*i];
11         *const_cast<double*>(&(dummy._value())) = *v;
12     }
13     void ampi_get_idx(void *buf, int *i, INT64 *idx) {
14         type &var = static_cast<type*>(buf)[*i];
15         if(!var._data()._is_registered()) *idx=0;
16         else *idx = var._data().tape_index();
17     }
18     void ampi_get_adj(INT64 *idx, double *x) {
19         if(*idx!=0) *x = dco::als::global_tape->_adjoint(*idx);
20     }
21     void ampi_set_adj(INT64 *idx, double *x) {
22         if(*idx!=0) dco::als::global_tape->_adjoint(*idx) += *x;
23     }
24     extern "C" void ampi_reset_entry(long int idx);
25     struct AMPI_data : tape::external_function_base_data {
26         int idx;
27         AMPI_data(const int nidx) : idx(nidx) {}

```

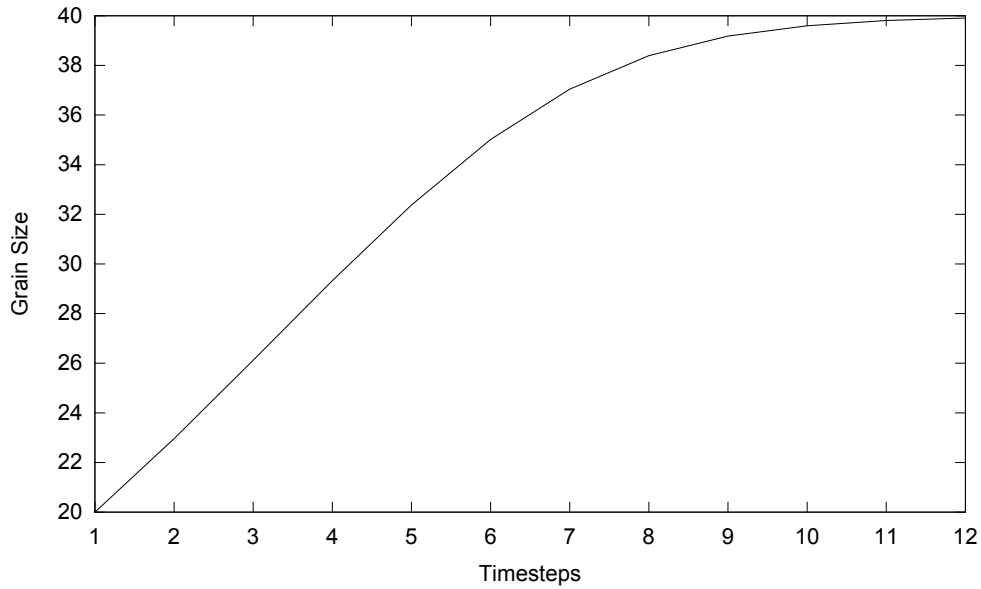


Figure 5.9: Steepest descent convergence in Sisyphe. The objective function is $J(p) = (\mathbf{z}(p) - \mathbf{z}_o)^2$ with n grid points, where $\mathbf{z} \in \mathbb{R}^n$ is the computed sediment elevation and $\mathbf{z}_o \in \mathbb{R}^n$ the observed sediment elevation. The grain size p is a scalar parameter in Sisyphe.

```

28     virtual ~AMPI_data() { mpi_reset_entry(idx); }
29 };
30 void mpi_tape_wrapper(AMPI_data *data) {
31     mpi_interpret_tape(data->idx);
32 }
33 void mpi_create_tape_entry(long int *i) {
34     if(!global_tape->is_active()) {
35         return;
36     }
37     AMPI_data *D = global_tape->create_ext_fcn_data<AMPI_data>(
38         mpi_tape_wrapper, *i);
39 }
40 void mpi_create_dummies(void *buf, int *size) {
41     type *values=static_cast<type*>(buf);
42     for(int i=0;i<*size;++i) {
43         type &dummy=values[i];
44         dummy=0;
45         global_tape->register_variable(dummy);
46     }
47 }
48 int mpi_is_tape_active () {
49     if (NULL != global_tape) return global_tape->is_active();
50     else return 0;
51 }

```

