Discrete Adjoints of PETSc through dco/c++ and Adjoint MPI

Johannes Lotz, Uwe Naumann, Max Sagebaum, and Michel Schanen[⋆]

LuFG Informatik 12: Software and Tools for Computational Engineering
RWTH Aachen University, Germany
{lotz,naumann,schanen}@stce.rwth-aachen.de
http://www.stce.rwth-aachen.de
Department of Mathematics and Center for Computational Engineering Science
RWTH Aachen University, Germany
sagebaum@mathcces.rwth-aachen.de
http://www.mathcces.rwth-aachen.de

Abstract. PETSc's [1] robustness, scalability and portability makes it the foundation of various parallel implementations of numerical simulation codes[1]. We formulate a least squares problem using a PETSc implementation as the model function and rely on adjoint mode Algorithmic Differentiation (AD) [2] for the accumulation of the derivative information. Various AD tools exist that apply the adjoint model to a given C/C++ code, while none is able to differentiate MPI [3] enabled code. We solved this by combining `dco/c++` and the Adjoint MPI library, leading to a fully discrete adjoint implementation of PETSc. We want to underline that this work differs from accumulating derivative information through AD for PETSc algorithms (see e.g. [4]). We compute derivative information of PETSc itself opening up the possibility of an enclosing optimization problem (as needed, e.g., by [5]).

1 Motivation

Our case study is the two-dimensional Bratu equation,

$$\nabla^2 \mathbf{u} = -\lambda \exp(\mathbf{u}), \tag{1}$$

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

$$u = b_i \text{ for } x = 0, x = 1, y = 0, y = 1$$

at the borders of the two dimensional square. 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 [6] 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.

⋆ This work was supported by the Fond National de la Recherche of Luxembourg under grant PHD-09-145.

1 See Applications/Publications [1].

The differential equation in (1) is solved by discretization using finite difference 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. We now formulate an optimization problem over the original solution to the Bratu equation.

We distinguish the following 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, we have observed values $\mathbf{u}^{ob}$ that allow us to rate the correctness of our model. This fact may be formulated as a least squares problem where we want to minimize the difference between the computed and observed values 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

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

is implemented in PETSc using its non-linear solver SNES for the computation of $\mathbf{u}$. We used the code found in example 5 of the SNES tutorials. The additional implementation of the cost functional S for the least squares problem is straightforward.

We now describe step by step how we generated an implementation of the gradient ∇F of PETSc that enables us to feed a gradient based solution method.

In Sect. 2 we present Algorithmic Differentiation (AD) as our method of choice for the gradient computation. Additionally, we provide a brief overview of PETSc's code structure and where challenges arise. In Sect. 3 we provide a technical description of our AD overloading tool `dco/c++`. It is used to generate the adjoint code of PETSc. However, PETSc relies on the BLAS [2] and LAPACK [3] library for the sequential computation. We provide a methodical description of how we achieved adjoints of these library. Sect. 4 covers the adjoining of the MPI communication using our in-house developed AMPI library.

2 Background

We resort to a Steepest Descent or Gradient Descent algorithm as a proof of concept in order to minimize the residual S. As the name hints, it relies on the

[2] `http://www.netlib.org/blas/`
[3] `http://www.netlib.org/lapack/`

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 $\mathbf{b}$. In order to acquire first-order gradient information by finite difference, one would have to perturb each of the n inputs b_i, $0 < i \leq n$. This requires $n+1$ runs of F. Once for each perturbation and once for the original computation. For $n \gg 1$ the gradient accumulation potentially becomes computationally infeasible.

2.1 Algorithmic Differentiation

AD is the chain rule of differential calculus applied symbolically to each statement of a given code. This can be done automatically by resorting to a compiler or by overloading the mathematical operations in a source code.

For a multivariate scalar function (e.g. calculation of cost function S) $y = F(\mathbf{x})$, $\mathbb{R}^n \to \mathbb{R}$ the chain rule may be used in two ways. First, the straightforward application leading to the tangent-linear model $\dot{y} = \nabla F(\mathbf{x}) \cdot \dot{\mathbf{x}}$, where $\dot{\mathbf{y}}$ is the directional derivative of the output y with respect to inputs $\mathbf{x}$ in direction $\dot{\mathbf{x}}$. Notice that in order to accumulate the entire gradient we have to iteratively set $\dot{\mathbf{x}}$ to each of the n Cartesian basis vector in $\mathbb{R}^n$ leading to a runtime cost of $\mathcal{O}(n) \cdot cost(F)$.