```
51 }
```

Listing 5.7: AMPI dco interface

The type of the generic identifier `INT64` is either set to `int` or `long int` depending on the size of the tape. `ampi_get_val` and `ampi_set_val` access the value component of the overloaded type. `ampi_get_idx` and `ampi_set_idx` access the identifier of the overloaded type inside the tape. To register the AMPI calls in the `dco/c++` tape in `ampi_create_tape_entry` the generic external function interface of `dco/c++` is used. An object data of derived type `AMPI_data` is created that stores the tape index of the AMPI tape in the `dco/c++` tape. Each time this entry is hit in the reverse run, the AMPI interpreter `ampi_interpret_tape` is called through the `ampi_tape_wrapper` together with the restored AMPI tape index. `ampi_reset_entry` deletes the entry `idx` in the AMPI tape. `ampi_is_tape_active` allows AMPI to check whether the `dco/c++` tape is switched on or off. This rather short interface provides full compatibility with `dco/c++` with all the strings attached; it allows restarts, tape switch on and off and more.

5.4.2 Motivation

The case study is the same generic setup as for the textbook example (see Section 2.2.1). The simulation function is based on the two-dimensional Bratu equation,

$$\nabla^2 \mathbf{u} = -\lambda \exp(\mathbf{u}), \quad (5.1)$$

describing a solid fuel ignition with the parameter $0 < \lambda < 6$ and boundary conditions

$$u = b_i \text{ for } x = 0, x = 1, y = 0, y = 1, 0 \leq i < n, \quad ,$$

at the borders of the two dimensional square with n boundary points. For a 4x4 grid, $\mathbf{b}$ is of size 12 while there are only 4 inner points. The Bratu equation is part of the MINPACK-2 test suite [3] as well as an example code of the non-linear solver *SNES* in PETSc. It serves as a code base for our least squares problem. The differential equation in Equation (5.1) is solved by discretization using finite differences on a two-dimensional grid. We do not take a detailed look at the actual implementation in PETSc, but rather take a black box perspective of the code. An optimization problem is formulated over the original solution to the Bratu equation. There are three types of grid points:

- the inner computed grid points $\mathbf{u}$
- the n boundary grid points $\mathbf{b}$ used in the boundary condition
- and a subset $\mathbf{u}^s$ of m observation points of the inner grid points $\mathbf{u}$, for which additional observed values $\mathbf{u}^{ob}$ are assumed to be provided.

The computed values $\mathbf{u}^s$ are a subset of the inner points $\mathbf{u}$ dependent on the boundary conditions. Additionally, there are observed values $\mathbf{u}^{ob}$ that allow us to rate the correctness of the model. This fact may be formulated as a least squares problem where the difference between the computed and observed values ought to be minimized by adapting the approximated or guessed boundary conditions.

$$S = \frac{1}{2} \sum_{i=1}^m (u_i^s - u_i^{ob})^2,$$

The computation of the cost functional S depending on the boundary conditions

$$S = F(\mathbf{b}) : \mathbb{R}^n \rightarrow \mathbb{R}$$

is implemented in PETSc using its non-linear solver SNES for the computation of $\mathbf{u}$. The code found in example 5 of the SNES tutorials is used. The additional implementation of the cost functional S for the least squares problem is straightforward. It follows a step by step description on the implementation of the gradient ∇F of PETSc that enables us to feed a gradient based solution method. The Steepest Descent or Gradient Descent algorithm serves as a proof of concept in order to minimize the residual S . As the name hints, it relies on the gradient to iteratively compute a better fit of the computed observation point values $\mathbf{u}^s$ to the actual observations $\mathbf{u}^{ob}$ by adapting the boundary conditions according to

$$\mathbf{b}_{n+1} = \mathbf{b}_n + \alpha \nabla F(\mathbf{b}_n),$$

where ∇F is the gradient of the aforementioned residual S with respect to the boundary conditions $\mathbf{b}$.

5.4.3 PETSc

As mentioned above, a tutorial example is used as a test case in PETSc. Tutorial 5 of the SNES solver was chosen (`hg/petsc/dco_examples/snes/examples/tutorial`). It implements a solution of the Bratu differential equation Equation (5.1). The source code is available as a mercurial repository on the software disc. The parameter λ was originally set to 6. The value is changed to 1.0 in order to have a more stable system. The other changes are only related to `dco/c++` and will be explained in the next section. PETSc is compiled with the default options, although custom BLAS [33] and LAPACK [2] libraries are provided. The MPI calls in PETSc involving data of type `PetscReal` and `PetscScalar` are all replaced by adjoint MPI calls. The aforementioned MPI calls may be separated into two types. For one there are several collective invocations of `Allreduce`. The other one is the persistent MPI communication in `src/vec/vec/utils/vpstat.c`, another form of nonblocking communication described in Section 3.4.3.

Adjoint BLAS and LAPACK

PETSc requires a BLAS and LAPACK library to be linked to the main executable. They provide all the fundamental numerical operations in PETSc. The first BLAS and LAPACK implementations were introduced in 1979 [33] and 1999 [2], respectively. They still represent the basic blocks of modern simulation codes. BLAS and LAPACK are written in either Fortran or C (`cblas`, `clapack`) and PETSc assumes external linking with no type checking. Specialized BLAS and LAPACK libraries rely on the same interface while exploiting hardware specific performance enhancements. Using the overloaded active types `dco::als::type` of `dco/c++` forces us to leave the domain of C and move to C++. The f2c BLAS and LAPACK version 3.1.1.1 has been overloaded with the `dco::als::type` and is available on the disc in `git/dco_blaslapack`. A few amendments had to be made to make the C code compile with a C++ compiler. In particular due to C unions, a C++11 enabled C++ compiler has to be used (here `gcc 4.7.2`).

Adjoint MPI and PETSc

The MPI calls in our PETSc implementation were tracked using a custom MPI header in order to pinpoint the MPI calls that need to be adjointed. In summary, there were two types of MPI communication. First, persistent communication consisting of an `MPI_Send/Recv_init`, `MPI_Start`, `MPI_Wait` and `MPI_Request_free` (see Section 3.4.3). And second collective communication in the shape of an `MPI_Allreduce`. The source repository with all the code amendments is available on the disc in a mercurial repository (`hg/petsc`) which was cloned from the official PETSc repository in order to make merge backs as easy as possible.

5.4.4 Continuous Nonlinear Solvers

This section is taken from the work done in [47]. It summarizes the process of deriving the adjoint model of a nonlinear solver and how the memory footprint of the continuous solver is considerably reduced compared to a discrete differentiation of the nonlinear solver using an AD tool.

Given a solver of parametrized systems of nonlinear equations:

$$\mathbf{r} = F(\mathbf{x}, \boldsymbol{\lambda}) : \mathbb{R}^n \rightarrow \mathbb{R}^m \quad .$$

For $\boldsymbol{\lambda}$, we want to find a vector $\mathbf{x} = \mathbf{x}(\boldsymbol{\lambda})$ such that $F(\mathbf{x}, \boldsymbol{\lambda}) = 0$. Differentiation of $F(\mathbf{x}, \boldsymbol{\lambda}) = 0$ at the solution $\mathbf{x} = \mathbf{x}^*$ with respect to $\boldsymbol{\lambda}$ yields

$$\frac{dF}{d\boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) = \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda}) + \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda}) \cdot \frac{\partial \mathbf{x}}{\partial \boldsymbol{\lambda}} = 0 \quad .$$

According to the adjoint model, they then derive the adjoint statement in order to compute

$$\boldsymbol{\lambda}_{(1)} := \boldsymbol{\lambda}_{(1)} - \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^{-T} \cdot \mathbf{x}_{(1)} \quad .$$

The transposed inverse of $\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})$ is computed using PETSc, because the Jacobian has to be set in this example by the user during the initialization. So the system

$$\frac{\partial F}{\partial \mathbf{x}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \mathbf{z} = -\mathbf{x}_{(1)}$$

is solved. Finally, in order to compute $\boldsymbol{\lambda}_{(1)}$, the adjoint model of F is used to obtain:

$$\boldsymbol{\lambda}_{(1)} = \boldsymbol{\lambda}_{(1)} + \frac{\partial F}{\partial \boldsymbol{\lambda}}(\mathbf{x}, \boldsymbol{\lambda})^T \cdot \mathbf{z}$$

In matrix-free solvers, the adjoint model of F has to be called once per iteration. This yields a far smaller memory footprint than adjoining the entire nonlinear solver, namely only one computation of the adjoint mode of F as opposed to the entire nonlinear solver.

5.4.5 Results

The benchmarks were conducted in part on our specialized AD workstation (Heisenberg) as well as on the RWTH cluster. First, the outcome should be a confirmation of the results in the Sisyph case study (see Section 5.3.3). It showed that the slowdown created by AMPI should be negligible with no scalability penalty in the adjoint code when compared to the original code.

Heisenberg is used to compare a single process versus a dual process setup. Again, there is a minimal problem size defined by the smallest runtime while running PETSc passively without AD amounting to 10000 grid points. The maximum size of the tape that Heisenberg can cope with is a tape size created by 90000 grid points. To compute bigger test cases one has two choices. Either apply a checkpointing scheme [20] or replace the computationally most expensive part, namely the nonlinear solver altogether (see Section 3.1.2). Replacing the nonlinear solver SNES in PETSc with a continuously differentiated version of SNES is a work done by Johannes Lotz at STCE. He is comparing the benefits and drawbacks of discrete and continuous differentiation and uses dco/c++ to implement different ways of combining the two. Consequently, combining discrete and continuous adjoints in this use case was done using dco/c++ and its external function interface. Instead of differentiating the nonlinear solver of PETSc discretely, statement by statement, it is instead treated as an intrinsic function that is adjointed mathematically and then implemented accordingly. The outcome of this research is presented in [47]. The switch between

active and passive code segments is again done the same way as in Sisyphe (see Section 5.3.3). While using continuous adjoints of the linear solver, the operations are most of the time not traced and the tape is switched off. As described before Section 5.4.4, discrete adjoints are only used to compute one single matrix-vector product. This enabled us to compute considerably bigger problem sizes.

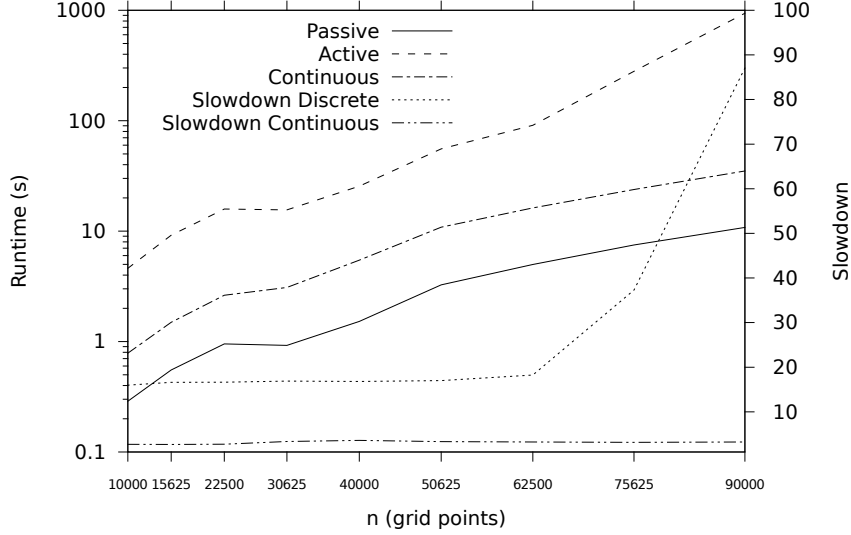


Figure 5.10: Benchmarks on Heisenberg of PETSc with `dco/c++` and AMPI: Slowdown caused by `dco/c++` using one process

The benchmarks on Heisenberg confirm the prior results with Sisyphe (see Figure 5.10 and Figure 5.11). The slowdown of `dco/c++` is generally lower than the one of CompAD. As anticipated in Section 3.1.1 a faster AD tool diminishes the difference in the slowdown between the AMPI enabled code using two processes and the one with only `dco/c++`. The surges of the slowdowns at big problem sizes is due to the missing RAM and the swap space that is used. The slowdown of the continuously differentiated non linear solver is much lower yielding a value of around 2.6 for the sequential as for the parallel AMPI enabled run. Again, there is no additional slowdown due to AMPI.

In Figure 5.12 the global tape size of the parallel computations is compared to the sequential computation. Since a communication is equivalent to an assignment, the communication has to be traced and thus the memory requirement for the adjoint computation increases. The overhead created by the communication is between 80% and 169%. Note that exactly the same numerical results are computed.

On the RWTH Compute Cluster the picture is more ambiguous (see Figure 5.13). The slowdown of `dco/c++` varies a lot with two spikes. One with 1 process at 21 and the other one with 8 processes at 37. Fortunately, with the spike taking place with 1 process, AMPI can be excluded as the main cause. The slowdown with 4, 16 and 32 processes evolve around 7.5 to 9.8. The spikes might be created by cache effects since they were reproducible at every run. In the end, there is no indication of a detrimental effect of AMPI on the runtime performance.

It was proven that the combination of `dco/c++` and Adjoint MPI is robust enough to compute semi-automatic discrete adjoints of PETSc. Three distinct steps were necessary. First, `dco/c++` had to be applied to PETSc by overloading all variables of type `PetscReal` and `PetscScalar`. Second, BLAS and LAPACK were adjointed using again `dco/c++`. Finally, the adjoint MPI library was used to adjoint all the MPI communication.

The PETSc benchmarks confirm what has been claimed in Section 3.1.1 and Section 3.1.2. The runtimes show no detrimental effect caused by the AMPI library; the slowdown even drops from to

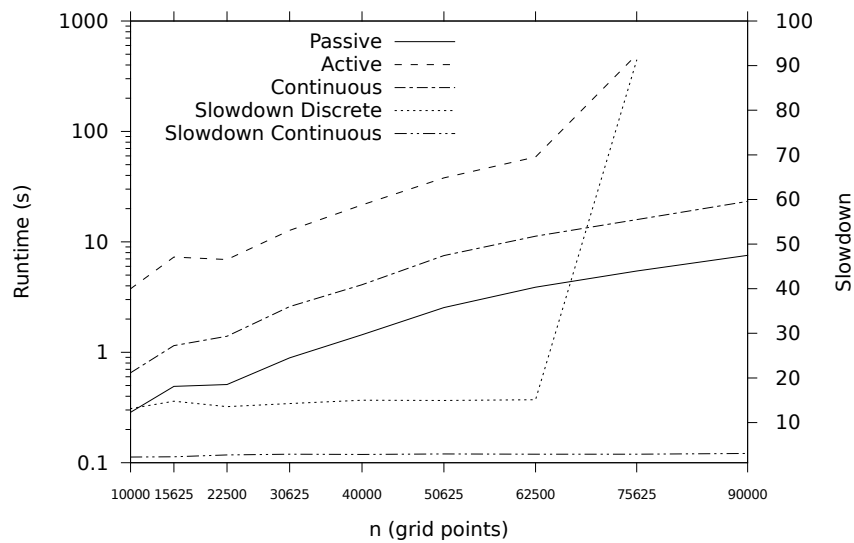


Figure 5.11: Benchmarks on Heisenberg of PETSc with `dco/c++` and AMPI: Slowdown caused by `dco/c++` and AMPI using two processes

sequential to the parallel test case. However, as PETSc is a very efficient code, the memory consumption becomes a huge issue when differentiating in full discrete mode. The remedy was the implementation of a continuously differentiated nonlinear solver as described in Section 5.4.4. This diminishes the memory footprint considerably and yields the same slowdown for the sequential as for the parallel version.

The increased need for continuous adjoints in high performance computing is obvious. The general structure of a code should be that a computational intensive core is adjointed continuously while the surrounding data handling and setup is adjointed discretely. Unfortunately, an adjoint of the underlying simulation model is not always available or even feasible. Here, discrete adjoints remain the only available method, however feasibility remains an issue due to the adjoint memory wall.

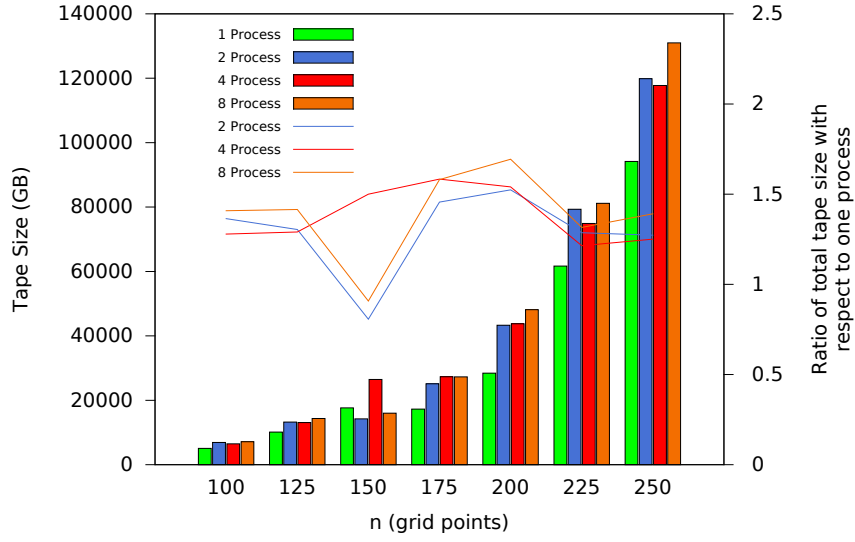


Figure 5.12: Memory footprint on Heisenberg of PETSc with `dco/c++` and AMPI for various number of processes. n^2 defines the problem size, where n is the number of grid points. The bars represent the absolute memory footprint while the lines are the ratio of the total tape size over all processes with respect to the tape size using a single process.

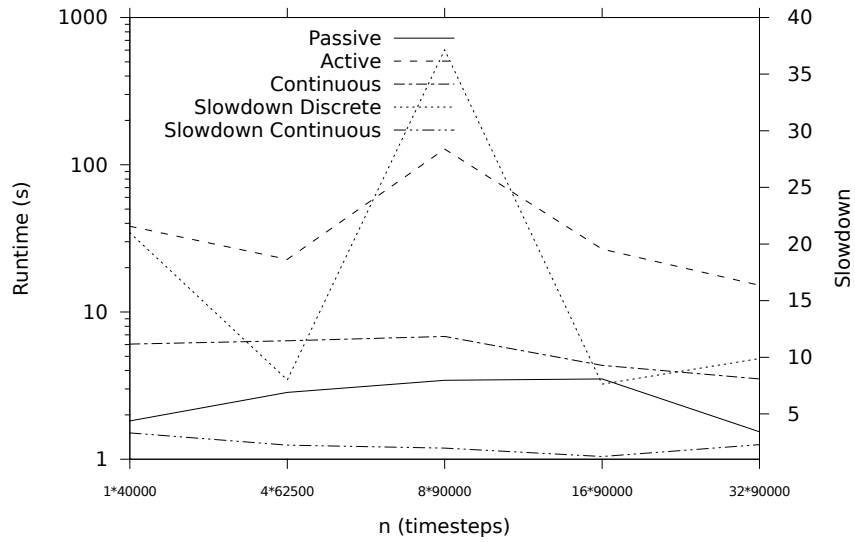


Figure 5.13: Benchmarks on the RWTH Compute Cluster of PETSc with `dco/c++` and AMPI: Slowdown caused by `dco/c++` and AMPI with different number of processes (1,4,8,16,32) and problem sizes

Chapter 6

Discussion

The method of semantics driven adjoints for MPI is based on heuristics and interpretation of human language. However, the same is true for the MPI standard itself. A grammar driven, compiler based interpretation of the MPI standard and thus the MPI calls is not possible. The generation of adjoint communication patterns in the PGAS extended SAC proved to be very effective especially for the one-sided communication. The adjoint pattern description often revealed shortcomings of intuitively derived patterns. It allowed for the first time to systematically derive robust adjoint patterns of one-sided communication.

The first striking feature of adjoining MPI enabled programs is the tremendous memory requirement of the tracing or taping in the forward section (Section 3.1.2). Memory is already a huge issue on parallel systems. The memory requirement of nearly one location per operation in the original code leads to confusion about how to realize this in an exascale, petascale or even in the most basic parallel environments. However, there is no more efficient way of computing gradients with a small number of outputs m and a high number of inputs n than using adjoints due to the runtime complexity of $\mathcal{O}(m) \cdot \text{cost}(F)$ for a gradient computation. Moreover, given suitable AD tools, discrete adjoints are easy to maintain alongside the original code. The only alternative are continuous adjoints which, even if they exist, are harder to maintain.

The general issues with discrete adjoints are worsened by using AMPI on cluster systems. If continuous adjoints exist, they are often the preferred solution for computationally expensive codes (see Section 3.1.2). This setup was shown in Section 5.4 using PETSc. The case studies emphasize the need for a combination of discrete and continuous adjoints. AMPI is useful to differentiate code that has a rather low computational complexity in the original code, but reaches its limits when the computational complexity is high. This is not due to AMPI per se, but the very nature of adjoints and the hardware of cluster systems.

An execution of a numerical code may be displayed as a cake diagram where work is distributed among processes that interact with another through a network or some other link (dashed red lines). The computation starts on the outside and proceeds to the inner core of the computation. On the outer layer (yellow), the initialization, finalization and IO is taking place. On the inner layer (green) represents the numerical core (e.g. nonlinear solver). It potentially has a higher time and memory complexity than the outer layers. The numerical core is where checkpointing techniques (see Section 3.1.2) is probably applied in order to reduce the memory complexity when running in adjoint mode. Parallelism is increased because at this level shared-memory models (e.g. OpenMP) are joining the distributed parallelism. The work is split up among threads (dashed red lines). This is where the work of [16] is situated. This work claims that there is strong evidence that adjoint code of the inner core does not scale on current cluster systems. If possible, this inner core should be adjoint using continuous adjoints. The use case for AMPI

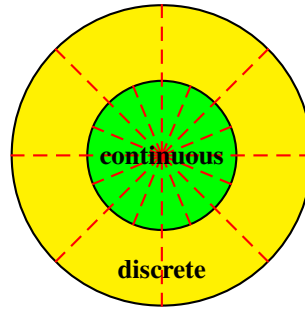


Figure 6.1: Use case for AMPI

(see Figure 6.1) and for discrete adjoints is definitely on the outer layers. Where the exact frontier between the outer and inner layer is, is still an open for debate. It is clear that this frontier is linked to the time complexity of the original code (see Section 3.1.2).

Finally, the feasibility of adjoint MPI patterns has been proven to be not always given. The accumulation (`MPI_Accumulate`) together with the product operation (`MPI_PROD`) is the first MPI communication that has been proven to be not adjointable. Only a shortcut version is possible. Due to the large number of newly introduced one-sided MPI calls in MPI-3.0 it is highly probable that more non adjointable MPI communications exist. The access restriction axiom is a try at formulating guidelines for developers that allows them to safely write adjointable MPI code.

Chapter 7

Summary and Conclusion

A theoretical framework has been established to link MPI code execution with the Algorithmic Differentiation (AD) formalism. The Partitioned Global Address Space (PGAS) notation is merged with the Single Assignment Code (SAC) abstraction in AD. The SAC notation allows to apply mathematical models to a programming code. The PGAS notation introduces MPI communication into the SAC and allows us to apply the AD differentiation models on parallel MPI enabled codes.

First, common observations are formulated in order to estimate the runtime behavior and memory requirement of adjointed MPI programs on current cluster systems. In general, adjointed code should scale as well or even better than the original; under the condition that there is enough memory. However, on the memory side the situation is rather grim, because the adjoint memory wall is only worsened on cluster machines. The available memory per node is not proportional to the number of nodes. So even a linear scalable original code will not scale as well after it has been adjointed. This runtime hit may currently only be faced with continuous adjoints.

The most common point-to-point, collective and one-sided MPI communications are expressed via the PGAS extended SAC notation. AD is applied to these communication patterns in order to extract a description of the adjoint communication patterns. The nonblocking point-to-point and one-sided communications require additional constructs to express the access restrictions of buffers. These access restrictions are called *locks* and are directly conveyed to the adjoint pattern description. The collective patterns are black box communications where the internal communication structure varies between MPI implementations. In particular the reduction has been proven difficult due to the additional operation on the communicated buffers. For each adjoint pattern a slowdown factor with respect to the original pattern was deduced and verified via a prototyped implementation. Due to the extend of the MPI standard a repetitive and complete coverage was dismissed in favor of a focus on the different groups of MPI communication (point-to-point, collective, one-sided) and on the challenging parts of the MPI 2.2 standard.

In the end, closure and feasibility of nearly all MPI 2.2 communications is given. However, the one-sided accumulation `MPI_Accumulate` together with the product operation `MPI_PROD` has been proven to be not adjointable with one-sided communication. It is the only communication in the MPI 2.2 standard where no efficient workaround could be found. However, this is exactly the area where most of the progress takes place in the MPI 3.0 standard. One-sided communication is a huge issue since it is not symmetric anymore. The safest recommendation for AD users would be to avoid one-sided communication altogether. Because this is a rather harsh restriction, a new guideline was proposed which forces certain constraints on the user. The user is prohibited concurrent read/read and read/write access of an MPI buffer inside an access epoch. In MPI, only concurrent read/write access is forbidden. However, this does not solve the problem with non adjointable communications like the accumulation.

If fully discrete adjoints are doable, the adjoint MPI library (AMPI) proves to be a robust and efficient way to extend the current AD tools that already cover C, C++ and Fortran. Applied to PETSc, AMPI was deployed for its prime use case. The computational kernel was adjoint continuously while all the outer data handling and initialization was differentiated discretely in conjunction with AMPI. This use case should serve as an illustrative guideline of applying AMPI.

Chapter 8

Future Work

With the MPI 3.0 standard just being released, MPI is becoming a dynamic standard again. It evolves constantly and adapts to the current needs of developers on large cluster systems. This work covered the MPI 2.2 standard and gave insights into current use cases for adjoint MPI that are possible today. A lot of debate is going on about the future of parallel programming where all the paradigms, message passing, shared-memory and accelerator, compete. Eventually, MPI will also change and adapt.

MPI 3.0 introduced nonblocking collectives which mark a huge gain for developers. Instead of reimplementing them using nonblocking point-to-point communication, they can now rely on the standard itself to provide an efficient solution. They do not pose any challenge to the generation of adjoint communication and may be handled similarly to the nonblocking communication. The other big change in MPI 3.0 is the extension of one-sided communication. It was concluded, that the one-sided interface of MPI 2.0 was a reason for its rather limited use in practice. It was often perceived as too cumbersome and too complex. As expected by a robust standard, MPI 3.0 stays backwards compatible to MPI 2.0 and essentially introduces new one-sided communication calls. Their effects on developers is still subject to a lot of debate inside the community. As an AD tool developer things have not become easier. MPI 3.0 has more options, more features, more detail concerning one-sided communication. And every additional option has potentially huge drawbacks when generating adjoints, due to the read and write reversal (see Section 2.6.2). A simple, small interface for one-sided communication is currently out of reach. The impact of the AD community on MPI is too small to expect any consideration with regard to adjoints. As it stands today, the focus in parallel programming is on specialization rather than generalization.

The task for future AMPI users and developers will revolve around one-sided communication. That there will be features that may not be supported when computing adjoints has been claimed in this work. To extract the features that may work with adjoints is one, if not the important step. There has to be a clear guideline for AD tool users of *dos and don'ts* that are simple and clear and that they may rely on. Something that leads to bad performance and frustration is not a working solution and will not draw new users to AD. Giving users a clear guideline should allow them to adapt their code as effortless as possible.

Another open issue is that of adjoints on high performance clusters. The fact that the memory per node steadily decreases will be a huge challenge for the AD community as a whole. The adjoint model is just not avoidable for certain applications as it is sometimes the only feasible solution that enables the computation of the desired derivative information. It is not a question whether adjoints will be computed using parallel programming, but how well the generation performs on general purpose computer clusters. Another solution could be to design special hardware for the generation of adjoints that keep the cost of hardware below the one of generic clusters. For a fully discrete solution using AMPI, even when resorting to checkpoints, memory has to become less sparse.

Appendix A

User Documentation

Adjoint MPI is an open source library written in C aimed at adjoining MPI communication. Its main purpose is in the field of Algorithmic Differentiation, but it may very well have different applications for communication reversal as the adjoint increment operation can easily be replaced. The library covers a subset of the MPI standard with the communications in Table A.1 and comes with C and Fortran interfaces.

MPI Call	Covered
Point-to-Point	
MPI_Send, MPI_Recv	✓
MPI_Bsend, MPI_Rsend, MPI_Ssend	-
MPI_Isend, MPI_Irecv, MPI_Wait	✓
MPI_Sendrecv_replace	✓
MPI_Start	✓
Collective	
MPI_Barrier	✓
MPI_Bcast	✓
MPI_Gather, MPI_Scatter	✓
MPI_Gatherv, MPI_Scatterv	✓
MPI_Allgather, MPI_Allgatherv	-
MPI_Alltoall, MPI_Alltoallw	-
MPI_Reduce, MPI_Allreduce	✓
MPI_Reduce_Scatter	-
MPI_Scan, MPI_Exscan	-
MPI_Start, MPI_Recv_init, MPI_Send_init	✓

Table A.1: Commonly used MPI calls and those supported by the Adjoint MPI library

The library provides a transparent way of adjoining MPI by focusing on the ease of use rather than completeness or performance. We distinguish between two types of users. First, the AD tool developer who wants to integrate the AMPI library and thus enables his or her tool to differentiate MPI parallelized code. Second, there is the user who uses the AD tool, which relies on the AMPI library in order to differentiate his MPI code. We cover both perspectives and provide a documentation for both use cases.

A.1 AD Tool Developer

The AD tool developer has to provide an implementation of the interface routines that allow the AMPI library to exchange data with the AD tool. We assume that the developer is familiar with general internals of overloading and source transformation AD tools. Since AMPI does not handle the data, it has to assume a data abstraction that is common to all AD tools. One has to distinguish between overloading and source transformation tools.

With overloading tools one common element is a pointer of type `void*` to the memory location of the values. AMPI does not need a specific type for the active data. It only needs the memory location where the data element is stored. The second element is that every value has to be provided with a location where the corresponding adjoint in the reverse section is stored. This is called the index `idx` which is of type `INT64` and it establishes a link between the value and the adjoint. `INT64` is set through a preprocessor macro for both the AD tool and AMPI.

As we aim for ease of use, we tried to keep the number of interface routines as small as possible. We ended up with the following 7 routines. Since MPI deals with buffers as arrays, we allow to access the data elements with a base pointer `buf` and an offset `*i`. There are only pointers in the signature in order to avoid copies.

```
1 /* Get a value *v of a specific tape variable buf[i] */
2 extern void ampi_get_val(void* buf, int* i, double* v);
```

The value of the active data element at the base pointer `buf` with offset `*i` is copied into `*v`.

```
1 /* Set a value *v of a specific tape variable buf[i] */
2 extern void ampi_set_val(void* buf, int* i, double* v);
```

The value in `*v` is copied to the value of the active data element `buf` with offset `*i`.

```
1 /* Get an adjoint *a of a specific tape entry with index *idx */
2 extern void ampi_get_adj(INT64* idx, double* a);
```

The adjoint with index `*idx` is copied into `*a`.

```
1 /* Set an adjoint *a of a specific tape entry with index *idx */
2 extern void ampi_set_adj(INT64* idx, double* a);
```

The value in `*a` is copied into the adjoint with index `*idx`.

```
1 /* Get a tape index of a variable buf[i] */
2 extern void ampi_get_idx(void* buf, int* i, INT64* idx);
```

The index of the active data element with base pointer `buf` and offset `*i` is copied into `*idx`.

```
1 /* Create a tape entry in the external tape, indicating an external
   AMPI call */
2 extern void ampi_create_tape_entry(long int *mpi_opcode);
```

Create a tape entry on the trace of the AD tool that marks an MPI call in the forward section. This is used to trigger the corresponding adjoint MPI call in the reverse section. Optionally, the name of the MPI routine may be given through an opcode argument `*mpi_code`. This may be useful if the AD tool needs to know which MPI routine was called (e.g. tape consistency checks).

```
1 /* Create size tape entries to store the values of buf. Refer to
   receive buffer
   without initialization */
3 extern void ampi_create_dummies(void* buf, int *size);
```

This routine allows AMPI to allocate a number `size` of elements on the tape of the AD tools in order to store the value, adjoint and index. This is in particular used by receiving MPI calls that overwrite local data.

Implementing these 7 routines is referred to as the AMPI interface needed by the overloading AD tool in order to differentiate MPI code. All AMPI routines in the code are directly linked to the AMPI library.

In the case of source transformation the tool has to reverse the MPI function calls itself. We assume that in this case, the buffers of the values and the adjoints are all of type `double`. In essence every MPI communication function has a forward and a reverse counterpart. The forward call is in the forward section and has for example in the case of a blocking send `MPI_Send` the following signature:

```
int AMPI_Send_f(double *buf, int count, MPI_Datatype datatype, int
    dest, int tag, MPI_Comm comm);
```

The reverse call has the following signature

```
int AMPI_Send_b(double *buf, int count, MPI_Datatype datatype, int
    src, int tag, MPI_Comm comm);
```

The function calls are prepended by an `A` while we distinguish between the forward and the reverse call through the suffix `_f` and `_b`, respectively. The source transformation tool has to provide the reversal logic by those suffixes. In the forward call, the values are handed over while in the reverse call the adjoints are placed in the buffer argument. Obviously, the source transformation tool may differentiate function signatures in a different way. In that case a wrapper has to be provided. Please take a look at the second-order use case with the source transformation tool `dco` (see Section 4.3.2).

Please take a look at the next section for further documentation on using the AMPI library from an AD Tool user's perspective.

A.2 AD Tool User

Please refer to the AD tool documentation for activating a given sequential code. This documentation covers only the MPI calls and how to build and link the AMPI library in a C/C++ setting. AMPI comes with a Fortran interface that was used in the CompAD use case in Section 5.3.2.

A.2.1 Installation

Extract the downloaded and zipped tarball:

```
tar xvzf libampi-1.0.tar.gz
```

Or clone the mercurial repository:

```
hg clone AdjointMPI.hg ./AdjointMPI
```

An extracted version of AMPI is available on the CD. The structure of the directory is as follows:

- `src`, `include`: source and header files for the build process
- `examples`: small example codes
- `interfaces`: interfaces for various AD tools
- `test`: testing suite tailored at running with `dco/c++`.
- `patterns`: pattern testing suite for various adjoint MPI patterns, no library code is used here

Run the configure script and make:

```
./configure
make
```

You will have the AMPI library in `./src/libAMPI.a`. Notice that at this point there is no link to any AD tool. Congratulations, you have build the AMPI library.

To use the testing suite you must have a working `dco/c++` installation and the following environment variables have to be defined:

- `$DCO_BASE_DIR`: path to the `dco/c++` base directory
- `AMPI_DIR`: path to the AMPI base directory

Running make in `./test/` should yield no errors.

The patterns in `./patterns/` mimick the actual implementations of the adjoint communication patterns. They do not rely on the library itself. The adjoint patterns for one-sided communication have been implemented in the context of the Adjointable MPI library (see Chapter 4).

A.2.2 Usage

If a source transformation tool is used, there is no need for further code adaptations as the tool should handle the necessary amendments. Only the linker flags have to be considered. For overloading tools, the MPI header `#include <mpi.h>` needs to be replaced by `#include "ampi_tape.h"` for C code and `#include "ampi_tape.hpp"` for C++ code. The `MPI_Init` and `MPI_Finalize` routines have to be replaced with their AMPI counterpart `AMPI_Init` and `AMPI_Finalize`. On top of that, one only has to deal with the communication routines. All the other MPI routines, types and variables are left unchanged. Thus the prefix of communication routines need to be changed from `MPI_` to `AMPI_`. In order to compile your code you need three ingredients:

- A compiled AMPI library `libAMPI.a`,
- an interface to your AD tool `ampi_interface.cpp`
- and adapt the compile flags.

The compile flags for the interface and the code are `-I$ (AMPI_DIR)` while in the linking step one has to add the AMPI library `$(AMPI_DIR)/src/libAMPI.a`. Your differentiated code should now run seamlessly while AMPI is dealing with the adjoining of the MPI calls.

A.2.3 Fortran

As AMPI is only written in C, the MPI Fortran Interface is not used in case of a differentiated MPI Fortran code. Instead the AMPI Fortran calls are mapped on the AMPI C interface through a wrapper. Three additional files are provided with AMPI that implement such a wrapper.

- `ampi_fortran_interface.F90`,
- `ampi_fortran_stubs.c` and `ampi_fortran_stubs.h`.

As the name hints, `ampi_fortran_interface.F90` implements the AMPI Fortran interface. Notice that the MPI Fortran and MPI C interface do differ. For example the error codes are the return values in C, whereas they are the last argument in Fortran. The MPI request are structures in C and integers in Fortran. There are other differences that are explained in detail in the MPI standard. Here, the AMPI Fortran interface is equivalent to the MPI Fortran interface. All of this is dealt with on the C side in `ampi_fortran_stubs.c` in order to convert the AMPI Fortran calls to AMPI C calls.

A.2.4 Example

In this example we activate a code that implements two send and receive pairs. A value x is squared and then sent from process 0 to process 1 where $\sin(x)$ is computed and sent back from process 1 to process 0. Finally, all the buffer elements on process 0 are summed up or reduced to one value.

```

1 #include <iostream>
2 #include <cmath>
3 #include <cstdlib>
4 #include <mpi.h>
5
6 using namespace std;
7
8 void comp(double *x, double &y, int &n) {
9     int rank;
10    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
11    double *buf=new double[n];
12    y=0;
13    if(rank==0) {
14        for(int i=0;i<n;i++) x[i]=x[i]*x[i];
15        MPI_Send(x,n,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
16        MPI_Recv(buf,n,MPI_DOUBLE,1,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
17        for(int i=0;i<n;i++) y+=buf[i];
18    }
19    if(rank==1) {
20        MPI_Recv(buf,n,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
21        for(int i=0;i<n;i++) buf[i]=sin(buf[i]);
22        MPI_Send(buf,n,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
23    }
24    delete [] buf;
25 }
26
27
28 int main(int argc, char *argv[]) {
29     MPI_Init(&argc, &argv);
30     int numprocs;
31     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
32     int n=atoi(argv[1]);
33     cout << "Problem size: " << n << endl;
34     double *x=new double[n];
35     double y=0;
36     for(int i=0;i<n;i++) x[i]=(double) i;
37     comp(x,y,n);
38     cout << "Result:" << y << endl;
39     MPI_Finalize();
40     delete [] x;
41     return 0;
42 }

```

In `main()` a variable `n` is read in as a command line argument. It defines the size of the MPI buffers. The input `x` is set to `x[i]=(double) i`; and the `comp(x, y, n)` function is called. The MPI functions

`MPI_Init`, `MPI_Finalize` and `MPI_Comm_size` should be known to any MPI programmer.

In `comp(x, y, n)` the values of the array `x` are first squared on process 0 and then sent from process 0 to process 1. Here, the sine of each buffer element is computed (`buf[i]=sin(buf[i])`); and the result is being sent back from process 1 to process 0. On process 0 the buffer elements are reduced with a sum to the single value `y` (`for(int i=0; i<n; i++) y+=buf[i]`);

We now want to compute the gradient of `y` with respect to all the inputs in `x`. Any AD tool can be applied, here we use `dco/c++`. However, the focus is on the MPI routines and how they need to be changed.

```

1 #include <iostream>
2 #include <cmath>
3 #include <cstdlib>
4 #include "dco.hpp"
5 #include "ampi_tape.hpp"
6
7 using namespace std;
8
9 typedef dco::gals<double>::type active;
10
11 void comp(active *x, active &y, int &n) {
12     int rank;
13     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
14     active *buf=new active[n];
15     y=0;
16     if(rank==0) {
17         for(int i=0; i<n; i++) x[i]=x[i]*x[i];
18         AMPI_Send(x, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
19         AMPI_Recv(buf, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
20         ;
21         for(int i=0; i<n; i++) {
22             y+=buf[i];
23         }
24     }
25     if(rank==1) {
26         AMPI_Recv(buf, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE)
27         ;
28         for(int i=0; i<n; i++) {
29             buf[i]=sin(buf[i]);
30         }
31         AMPI_Send(buf, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
32     }
33     delete [] buf;
34 }
35
36 int main(int argc, char *argv[]) {
37     AMPI_Init(&argc, &argv);
38     dco::gals<double>::global_tape = dco::gals<double>::tape::create(1
39         e4);
40     int numprocs, rank;

```

```

39 MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
40 MPI_Comm_rank(MPI_COMM_WORLD, &rank);
41 int n=atoi(argv[1]);
42 cout << "Problem size: " << n << endl;
43 active *x=new active[n];
44 active y=0;
45 for(int i=0;i<n;i++) x[i]=(active) i;
46 if(rank==0) {
47     for(int i=0;i<n;i++)
48         dco::gals<double>::global_tape->register_variable(x[i]);
49 }
50 active *x_saved=new active[n];
51 for(int i=0;i<n;i++) x_saved[i]=x[i];
52 comp(x,y,n);
53 cout << "Result:" << y << endl;
54 cout << "Derivatives:" << endl;
55 if(rank == 0) dco::gals<double>::set(y, 1., -1);
56 if(rank == 1) dco::gals<double>::set(y, 0., -1);
57 dco::gals<double>::global_tape->interpret_adjoint();
58 //if(rank == 0) {
59     double g=0;
60     for(int i=0;i<n;i++) {
61         dco::gals<double>::get(x_saved[i], g, -1);
62         cout << g << endl;
63     }
64     //}
65
66 AMPI_Finalize();
67 delete [] x; delete [] x_saved;
68 return 0;
69 }

```

Listing A.1: AMPI example

First, an additional header `ampi_interface.hpp` has to be included. We assume a generic active type `active` that is set to the `dco/c++` adjoint type `dco::gals<double>::type`. In `main()` the AD tool specific calls should not be described. Any AD developer will however recognize the setting of independents (48) and of the dependents (line 55) followed by a call of the tape interpreter (line 57). Notice that the seeding and setting of the independents have to be both only be executed on process 0. Only the `MPI_Init` and `MPI_Finalize` calls have to be prefixed by an `A` in line 36 and 66. In `comp(x,y,n)`, the MPI communication functions `send` (`MPI_Send`) and `receive` (`MPI_Recv`) have to be prefixed with an `A`.

It becomes clear that only a small number of adaptations are needed. They are limited to the inclusion of the header file and the prepending of 'A' to the initialization, finalization and every communication call. No particular insight into the inner workings of AMPI is required.

Appendix B

Developer Documentation

The developer documentation is entirely generated by Doxygen. This is the generated documentation as of the day this work has been submitted. For a more recent version, please refer to the AMPI repository where this documentation is included. The documentation is generated using the command `make doc`.

Adjoint MPI

Generated by Doxygen 1.8.6

Sat Jul 18 2015 13:35:33

Contents

1	Main Page	1
2	File Index	1
2.1	File List	1
3	File Documentation	2
3.1	ampi.h File Reference	2
3.1.1	Data Structure Documentation	4
3.1.2	Function Documentation	4
3.2	ampi_interface.h File Reference	15
3.2.1	Macro Definition Documentation	15
3.2.2	Function Documentation	15
3.3	ampi_stack.h File Reference	16
3.3.1	Data Structure Documentation	16
3.3.2	Function Documentation	17
3.4	ampi_tape.h File Reference	18
3.4.1	Data Structure Documentation	20
3.4.2	Macro Definition Documentation	20
3.4.3	Function Documentation	21
3.5	ampi_tape.hpp File Reference	23
	Index	24

1 Main Page

The adjoint MPI library served as a prototyping library for developing and applying adjoint MPI patterns in the context of my disseration "Semantics Driven Adjoints of the Message Parsing Interface". It is provided as is under the MIT licence. In the disseration you will find the theoretical backround for the reversing or adjoining MPI communication. The unique feature of this library is its generic interface in [ampi_interface.h](#) that allows an easy integration into other AD tools not yet aware of MPI calls.

2 File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

[ampi.h](#)

Header file for source transformation tools. This file constitutes the interface for source transformation tools. The AMPI tape is dependent on these routines. All MPI routines are subdivided into their forward `_f` and backward `_b` counterpart. The forward routines are called during the forward/taping run. The backward routines are called during the reverse/interpretation run

2

ampi_interface.h

AMPI interface routines which are defined as external. These routines need to be implemented by the external AD tool library. They define the data flow between the external tape and the AMPI tape

15

ampi_stack.h

The AMPI stack is used for tracing the operations in the **AMPI_Reduce()** and **AMPI_Allreduce()**

16

ampi_tape.h

Ampi_tape.h provides the AMPI tape intended to be used by an overloading AD tool. The AMPI tape handles and saves all the relevant MPI information, while the data is kept in the trace of the AD tool. Interface functions are defined that allow AMPI and the AD tool to exchange information. This interface routines need to be implemented for each AD tool. The MPI communications are directly mapped to the active routines defined in this file by calling the forward active routines in **ampi.h**. In the reverse section, the AD tool calls the AMPI tape through **ampi_interpret_tape()** when it hits an AMPI operation. The AMPI tape then executes the reverse MPI communication

18

ampi_tape.hpp

A C++ wrapper for the C header **ampi_tape.h**

23

3 File Documentation

3.1 ampi.h File Reference

Header file for source transformation tools. This file constitutes the interface for source transformation tools. The AMPI tape is dependent on these routines. All MPI routines are subdivided into their forward **_f** and backward **_b** counterpart. The forward routines are called during the forward/taping run. The backward routines are called during the reverse/interpretation run.

```
#include <mpi.h>
#include <ampi_interface.h>
#include <ampi_stack.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
```

Data Structures

- struct **AMPI_Request**
- struct **AMPI_Tupel**

Macros

AMPI_Request Send/Receive

*Defines for distinguishing send and receive in an **AMPI_Request***

- **#define AMPI_IS 1**
- **#define AMPI_IR 2**

Legacy defines with active predefined constants

- **#define AMPI_COMM_WORLD MPI_COMM_WORLD**
- **#define AMPI_MAX_PROCESSOR_NAME MPI_MAX_PROCESSOR_NAME**
- **#define AMPI_Status MPI_Status**

Active Reduce Operations

- `#define AMPI_SUM 1`
- `#define AMPI_PROD 2`
- `#define AMPI_MIN 3`
- `#define AMPI_MAX 4`

Functions

- `int AMPI_Init_f` (int *argc, char ***argv)
- `int AMPI_Init_b` (int *argc, char ***argv)
- `int AMPI_Comm_size` (MPI_Comm comm, int *numprocs)
- `int AMPI_Comm_rank` (MPI_Comm comm, int *rank)
- `int AMPI_Get_processor_name` (char *name, int *namelength)
- `int AMPI_Barrier` (MPI_Comm comm)
- `int AMPI_Finalize_f` ()
- `int AMPI_Finalize_b` ()
- `int AMPI_Send_f` (double *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
- `int AMPI_Send_b` (double *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm)
- `int AMPI_Recv_f` (double *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Status *status)
- `int AMPI_Recv_b` (double *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Status *status)
- `int AMPI_Isend_f` (double *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, `AMPI_Request` *request)
- `int AMPI_Isend_b` (double *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, `AMPI_Request` *request)
- `int AMPI_Irecv_f` (double *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, `AMPI_Request` *request)
- `int AMPI_Irecv_b` (double *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, `AMPI_Request` *request)
- `int AMPI_Wait_f` (`AMPI_Request` *request, MPI_Status *status)
- `int AMPI_Wait_b` (`AMPI_Request` *request, MPI_Status *status)
- `int AMPI_Waitall_f` (int count, `AMPI_Request` *requests, MPI_Status *status)
- `int AMPI_Waitall_b` (int count, `AMPI_Request` *requests, MPI_Status *status)
- `int AMPI_Awaitall_f` (int count, `AMPI_Request` *requests, MPI_Status *status)
- `int AMPI_Awaitall_b` (int count, `AMPI_Request` *requests, MPI_Status *status)
- `int AMPI_Bcast_f` (double *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- `int AMPI_Bcast_b` (double *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
- `int AMPI_Reduce_f` (double *sendbuf, double *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
- `int AMPI_Reduce_b` (double *sendbuf, double *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
- `int AMPI_Allreduce_f` (double *sendbuf, double *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
- `int AMPI_Allreduce_b` (double *sendbuf, double *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
- `int AMPI_Gather_f` (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
- `int AMPI_Gather_b` (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
- `int AMPI_Scatter_f` (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
- `int AMPI_Scatter_b` (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)

- int [AMPI_Sendrecv_replace_f](#) (double *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
- int [AMPI_Sendrecv_replace_b](#) (double *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

3.1.1 Data Structure Documentation

3.1.1.1 struct AMPI_Request

AMPI request, replacing the MPI request

Data Fields

double *	a	Incoming or outgoing adjoints of the mapped tape.
long int	aw	Anti wait flag
void *	buf	Only used in the tape. Here we store the pointer to the active buffer that is conveyed to the Wait.
MPI_Comm	comm	Original communicator
int	dest	Destination or source.
int	oc	Operation code
MPI_Request *	request	Original request
int	size	Size of the buffer.
MPI_Status	status	Original status
int	tag	MPI tag
double *	v	Incoming or outgoing values of the mapped active buffer.
long int	va	Tape index

3.1.1.2 struct AMPI_Tupel

Tupel used in the MPI_MAX and MPI_MIN reduction to save the rank of the process with the maximum or minimum

Data Fields

int	j	rank
double	v	value

3.1.2 Function Documentation

3.1.2.1 int AMPI_Allreduce_b (double * *sendbuf*, double * *recvbuf*, int *count*, MPI_Datatype *datatype*, MPI_Op *op*, MPI_Comm *comm*)

Reverse active allreduce. All the adjoints of all the processes have to be summed up and sent to all processes, essentially amounting to an allreduce with an MPI_SUM operation. We then apply the adjoint computation like in the common reduction.

Parameters

<i>sendbuf</i>	The incoming adjoint buffer address
<i>recvbuf</i>	The broadcast adjoint buffer address
<i>count</i>	Number of buffer elements
<i>datatype</i>	MPI data type of the buffer elements
<i>op</i>	Reduction operation (MPI_SUM, MPI_PROD, MPI_MAX or MPI_MIN)
<i>comm</i>	MPI communicator

Returns

error code

3.1.2.2 `int AMPI_Allreduce_f ( double * sendbuf, double * recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )`

Forward active allreduce. See corresponding papers for additional information. The information necessary for the adjoint operation is saved.

MPI_SUM and MPI_PROD: The result and the input send buffer are saved. MPI_MAX and MPI_MIN: The rank of the process which holds the maximum oder minimum is saved. Hence, we execute an MPI_MAXLOC or MPI_MINLOC after copying the data accordingly.

Parameters

<i>sendbuf</i>	The sent value buffer
<i>recvbuf</i>	The received value buffer. In case of the MPI_PROD all the processes have the result.
<i>count</i>	Number of buffer elements
<i>datatype</i>	MPI data type of the buffer elements
<i>op</i>	Reduction operation (MPI_SUM, MPI_PROD, MPI_MAX or MPI_MIN)
<i>comm</i>	MPI communicator

Returns

error code

3.1.2.3 `int AMPI_Awaitall_b ( int count, AMPI_Request * requests, MPI_Status * status )`

Experimental implementation of the anti-wait. See corresponding paper for additional information. In the reverse routine, all the wait operations for the adjoint communications are executed here, as opposed to single waits in the non blocking sends and receives.

Parameters

<i>count</i>	Number of requests
<i>requests</i>	Active requests
<i>status</i>	Dummy argument. Status is inside active requests

Returns

error code

3.1.2.4 `int AMPI_Awaitall_f ( int count, AMPI_Request * requests, MPI_Status * status )`

Experimental implementation of the anti-wait. See corresponding paper for additional information. In the forward routine, the anti-wait flag in the active requests are set.

Parameters

<i>count</i>	Number of requests
<i>requests</i>	Active requests
<i>status</i>	Original statuses. Dummy argument

Returns

error code

3.1.2.5 `int AMPI_Barrier ( MPI_Comm comm )`

Active barrier. Amounts to a wrapper of MPI_Barrier. Does not need to be traced.

Parameters

<i>comm</i>	Communicator with the processes that execute a barrier
-------------	--

Returns

error code

3.1.2.6 `int AMPI_Bcast_b ( double * buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm )`

Reverse active broadcast. An MPI_Reduce with the MPI_SUM operation is executed on the adjoints.

Parameters

<i>buf</i>	Send buffer with adjoints that are to be reduced. After AMPI_Bcast_f is called this is the received buffer of reduced adjoint on the root process.
<i>count</i>	Number of buffer elements
<i>datatype</i>	MPI data type of the buffer elements
<i>root</i>	Process with the buffer that the adjoints are reduced to
<i>comm</i>	MPI communicator

Returns

error code

3.1.2.7 `int AMPI_Bcast_f ( double * buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm )`

Forward active broadcast. An MPI_Broadcast is executed.

Parameters

<i>buf</i>	Buffer of broadcast value
<i>count</i>	Number of buffer elements
<i>datatype</i>	MPI data type of the buffer elements
<i>root</i>	Process with the buffer that is broadcast
<i>comm</i>	MPI communicator

Returns

error code

3.1.2.8 `int AMPI_Comm_rank ( MPI_Comm comm, int * rank )`

Active variant of MPI_Comm_rank. The rank is saved in a global variable to avoid repeated calls to MPI_Comm_rank.

Parameters

<i>comm</i>	Communicator
<i>rank</i>	Rank of calling process

Returns

error code

3.1.2.9 `int AMPI_Comm_size ( MPI_Comm comm, int * numprocs )`

Legacy active variant of MPI_Comm_size. Only a wrapper of MPI_Comm_size.

Parameters

<i>comm</i>	Communicator
<i>numprocs</i>	Number of processes

Returns

error code

3.1.2.10 int AMPI_Finalize_b ()

Active reverse MPI_Finalize(). Nothing is done here.

Returns

error code

3.1.2.11 int AMPI_Finalize_f ()

Active forward MPI_Finalize(). Nothing is done here.

Returns

error code

3.1.2.12 int AMPI_Gather_b (void * *sendbuf*, int *sendcnt*, MPI_Datatype *sendtype*, void * *recvbuf*, int *recvcnt*, MPI_Datatype *recvtype*, int *root*, MPI_Comm *comm*)

Active reverse gather. MPI_Scatter wrapper. *sendbuf* are the received adjoints while *recvbuf* are the sent adjoints.

3.1.2.13 int AMPI_Gather_f (void * *sendbuf*, int *sendcnt*, MPI_Datatype *sendtype*, void * *recvbuf*, int *recvcnt*, MPI_Datatype *recvtype*, int *root*, MPI_Comm *comm*)

Active forward gather. MPI_Gather wrapper.

3.1.2.14 int AMPI_Get_processor_name (char * *name*, int * *namelength*)

Legacy. Wrapper to MPI_Get_processor_name

Parameters

<i>name</i>	Processor name
<i>namelength</i>	Processor name length.

Returns

error code

3.1.2.15 int AMPI_Init_b (int * *argc*, char *** *argv*)

Active reverse MPI Init. AMPI data structures are destroyed and MPI_Finalize() is called.

Parameters

<i>argv</i>	Dummy argc. Not used
-------------	----------------------

<i>argc</i>	Dummy argv. Not used
-------------	----------------------

Returns

error code

3.1.2.16 int AMPI_Init_f (int * *argc*, char * *argv*)**

Active forward MPI_Init. First chunk of the AMPI tape is allocated.

Parameters

<i>argc</i>	argc of passive code
<i>argv</i>	argv of passive code

Returns

error code

3.1.2.17 int AMPI_lrecv_b (double * *buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*, AMPI_Request * *request*)

Active reverse non blocking receive. This amounts to an MPI_Wait. For sementical reasons, the sent adjoints are copied to the adjoint buffer buf that is conveyed together with the MPI_Request through the [AMPI_Request](#).

Parameters

<i>buf</i>	Points to the buffer address of the sent adjoints in the request
<i>count</i>	Dummy argument
<i>datatype</i>	Dummy argument
<i>dest</i>	Dummy argument
<i>tag</i>	Message tag
<i>comm</i>	MPI Dummy argument
<i>request</i>	Active MPI_Request. It conveys the buffer address for the adjoints from the reverse wait to the reverse send (=receive).

Returns

error code

3.1.2.18 int AMPI_lrecv_f (double * *buf*, int *count*, MPI_Datatype *datatype*, int *dest*, int *tag*, MPI_Comm *comm*, AMPI_Request * *request*)

Active forward non blocking receive

Parameters

<i>buf</i>	Buffer with values that are to be received
<i>count</i>	Number of buffer elements
<i>datatype</i>	MPI data type of the buffer elements
<i>dest</i>	Rank of source process
<i>tag</i>	Message tag
<i>comm</i>	MPI communicator
<i>request</i>	Active MPI_Request. Additional information (buf, count, dest, oc, comm and the original MPI_Request) is saved in here for later use in AMPI_Wait_f or AMPI_Waitall_f.

Returns

error code

3.1.2.19 `int AMPI_Isend_b (double * buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm,
AMPI_Request * request)`

Active reverse non blocking send. This amounts to an MPI_Wait. The received adjoints are copied to the adjoint buffer that is conveyed together with the MPI_Request through the [AMPI_Request](#).

Parameters

<i>buf</i>	Points to the buffer address for the received adjoints in the request
<i>count</i>	Dummy argument
<i>datatype</i>	Dummy argument
<i>src</i>	Dummy argument
<i>tag</i>	Message tag
<i>comm</i>	MPI Dummy argument
<i>request</i>	Active MPI_Request. It conveys the buffer address for the adjoints from the reverse wait to the reverse send (=receive).

Returns

error code

3.1.2.20 `int AMPI_Isend_f ( double * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request * request )`

Active forward non blocking send

Parameters

<i>buf</i>	Buffer with values that are sent
<i>count</i>	Number of buffer elements
<i>datatype</i>	MPI data type of the buffer elements
<i>dest</i>	Rank of destination process
<i>tag</i>	Message tag
<i>comm</i>	MPI communicator
<i>request</i>	Active MPI_Request. Additional information (<i>buf</i> , <i>count</i> , <i>dest</i> , <i>oc</i> , <i>comm</i> and the original MPI_Request) is saved in here for later use in AMPI_Wait_f or AMPI_Waitall_f.

Returns

error code

3.1.2.21 `int AMPI_Recv_b ( double * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Status * status )`

Active reverse receive.

The active reverse receive is a wrapper of MPI_Send. The status is ignored.

Parameters

<i>buf</i>	Buffer with adjoints that are sent
<i>count</i>	Number of buffer elements
<i>datatype</i>	MPI data type of the buffer elements
<i>dest</i>	Rank of destination process
<i>tag</i>	Message tag
<i>comm</i>	MPI communicator
<i>status</i>	Ignored. Only present for consistency with the MPI signatures

Returns

error code

3.1.2.22 `int AMPI_Recv_f ( double * buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Status * status )`

Active forward receive.

The active forward receive is a wrapper of `MPI_Recv`.

Parameters

<i>buf</i>	Buffer with values that are received
<i>count</i>	Number of buffer elements
<i>datatype</i>	MPI data type of the buffer elements
<i>src</i>	Rank of source process
<i>tag</i>	Message tag
<i>comm</i>	MPI communicator
<i>status</i>	MPI status of the received value message

Returns

error code

3.1.2.23 `int AMPI_Reduce_b ( double * sendbuf, double * recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )`

Reverse active reduction. The adjoint of a reduction is a broadcast with an additional operation on the incoming distributed adjoint. There is no MPI routine that implements this. So we have to decompose the communication into the broadcast and a local operation on each process. In case of MPI_PROD, we have currently implemented the short solution, where the local adjoint is computed by dividing the global result with the local result.

Parameters

<i>sendbuf</i>	The incoming adjoint buffer address
<i>recvbuf</i>	The broadcast adjoint buffer address
<i>count</i>	Number of buffer elements
<i>datatype</i>	MPI data type of the buffer elements
<i>op</i>	Reduction operation (MPI_SUM, MPI_PROD, MPI_MAX or MPI_MIN)
<i>root</i>	Process with the buffer that the adjoints are broadcast from
<i>comm</i>	MPI communicator

Returns

error code

3.1.2.24 `int AMPI_Reduce_f ( double * sendbuf, double * recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )`

Forward active reduce. See corresponding papers for additional information. Depending on the operation *op*, an MPI communication is executed and the information necessary for the adjoint operation is saved.

MPI_SUM: An MPI_Reduce is executed. MPI_PROD: An MPI_Allreduce is executed, because all the processes need to result of the multiplication. MPI_MAX and MPI_MIN: The rank of the process which holds the maximum or minimum is saved. Hence, we execute an MPI_MAXLOC or MPI_MINLOC after copying the data accordingly.

Parameters

<i>sendbuf</i>	The sent value buffer
<i>recvbuf</i>	The received value buffer. In case of the MPI_PROD all the processes have the result.
<i>count</i>	Number of buffer elements
<i>datatype</i>	MPI data type of the buffer elements
<i>op</i>	Reduction operation (MPI_SUM, MPI_PROD, MPI_MAX or MPI_MIN)
<i>root</i>	Process with the buffer that the values are reduced to irrelevant in case of MPI_PROD.

<i>comm</i>	MPI communicator
-------------	------------------

Returns

error code

3.1.2.25 `int AMPI_Scatter_b ( void * sendbuf, int sendcnt, MPI_Datatype sendtype, void * recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm )`

Active reverse scatter. MPI_Gather wrapper. *sendbuf* are the received adjoints while *recvbuf* are the sent adjoints.

3.1.2.26 `int AMPI_Scatter_f ( void * sendbuf, int sendcnt, MPI_Datatype sendtype, void * recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm )`

Active forward scatter. MPI_Scatter wrapper.

3.1.2.27 `int AMPI_Send_b ( double * buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm )`

Active reverse send.

The active reverse send amounts to an MPI_Receive with MPI_STATUS_IGNORE.

Parameters

<i>buf</i>	Buffer with adjoints that are received
<i>count</i>	Number of buffer elements
<i>datatype</i>	MPI data type of the buffer elements
<i>src</i>	Rank of source process
<i>tag</i>	Message tag
<i>comm</i>	MPI communicator

Returns

error code

3.1.2.28 `int AMPI_Send_f ( double * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )`

Active forward send.

The forward send amounts to a wrapper of the MPI_Send.

Parameters

<i>buf</i>	Buffer with values that are to be sent
<i>count</i>	Number of buffer elements
<i>datatype</i>	MPI data type of the buffer elements
<i>dest</i>	Rank of destination process
<i>tag</i>	Message tag
<i>comm</i>	MPI communicator

Returns

error code

3.1.2.29 `int AMPI_Sendrecv_replace_b ( double * buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status * status )`

Active reverse combined send and receive where the same buffer is used. MPI_Sendrecv_replace wrapper. The reversal amounts to switching destination and source

3.1.2.30 `int AMPI_Sendrecv_replace_f ( double * buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status * status )`

Active forward combined send and receive where the same buffer is used. `MPI_Sendrecv_replace` wrapper.

3.1.2.31 `int AMPI_Wait_b ( AMPI_Request * request, MPI_Status * status )`

Active reverse wait. Depending on the associated communication a non blocking send (for a receive) or a receive (for a send) is executed with the adjoint buffer address. The information for the reverse communication is assumed to be stored in the active request.

Parameters

<i>request</i>	Active MPI request with all the information needed for the reverse communication
<i>status</i>	Dummy argument. Status is saved in the active requests

Returns

error code

3.1.2.32 `int AMPI_Wait_f ( AMPI_Request * request, MPI_Status * status )`

Active forward wait. An `MPI_Wait` is executed. In the request, additional information is provided (destination, source, value buffer address, communicator, count, type of communication).

Parameters

<i>request</i>	Active MPI request
<i>status</i>	The original status

Returns

error code

3.1.2.33 `int AMPI_Waitall_b ( int count, AMPI_Request * requests, MPI_Status * status )`

Active reverse waitall. This marks a performance loss in the active case. For each forward wait, a send or receive communication has to be started. Therefore we call an active reverse wait for each active request.

Parameters

<i>count</i>	Number of requests
<i>requests</i>	Active requests
<i>status</i>	Dummy argument. Status is saved in the active requests

Returns

error code

3.1.2.34 `int AMPI_Waitall_f ( int count, AMPI_Request * requests, MPI_Status * status )`

Active forward waitall. An `MPI_Waitall` is executed. The original `MPI_Request` requests are saved into the active requests.

Parameters

<i>count</i>	Number of requests
<i>requests</i>	Active requests
<i>status</i>	Original statuses

Returns

error code

3.2 `ampi_interface.h` File Reference

AMPI interface routines which are defined as external. These routines need to be implemented by the external AD tool library. They define the data flow between the external tape and the AMPI tape.

Macros

- `#define INT64` long int

Functions

- void `ampi_interpret_tape` (long int idx)
- void `ampi_reset_entry` (long int idx)
- void `ampi_get_val` (void *buf, int *i, double *v)
- void `ampi_set_val` (void *buf, int *i, double *v)
- void `ampi_get_adj` (INT64 *idx, double *a)
- void `ampi_set_adj` (INT64 *, double *)
- void `ampi_get_idx` (void *buf, int *i, INT64 *idx)
- void `ampi_create_tape_entry` (long int *i)
- void `ampi_create_dummies` (void *buf, int *size)
- int `ampi_is_tape_active` ()

3.2.1 Macro Definition Documentation

3.2.1.1 `#define INT64` long int

Defines the type of the tape index as defined by the AD tool. Due to circular dependence, this has to be defined here in addition to the `ampi` interface.

3.2.2 Function Documentation

3.2.2.1 void `ampi_create_dummies` (void * buf, int * size)

Create size tape entries to store the values of buf. Refer to receive buffer without initialization

3.2.2.2 void `ampi_create_tape_entry` (long int * i)

Create a tape entry in the external tape, indicating an external AMPI call

3.2.2.3 void `ampi_get_adj` (INT64 * idx, double * a)

Get an adjoint a from a specific tape entry with index idx

3.2.2.4 void `ampi_get_idx` (void * buf, int * i, INT64 * idx)

Get a tape index from a variable buf[i]

3.2.2.5 void `ampi_get_val` (void * *buf*, int * *i*, double * *v*)

Get a value *v from a specific tape variable `buf[i]`

3.2.2.6 void `ampi_interpret_tape` (long int *idx*)

Call AMPI tape interpreter from external tape

3.2.2.7 void `ampi_set_adj` (INT64 * , double *)

Set an adjoint a from a specific tape entry with index `idx`

3.2.2.8 void `ampi_set_val` (void * *buf*, int * *i*, double * *v*)

Set a value *v from a specific tape variable `buf[i]`

3.3 `ampi_stack.h` File Reference

The AMPI stack is used for tracing the operations in the `AMPI_Reduce()` and `AMPI_Allreduce()`.

```
#include <stdlib.h>
```

Data Structures

- struct `ampi_stack`

Macros

- #define `CHUNK_SIZE` 1000

Functions

- void `AMPI_push` (`ampi_stack` *s, double val)
Push value.
- double `AMPI_pop` (`ampi_stack` *s)
Pop value.
- void `AMPI_stack_init` (`ampi_stack` *s)
Create stack.
- void `AMPI_destroy` (`ampi_stack` *s)
Destroy stack.
- int `AMPI_full` (`ampi_stack` *s)
Check whether stack is full. Only used by internally.
- void `AMPI_expand` (`ampi_stack` *s)
Expand stack. Only used internally.
- void `AMPI_shrink` (`ampi_stack` *s)
Shrink stack. Only used internally.
- int `AMPI_empty` (`ampi_stack` *s)
Check whether stack is empty. Only used internally.

3.3.1 Data Structure Documentation

3.3.1.1 struct `ampi_stack`

AMPI stack used to save values in particular for the reductions

Data Fields

long int	size	size of the stack
long int	top	top of the stack
double *	v	value

3.3.2 Function Documentation

3.3.2.1 `void AMPI_destroy ( ampi_stack * s )`

Parameters

<i>s</i>	stack to be destroyed
----------	-----------------------

3.3.2.2 `int AMPI_empty ( ampi_stack * s )`

Parameters

<i>s</i>	stack to be checked
----------	---------------------

Returns

non zero if stack is empty

3.3.2.3 `void AMPI_expand ( ampi_stack * s )`

Parameters

<i>s</i>	stack to expanded
----------	-------------------

3.3.2.4 `int AMPI_full ( ampi_stack * s )`

Parameters

<i>s</i>	stack to be checked
----------	---------------------

Returns

non zero if stack is full

3.3.2.5 `double AMPI_pop ( ampi_stack * s )`

Parameters

<i>s</i>	stack
----------	-------

Returns

value

3.3.2.6 `void AMPI_push ( ampi_stack * s, double val )`

Parameters

<i>s</i>	stack
<i>val</i>	value

3.3.2.7 void AMPI_shrink (*ampi_stack* * *s*)

Parameters

<i>s</i>	stack to be shrunk
----------	--------------------

3.3.2.8 void AMPI_stack_init (*ampi_stack* * *s*)

Parameters

<i>s</i>	created stack
----------	---------------

3.4 *ampi_tape.h* File Reference

[ampi_tape.h](#) provides the AMPI tape intended to be used by an overloading AD tool. The AMPI tape handles and saves all the relevant MPI information, while the data is kept in the trace of the AD tool. Interface functions are defined that allow AMPI and the AD tool to exchange information. This interface routines need to be implemented for each AD tool. The MPI communications are directly mapped to the active routines defined in this file by calling the forward active routines in [ampi.h](#). In the reverse section, the AD tool calls the AMPI tape through [ampi_interpret_tape\(\)](#) when it hits an AMPI operation. The AMPI tape then executes the reverse MPI communication.

```
#include <stdlib.h>
#include <assert.h>
#include <ampi.h>
#include <uthash.h>
```

Data Structures

- struct [AMPI_ht_el](#)
- struct [ampi_tape_entry](#)

Macros

- #define [AMPI_DOUBLE](#) MPI_DOUBLE
- #define [AMPI_CHUNK_SIZE](#) 500000

Internal defines for the reduction operation

- #define **AMPI_REDUCE_ADD** 1
- #define **AMPI_REDUCE_MUL** 2
- #define **AMPI_REDUCE_MIN** 3
- #define **AMPI_REDUCE_MAX** 4

Internal defines for the MPI communications

- #define **SEND** 1
- #define **RECV** 2
- #define **ISEND** 3
- #define **Irecv** 4
- #define **WAIT** 5
- #define **WAITALL** 6
- #define **AWAITALL** 7

- `#define BCAST 8`
- `#define REDUCE 9`
- `#define ALLREDUCE 10`
- `#define MPI_DUMMY 11`
- `#define MPI_ADUMMY 12`
- `#define SENDRECVREPLACE 13`
- `#define SCATTER 14`
- `#define GATHER 15`
- `#define SCATTERV 16`
- `#define GATHERV 17`
- `#define SEND_INIT 18`
- `#define RECV_INIT 19`
- `#define START 20`
- `#define STARTALL 21`

Functions

- `int AMPI_Reset_Tape ()`
- `int AMPI_Store_State ()`
- `int AMPI_Restore_State ()`
- `int AMPI_Init (int *argc, char ***argv)`
- `int AMPI_Finalize ()`
- `int AMPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
Active blocking send with active buffer.
- `int AMPI_Recv (void *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Status *status)`
Active blocking receive with active buffer.
- `int AMPI_Isend (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`
Active non blocking send with active buffer.
- `int AMPI_Irecv (void *buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Request *request)`
Active non blocking receive with active buffer.
- `int AMPI_Wait (MPI_Request *, MPI_Status *)`
Active wait. If the MPI_Request is in the hash table, the communication is active.
- `int AMPI_Waitall (int, MPI_Request *, MPI_Status *)`
Active waitall. If the MPI_Request is in the hash table, the communication is active, otherwise only MPI_Wait is called.
- `int AMPI_Waitany (int count, MPI_Request array_of_requests[], int *index, MPI_Status *status)`
Active waitall. If the MPI_Request is in the hash table, the communication is active, otherwise only MPI_Wait is called. If no requests are left, MPI_Waitany is called one last time to set the MPI library's handlers correctly.
- `int AMPI_Awaitall (int, AMPI_Request *, MPI_Status *)`
- `int AMPI_Bcast (void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
Active broadcast.
- `int AMPI_Reduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
Active reduce.
- `int AMPI_Allreduce (void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)`
Active allreduce.
- `int AMPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)`
Active scatter.
- `int AMPI_Gather (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)`
Active gather.

- int [AMPI_Send_init](#) (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
Active receive init. The active AMPI_Requests are registered. MPI_Recv_init is not executed in AMPI. See AMPI_Start.
- int [AMPI_Recv_init](#) (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
Active receive init. The active AMPI_Requests are registered. MPI_Send_init is not executed in AMPI. See AMPI_Start.
- int [AMPI_Start](#) (MPI_Request *request)
Active start. Call AMPI_Isend or AMPI_Irecv for active request.
- int [AMPI_Startall](#) (int count, MPI_Request array_of_requests[])
Active startall. Call AMPI_Start count times.
- int [AMPI_Sendrecv_replace](#) (void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)
Active sendrecv with replace.
- void [ampi_print_tape](#) (void)
- void [ampi_print_tape_entry](#) (int *j)
- void [ampi_check_tape_size](#) (long int size)

3.4.1 Data Structure Documentation

3.4.1.1 struct AMPI_ht_el

The UT_hash hash table is used to link the MPI_Request to an active [AMPI_Request](#) when dealing with an overloading AD tool. This struct defines the hash table data structure.

See further documentation at <http://troydhanson.github.io/uthash/index.html>

Data Fields

UT_hash_ - handle	hh	Used internally by UT_hash
void *	key	Pointer to the MPI_Request
AMPI_Request	request	The active AMPI_Request

3.4.1.2 struct ampi_tape_entry

An element of the AMPI tape.

Data Fields

int *	arg	Array of arguments
MPI_Comm	comm	MPI_Communicator
INT64	idx	AD tool tape index
int	oc	Operation code
AMPI_Request *	request	Saved pointer to an active AMPI_Request
int	tag	MPI_Tag

3.4.2 Macro Definition Documentation

3.4.2.1 #define AMPI_CHUNK_SIZE 500000

Sets the chunk size of the AMPI tape. When the tape exceeds this size, a new chunk is allocated.

3.4.2.2 #define AMPI_DOUBLE MPI_DOUBLE

Sets the active MPI type. If all buffers of type MPI_DOUBLE should be communicated actively, set this to MPI_DOUBLE

3.4.3 Function Documentation

3.4.3.1 `int AMPI_Allreduce ( void * sendbuf, void * recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm )`

Parameters

<i>sendbuf</i>	Pointer to active send buffer
<i>recvbuf</i>	Pointer to active receive buffer

3.4.3.2 `int AMPI_Awaitall ( int , AMPI_Request * , MPI_Status * )`

Experimental implementation of an anti-waitall. See the corresponding paper for more information.

3.4.3.3 `int AMPI_Bcast ( void * buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm )`

Parameters

<i>buf</i>	Pointer to active buffer
------------	--------------------------

3.4.3.4 `void ampi_check_tape_size ( long int size )`

Check whether the tape size is larger than *size*. If not, a reallocation is triggered with an additional chunk.

3.4.3.5 `int AMPI_Finalize ( )`

AMPI Finalize is called *after* the reverse section. Hence `AMPI_Init_b` is called here.

Returns

error code

3.4.3.6 `int AMPI_Gather ( void * sendbuf, int sendcnt, MPI_Datatype sendtype, void * recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm )`

Parameters

<i>sendbuf</i>	Pointer to active send buffer
<i>recvbuf</i>	Pointer to active receive buffer

3.4.3.7 `int AMPI_Init ( int * argc, char *** argv )`

Initialize AMPI consisting of allocating the AMPI tape and calling `AMPI_Init_f`

Parameters

<i>argc</i>	Forwarded to MPI.
<i>argv</i>	Forwarded to MPI

Returns

error code

3.4.3.8 `int AMPI_Irecv ( void * buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Request * request )`

Parameters

<i>buf</i>	Pointer to active buffer
------------	--------------------------

3.4.3.9 `int AMPI_Isend ( void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request * request )`

Parameters

<i>buf</i>	Pointer to active buffer
------------	--------------------------

3.4.3.10 `void ampi_print_tape ( void )`

Call AMPI tape printer from external tape printer

3.4.3.11 `void ampi_print_tape_entry ( int * j )`

Call AMPI tape printer from external tape printer for one tape entry

3.4.3.12 `int AMPI_Recv ( void * buf, int count, MPI_Datatype datatype, int src, int tag, MPI_Comm comm, MPI_Status * status )`

Parameters

<i>buf</i>	Pointer to active buffer
------------	--------------------------

3.4.3.13 `int AMPI_Recv_init ( void * buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request * request )`

Parameters

<i>buf</i>	Pointer to active send buffer
------------	-------------------------------

3.4.3.14 `int AMPI_Reduce ( void * sendbuf, void * recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm )`

Parameters

<i>sendbuf</i>	Pointer to active send buffer
<i>recvbuf</i>	Pointer to active receive buffer

3.4.3.15 `int AMPI_Reset_Tape ( )`

AMPI taping routines which have an MPI counterpart that is adjoined. Reset the AMPI tape. All the tape entries are destroyed.

Returns

error code

3.4.3.16 `int AMPI_Scatter ( void * sendbuf, int sendcnt, MPI_Datatype sendtype, void * recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm )`

Parameters

<i>sendbuf</i>	Pointer to active send buffer
<i>recvbuf</i>	Pointer to active receive buffer

3.4.3.17 `int AMPI_Send ( void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm )`

Parameters

<i>buf</i>	Pointer to active buffer
------------	--------------------------

3.4.3.18 `int AMPI_Send_init ( void * buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request * request )`

Parameters

<i>buf</i>	Pointer to active receive buffer
------------	----------------------------------

3.4.3.19 `int AMPI_Sendrecv_replace ( void * buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status * status )`

Parameters

<i>buf</i>	Pointer to active buf
------------	-----------------------

3.4.3.20 `int AMPI_Start ( MPI_Request * request )`

3.4.3.21 `int AMPI_Startall ( int count, MPI_Request array_of_requests[] )`

3.5 `ampi_tape.hpp` File Reference

A C++ wrapper for the C header [ampi_tape.h](#).

```
#include <mpi.h>
#include "ampi_tape.h"
```

Index

AMPI_Request, [4](#)
AMPI_Tupel, [4](#)
AMPI_ht_el, [20](#)
AMPI_Allreduce
 [ampi_tape.h, 21](#)
AMPI_Allreduce_b
 [ampi.h, 4](#)
AMPI_Allreduce_f
 [ampi.h, 4](#)
AMPI_Awaitall
 [ampi_tape.h, 21](#)
AMPI_Awaitall_b
 [ampi.h, 5](#)
AMPI_Awaitall_f
 [ampi.h, 5](#)
AMPI_Barrier
 [ampi.h, 5](#)
AMPI_Bcast
 [ampi_tape.h, 21](#)
AMPI_Bcast_b
 [ampi.h, 6](#)
AMPI_Bcast_f
 [ampi.h, 6](#)
AMPI_CHUNK_SIZE
 [ampi_tape.h, 20](#)
AMPI_Comm_rank
 [ampi.h, 6](#)
AMPI_Comm_size
 [ampi.h, 6](#)
AMPI_DOUBLE
 [ampi_tape.h, 20](#)
AMPI_Finalize
 [ampi_tape.h, 21](#)
AMPI_Finalize_b
 [ampi.h, 7](#)
AMPI_Finalize_f
 [ampi.h, 7](#)
AMPI_Gather
 [ampi_tape.h, 21](#)
AMPI_Gather_b
 [ampi.h, 7](#)
AMPI_Gather_f
 [ampi.h, 7](#)
AMPI_Get_processor_name
 [ampi.h, 7](#)
AMPI_Init
 [ampi_tape.h, 21](#)
AMPI_Init_b
 [ampi.h, 7](#)
AMPI_Init_f
 [ampi.h, 8](#)
AMPI_Irecv
 [ampi_tape.h, 21](#)
AMPI_Irecv_b
 [ampi.h, 8](#)
AMPI_Irecv_f
 [ampi.h, 8](#)
AMPI_Isend
 [ampi_tape.h, 22](#)
AMPI_Isend_b
 [ampi.h, 8](#)
AMPI_Isend_f
 [ampi.h, 10](#)
AMPI_Recv
 [ampi_tape.h, 22](#)
AMPI_Recv_b
 [ampi.h, 10](#)
AMPI_Recv_f
 [ampi.h, 10](#)
AMPI_Recv_init
 [ampi_tape.h, 22](#)
AMPI_Reduce
 [ampi_tape.h, 22](#)
AMPI_Reduce_b
 [ampi.h, 12](#)
AMPI_Reduce_f
 [ampi.h, 12](#)
AMPI_Reset_Tape
 [ampi_tape.h, 22](#)
AMPI_Scatter
 [ampi_tape.h, 22](#)
AMPI_Scatter_b
 [ampi.h, 13](#)
AMPI_Scatter_f
 [ampi.h, 13](#)
AMPI_Send
 [ampi_tape.h, 23](#)
AMPI_Send_b
 [ampi.h, 13](#)
AMPI_Send_f
 [ampi.h, 13](#)
AMPI_Send_init
 [ampi_tape.h, 23](#)
AMPI_Sendrecv_replace
 [ampi_tape.h, 23](#)
AMPI_Sendrecv_replace_b
 [ampi.h, 13](#)
AMPI_Sendrecv_replace_f
 [ampi.h, 13](#)
AMPI_Start
 [ampi_tape.h, 23](#)
AMPI_Startall
 [ampi_tape.h, 23](#)
AMPI_Wait_b
 [ampi.h, 14](#)
AMPI_Wait_f
 [ampi.h, 14](#)
AMPI_Waitall_b
 [ampi.h, 14](#)
AMPI_Waitall_f

- ampi.h, [14](#)
- AMPI_destroy
 - ampi_stack.h, [17](#)
- AMPI_empty
 - ampi_stack.h, [17](#)
- AMPI_expand
 - ampi_stack.h, [17](#)
- AMPI_full
 - ampi_stack.h, [17](#)
- AMPI_pop
 - ampi_stack.h, [17](#)
- AMPI_push
 - ampi_stack.h, [17](#)
- AMPI_shrink
 - ampi_stack.h, [18](#)
- AMPI_stack_init
 - ampi_stack.h, [18](#)
- ampi.h, [2](#)
 - AMPI_Allreduce_b, [4](#)
 - AMPI_Allreduce_f, [4](#)
 - AMPI_Awaitall_b, [5](#)
 - AMPI_Awaitall_f, [5](#)
 - AMPI_Barrier, [5](#)
 - AMPI_Bcast_b, [6](#)
 - AMPI_Bcast_f, [6](#)
 - AMPI_Comm_rank, [6](#)
 - AMPI_Comm_size, [6](#)
 - AMPI_Finalize_b, [7](#)
 - AMPI_Finalize_f, [7](#)
 - AMPI_Gather_b, [7](#)
 - AMPI_Gather_f, [7](#)
 - AMPI_Get_processor_name, [7](#)
 - AMPI_Init_b, [7](#)
 - AMPI_Init_f, [8](#)
 - AMPI_Irecv_b, [8](#)
 - AMPI_Irecv_f, [8](#)
 - AMPI_Isend_b, [8](#)
 - AMPI_Isend_f, [10](#)
 - AMPI_Recv_b, [10](#)
 - AMPI_Recv_f, [10](#)
 - AMPI_Reduce_b, [12](#)
 - AMPI_Reduce_f, [12](#)
 - AMPI_Scatter_b, [13](#)
 - AMPI_Scatter_f, [13](#)
 - AMPI_Send_b, [13](#)
 - AMPI_Send_f, [13](#)
 - AMPI_Sendrecv_replace_b, [13](#)
 - AMPI_Sendrecv_replace_f, [13](#)
 - AMPI_Wait_b, [14](#)
 - AMPI_Wait_f, [14](#)
 - AMPI_Waitall_b, [14](#)
 - AMPI_Waitall_f, [14](#)
- ampi_stack, [16](#)
- ampi_tape_entry, [20](#)
- ampi_check_tape_size
 - ampi_tape.h, [21](#)
- ampi_create_dummies
 - ampi_interface.h, [15](#)
- ampi_create_tape_entry
 - ampi_interface.h, [15](#)
- ampi_get_adj
 - ampi_interface.h, [15](#)
- ampi_get_idx
 - ampi_interface.h, [15](#)
- ampi_get_val
 - ampi_interface.h, [15](#)
- ampi_interface.h, [15](#)
 - ampi_create_dummies, [15](#)
 - ampi_create_tape_entry, [15](#)
 - ampi_get_adj, [15](#)
 - ampi_get_idx, [15](#)
 - ampi_get_val, [15](#)
 - ampi_interpret_tape, [16](#)
 - ampi_set_adj, [16](#)
 - ampi_set_val, [16](#)
 - INT64, [15](#)
- ampi_interpret_tape
 - ampi_interface.h, [16](#)
- ampi_print_tape
 - ampi_tape.h, [22](#)
- ampi_print_tape_entry
 - ampi_tape.h, [22](#)
- ampi_set_adj
 - ampi_interface.h, [16](#)
- ampi_set_val
 - ampi_interface.h, [16](#)
- ampi_stack.h, [16](#)
 - AMPI_destroy, [17](#)
 - AMPI_empty, [17](#)
 - AMPI_expand, [17](#)
 - AMPI_full, [17](#)
 - AMPI_pop, [17](#)
 - AMPI_push, [17](#)
 - AMPI_shrink, [18](#)
 - AMPI_stack_init, [18](#)
- ampi_tape.h, [18](#)
 - AMPI_Allreduce, [21](#)
 - AMPI_Awaitall, [21](#)
 - AMPI_Bcast, [21](#)
 - AMPI_CHUNK_SIZE, [20](#)
 - AMPI_DOUBLE, [20](#)
 - AMPI_Finalize, [21](#)
 - AMPI_Gather, [21](#)
 - AMPI_Init, [21](#)
 - AMPI_Irecv, [21](#)
 - AMPI_Isend, [22](#)
 - AMPI_Recv, [22](#)
 - AMPI_Recv_init, [22](#)
 - AMPI_Reduce, [22](#)
 - AMPI_Reset_Tape, [22](#)
 - AMPI_Scatter, [22](#)
 - AMPI_Send, [23](#)
 - AMPI_Send_init, [23](#)
 - AMPI_Sendrecv_replace, [23](#)
 - AMPI_Start, [23](#)
 - AMPI_Startall, [23](#)

ampi_check_tape_size, [21](#)
 ampi_print_tape, [22](#)
 ampi_print_tape_entry, [22](#)
ampi_tape.hpp, [23](#)

INT64
 ampi_interface.h, [15](#)

Bibliography

- [1] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ in-depth series. Addison-Wesley, 2001.
- [2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, et al. LAPACK Users' Guide, Release 2.0. *SIAM, Philadelphia*, 326:327, 1995.
- [3] B. Averik, R. Carter, and J. Moré. The Minpack-2 Test Problem Collection (Preliminary Version). Technical Report 150, Mathematical and Computer Science Division, Argonne National Laboratory, Argonne, IL, 1991.
- [4] S. Balay, J. Brown, K. Buschelman, W. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2012. <http://www.mcs.anl.gov/petsc>.
- [5] C. Bendtsen and O. Stauning. FADBAD, A Flexible C++ Package for Automatic Differentiation. *Department of Mathematical Modelling, Technical University of Denmark*, 1996.
- [6] C. Bendtsen and O. Stauning. TADIFF, A Flexible C++ Package for Automatic Differentiation. Technical report, Citeseer, 1997.
- [7] C. Bischof, A. Carle, P. Khademi, and A. Maurer. The ADIFOR 2.0 System for Automatic Differentiation of Fortran 77 Programs. *IEEE Comp. Sci. & Eng.*, 3(3):18–32, 1996.
- [8] M. Bücker, B. Lang, D. an Mey, and C. Bischof. Bringing Together Automatic Differentiation and OpenMP. In *Proceedings of the 15th international conference on Supercomputing, ICS '01*, pages 246–251, New York, NY, USA, 2001. ACM.
- [9] M. Bücker, A. Rasch, and A. Vehreschild. Automatic Generation of Parallel Code for Hessian Computations. In M. S. Mueller, B. M. Chapman, B. R. de Supinski, A. D. Malony, and M. Voss, editors, *OpenMP Shared Memory Parallel Programming, Proceedings of the International Workshops IWOMP 2005 and IWOMP 2006, Eugene, OR, USA, June 1–4, 2005, and Reims, France, June 12–15, 2006*, volume 4315 of *Lecture Notes in Computer Science*, pages 372–381, Berlin / Heidelberg, 2008. Springer.
- [10] A. Carle and M. Fagan. Automatic Differentiation of Algorithms. chapter Automatically Differentiating MPI-1 datatypes: The Complete Story, pages 215–222. Springer-Verlag New York, Inc., New York, NY, USA, 2002.
- [11] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and Language Specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

- [12] B. Christianson. Reverse Accumulation and Attractive Fixed Points. *Optimization Methods and Software*, 3(4):311–326, 1994.
- [13] G. Corliss, C. Faure, A. Griewank, L. Hascoet, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*. Computer and Information Science. Springer, New York, 2002.
- [14] C. Faure and P. Dutto. Extension of Odyssée to the MPI Library - Reverse Mode -. Technical Report RR-3774, Oct. 1999.
- [15] C. Faure and P. Dutto. Extension of Odyssée to the MPI Library -Direct mode-. Technical Report RR-3715, June 1999.
- [16] M. Förster. *Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP*. Springer Fachmedien Wiesbaden, 2014.
- [17] A. Griewank. Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992.
- [18] A. Griewank et al. On Automatic Differentiation. *Mathematical Programming: recent developments and applications*, 6:83–107, 1989.
- [19] A. Griewank, D. Juedes, and J. Utke. Algorithm 755: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, 1996.
- [20] A. Griewank and A. Walther. Algorithm 799: Revolve: An Implementation of Checkpoint for the Reverse or Adjoint Mode of Computational Differentiation. *ACM Transactions on Mathematical Software*, 26(1):19–45, 2000. Also appeared as Technical University of Dresden, Technical Report IOKOMO-04-1997.
- [21] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105. SIAM, Philadelphia, PA, 2nd edition, 2008.
- [22] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, 1999.
- [23] J. W. Hanson, J. S. Caviness, and C. Joseph. Analytic Differentiation by Computer. *Communications of the ACM*, 5(6):349–355, 1962.
- [24] L. Hascoët, R.-M. Greborio, and V. Pascual. Computing Adjoints by Automatic Differentiation with TAPENADE. In B. Sportisse and F.-X. Le Dimet, editors, *École INRIA-CEA-EDF “Problèmes non-linéaires appliqués”*. Springer, 2005.
- [25] L. Hascoët, U. Naumann, and V. Pascual. ”To Be Recorded” Analysis in Reverse-Mode Automatic Differentiation. *Future Generation Computer Systems*, 21(8):1401 – 1417, 2005.
- [26] M. Heath. *Scientific Computing. An Introductory Survey*. McGraw-Hill, New York, 1998.
- [27] J.-M. Hervouet. TELEMAT Modelling System: An Overview. *Hydrological Processes*, 14(13):2209–2210, 2000.
- [28] T. Hilbrich, B. R. de Supinski, M. Schulz, and M. S. Müller. A Graph Based Approach for MPI Deadlock Detection. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS ’09, pages 296–305, New York, NY, USA, 2009. ACM.

- [29] P. Hovland and C. 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.
- [30] J. Huber, U. Naumann, O. Schenk, E. Varnik, , and A. Wächter. Algorithmic Differentiation and Nonlinear Optimization for an Inverse Medium Problem.
- [31] E. Kalnay. *Atmospheric Modeling, Data Assimilation, and Predictability*. Cambridge University Press, 2003.
- [32] G. Kedem. Automatic Differentiation of Computer Programs. Technical report, DTIC Document, 1976.
- [33] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.
- [34] G. Leibniz and J. Child. *The Early Mathematical Manuscripts of Leibniz*. Cosimo, Incorporated, 2008.
- [35] J. Lotz, K. Leppkes, and U. Naumann. dco/c++ - Derivative Code by Overloading in C++. Technical Report AIB-2011-06, RWTH Aachen, May 2011.
- [36] J. Lotz, U. Naumann, M. Sagebaum, and M. Schanen. Discrete Adjoints of PETSc through dco/c++ and Adjoint MPI. In *Euro-Par 2013 Parallel Processing*, pages 497–507. Springer Berlin Heidelberg, 2013.
- [37] L. D. Luca and R. Musmanno. A Parallel Automatic Differentiation Algorithm for Simulation Models, 1997.
- [38] U. Merkel, J. Riehme, and U. Naumann. Reverse Engineering of Initial & Boundary Conditions with TELEMAT and Algorithmic Differentiation. *TUC 2013*, page 25, 2013.
- [39] MPI Forum. MPI: A Message-Passing Interface Standard. Version 1.0, November 1992. available at: <http://www.mpi-forum.org> (Dec. 2013).
- [40] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009. available at: <http://www.mpi-forum.org> (Dec. 2013).
- [41] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.0, September 21th 2012. available at: <http://www.mpi-forum.org> (Apr. 2014).
- [42] U. Naumann. Optimal Jacobian Accumulation is NP-complete. *Math. Prog.*, 112:427–441, 2006.
- [43] U. Naumann. Call Tree Reversal is NP-Complete. In C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors, *Advances in Automatic Differentiation*, number 64 in LNCSE, pages 13–22. Springer, Berlin, 2008.
- [44] U. Naumann. DAG Reversal is NP-complete. *J. Discr. Alg.*, 7:402–410, 2009.
- [45] U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation*. Software, Environments, and Tools. SIAM, Philadelphia, PA, 2012.

- [46] 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*, pages 316–321, Berlin, Heidelberg, 2008. Springer-Verlag.
- [47] U. Naumann, J. Lotz, K. Leppkes, and M. Towara. Algorithmic Differentiation of Numerical Methods: Tangent and Adjoint Solvers for Parameterized Systems of Nonlinear Equations. *Transactions on Mathematical Software*, 2014. to appear.
- [48] U. Naumann and J. Riehme. A Differentiation-Enabled Fortran 95 Compiler. *ACM Transactions on Mathematical Software*, 31(4):458–474, 2005.
- [49] U. Naumann and J. Riehme. Computing Adjoints with the NAGWare Fortran 95 Compiler. In M. Bücker, G. Corliss, U. Naumann, P. Hovland, and B. Norris, editors, *Automatic Differentiation: Applications, Theory, and Implementations*, volume 50 of *Lecture Notes in Computational Science and Engineering*, pages 159–169. Springer Berlin Heidelberg, 2006.
- [50] I. Newton and D. Whiteside. *The Mathematical Papers of Isaac Newton*.: Number v. 7; v. 1691-1695 in The Mathematical Papers of Sir Isaac Newton. Cambridge University Press, 2008.
- [51] J. Nocedal and S. Wright. *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, 2006.
- [52] P. Heimbach and C. Hill and R. Giering. An Efficient Exact Adjoint of the Parallel MIT General Circulation Model, Generated via Automatic Differentiation. *Future Generation Computer Systems*, 21(8):1356–1371, 2005.
- [53] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1996.
- [54] L. B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer, Berlin, 1981.
- [55] A. Rasch, M. Bücker, and C. Bischof. Automatic Computation of Sensitivities for a Parallel Aerodynamic Simulation. In C. Bischof, M. Bücker, P. Gibbon, G. Joubert, T. Lippert, B. Mohr, and F. Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications*, volume 15 of *Advances in Parallel Computing*, pages 303–310, Amsterdam, The Netherlands, 2008. IOS Press. Co-published with NIC-Directors as Volume 38 of NIC Series, 2007.
- [56] F. Rauser, J. Riehme, K. Leppkes, P. Korn, and U. Naumann. On the Use of Discrete Adjoints in Goal Error Estimation for Shallow Water Equations. *Procedia Computer Science*, 1(1):107–115, 2010.
- [57] J. Riehme, U. Naumann, and B. Christianson. The Differentiation-Enabled NAGWare Fortran Compiler. *PAMM*, 7(1):1140207–1140208, 2007.
- [58] M. Sagebaum, N. R. Gauger, U. Naumann, J. Lotz, and K. Leppkes. Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries. Technical Report AIB-2013-04, RWTH Aachen, Feb. 2013.
- [59] M. Schanen, M. Foerster, B. Gendler, and U. Naumann. Compiler-based Differentiation of Numerical Simulation Codes. In *ICCGI 2011, The Sixth International Multi-Conference on Computing in the Global Information Technology*, pages 105–110, 2011.
- [60] M. Schanen, U. Naumann, and L. Hascoët. Interpretative Adjoints for Numerical Simulation Codes using MPI. In *Procedia Computer Science*, pages 1819–1827. Elsevier, 2010.

- [61] S. Sharma, S. Vakkalanka, G. Gopalakrishnan, R. Kirby, R. Thakur, and W. Gropp. A Formal Approach to Detect Functionally Irrelevant Barriers in MPI Programs. In A. Lastovetsky, T. Kechadi, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 265–273. Springer Berlin Heidelberg, 2008.
- [62] J. Utke, L. Hascoët, P. Heimbach, C. Hill, P. Hovland, and U. Naumann. Toward Adjoinable MPI. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*, Washington, DC, USA, 2009. IEEE Computer Society.
- [63] 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*, 34(4):18:1–18:36, 2008.

Index

MPI_Barrier, 59
MPI_Accumulate, 60
MPI_Broadcast, 45
MPI_Gatherv, 45
MPI_Gather, 45
MPI_Get, 60, 63
MPI_Irecv, 40, 63
MPI_Isend, 40
MPI_Put, 60
MPI_Recv_init, 43
MPI_Recv, 37
MPI_Request_free, 43
MPI_Scatterv, 45
MPI_Scatter, 45
MPI_Send_init, 43
MPI_Send, 37
MPI_Start, 43
MPI_Test, 44
MPI_Wait, 40, 43
MPI_Win_complete, 60
MPI_Win_fence, 60, 62
MPI_Win_lock, 59, 60
MPI_Win_post, 60
MPI_Win_start, 60
MPI_Win_unlock, 60
MPI_Win_wait, 60
TEST_ANY, 14, 59
WAIT_ANY, 14, 44, 59

access epoch, 59
access restriction axiom, 61
AD, *see* Algorithmic Differentiation
adjoint code, 25
adjoint model, 19, 83
adjoints
 continuous, 15, 35, 104
 discrete, 15, 35, 104, 117
Algorithmic Differentiation, 11, 18, 36
Amdahl's law, 31

antiwait, 43
assignment, 11
 mathematical, 12
 operation, 12
association by address, 23, 85
association by name, 23, 85

back end, 22
barrier, 59
 irrelevant, 59
 relevant, 59
blocking communication, 37, 44
broadcast, 45

Cartesian basis vector, 21
chain rule, 12, 18
checkpointing, 34, 117
closure, 14
cluster, 17
collective communication, 45
CompAD, 96
compiler, 22
concatenation, 85
concurrency, 28, 62
conjunction, 84
correctness, 14
cost function, 93

data flow reversal, 22
dcc, 84, 89
dco, 104
derived data types, 85
derived type, 21
differentiable, 12
differential calculus, 12
directional derivative, 21
disk tape, 91
distributed memory, 26

efficiency, 14, 32

- equal sign, 12
- extended SAC, 27
- feasibility, 14
- Fence, 62
- finite differences, 20, 21
- Fortran, 18, 95, 122
- forward over reverse, 84
- forward section, 22
- front end, 22
- Get, 62
- Gradient Descent, *see* Steepest Descent
- halo exchange, 93
- handwritten code, 18
- hardware, 31
- harvesting, 21
- heat equation, 92
- Heisenberg, 91
- higher order, 83
- intrinsic function, 38
- Jacobian, 19–21
- joint reversal, 87
- local memory, 61
- lock, 29, 36, 42
- machine precision, 20
- memory, 117
- memory hierarchy, 34
- memory layout
 - contiguous, 23
 - interleaved, 23
- memory wall, 31, 34
- Message-Passing Interface, 17
- MPI 1.1, 14, 44
- MPI 2.2, 14
- MPI 3.0, 117
- NAG Fortran Compiler, 96
- network, 18
- node, 18
- nonblocking communication, 40, 44, 59, 117
- nondeterminism, 14, 27, 29, 30, 44
- nullification, 25, 38
- one-sided communication, 59
- OpenMP, 18
- operator overloading, 22, 85
- optimization, 19
- origin, 60
- parallel systems, 31
- partial derivative, 20
- partitioned global address space, 26, 36, 84
- pattern, 31
- persistent communication, 43
- PETSc, 15, 104
- PGAS, *see* partitioned global address space
- point-to-point communication, 37, 40, 59, 93
- problem statement, 14
- public window, 61
- Put, 66
- reapplication, 18, 83
- recomputation, 35
- reference by address, 11
- reference by name, 11
- reverse section, 22
- rounding error, 20
- seeding, 21
- shadowing, 21, 85
- shared-memory, 18, 26
- single assignment code, 24, 36
- Sisyphe, 15, 95
- Software and Tools for Computational Engineering, 11, 104
- source transformation, 22
- speedup, 32
- stack, 23
- STCE, *see* Software and Tools for Computational Engineering
- Steepest Descent, 19, 95
- symbolic differentiation, 18
- synchronization, 60
 - collective, 60
 - passive, 14, 29, 59, 60
 - selective, 60
- tangent, 21, 85
- tangent-linear code, 24
- tangent-linear model, 20, 83, 84
- tangent-linear MPI, 85
- tape, 23
- target, 60
- TMPI, *see* tangent-linear MPI
- tuple, 85

Unified Parallel C, 26
UPC, *see* Unified Parallel C
update, 61
user defined operations, 85

vector mode, 21

well-behaved, 61
wildcards, 14, 44, 59