Second, the adjoint model based on the associativity of the chain rule:

$$\bar{\mathbf{x}} = \bar{\mathbf{x}} + \nabla F\left(\mathbf{x}\right)^{\mathsf{T}} \cdot \bar{y}. \tag{2}$$

$\bar{\mathbf{x}}$ are called the adjoints of the inputs $\mathbf{x}$, whereas $\bar{y}$ is the adjoint of the output y. Notice that the computation of the adjoints is in reverse order of the computation of the values, thus requiring a complete data flow reversal of a program. The *forward section* consists of the computation of the values, whereas the *reverse section* computes the adjoints while using the values saved in the forward section. Furthermore, it is essential to understand that with one output, the gradient accumulation is achieved by one adjoint computation of the corresponding adjoint code reducing the runtime complexity to $\mathcal{O}(1) \cdot cost(F)$. Of course, the constant factor may still be considerable. However, given the independence of the runtime from the input size n, the adjoint model may end up as the only feasible solution. All the other options for a gradient accumulation, finite difference and tangent-linear model, will become computationally too expensive at some input size $n \gg 1$.

The discrete approach with the tangent-linear and adjoint model may be applied through handwritten code, where the original code is transformed statement-wise into the derivative code. This work is very tedious and error prone. AD tools do this mechanical work semi-automatically. Source transformation tools are similar to the handwritten transformation, whereas operator overloading tools achieve the same goal by overloading every intrinsic operation.

Compared to the numerical method of finite difference, there are two advantages. One, we are able to extract exact derivatives with up to machine precision, whereas the precision of finite difference is largely dependent on the spacing factor h. Second, a similar method to the adjoint model is not available while using finite difference. It shifts the complexity from $\mathcal{O}(n) \cdot cost(F)$ to $\mathcal{O}(m) \cdot cost(F)$, with n and m being the inputs and outputs, respectively. This unique feature makes the adjoint model crucial for optimizations where the cost function is scalar, while being influenced by a large number of parameters.

2.2 PETSc

As mentioned above, we use a tutorial example as a test case in PETSc. We chose tutorial 5 of the SNES solver. It implements a solution of the Bratu differential equation (1). We had to make a few amendments to the code which will be explained later in this paper. The source code is available on our website [4].

The parameter λ is set to 6. We used a value of 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.

We compiled PETSc with the default options, although we provide a custom BLAS and LAPACK library described in Sect. 3.1 and 3.2. 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 in two types. For one there are several collective invocations of Allreduce. The other one is the persistent MPI communication in src/vec/vec/utils/vpscat.c. Both will be dealt with in Sect. 4.

3 Adjoint Model Generation Using `dco/c++`

`dco/c++` is implementing AD by overloading in C++. The range of capabilities covered by `dco/c++` is driven by various applications and research subjects. Current projects are in the area of financial engineering, atmospheric physics, or fluid mechanics. `dco/c++` is also used in research on the generation of discrete adjoints using parallel environments, in particular OpenMP and MPI as, e.g., used in PETSc.

The objective is to provide an efficient and robust tool for the computation of projections of derivatives of arbitrary order of a function given as an implementation in C/C++, while focusing on the adjoint mode. Additionally, the capability of coupling the robust overloading technique with optimized computer generated or hand-written external computations of adjoint projections is provided. This is used extensively for the adjoining of BLAS.

During various collaborative research and development projects, we were able to compute fast adjoints for real world applications. In some cases [7] we achieved a factor of roughly 3.5 for the ratio

$$R = \frac{\text{Run time of one adjoint computation}}{\text{Run time of one function evaluation}} .$$

[4] `https://www.stce.rwth-aachen.de/trac/petsc`

For being able to achieve such a factor, we make heavy use of the C++ template engine and we exploit algorithmical and mathematical insight, e.g., statement-level preaccumulation.

A standard run for computing adjoints using dco/c++ consists of a so called *forward run* being the forward section and generating a *tape* followed by exactly one *reverse run* being the reverse section in case of a scalar cost function. This structure follows from the requirement of making all computed values available in reverse order (data flow reversal). The forward run saves all required information in the tape used during the reverse run. This structure is also applied to the part of PETSc, which is to be differentiated. As PETSc is using BLAS as well as LAPACK library routines for numerical methods, we have to deal with those libraries, too. The treatment is sketched in the following subsections.

3.1 BLAS

BLAS (Basic Linear Algebra Subprograms) [8] is used by PETSc for non-parallel tasks and is a set of basic routines operating on scalars, vectors and matrices of type 'double' including, e.g., scaling of a vector by a scalar, or matrix vector products. The implementation of those basic operations is typically optimized for performance by manufacturers for the specific hardware that is running the computation. It is therefore desirable to stick to the supported BLAS implementation – at least during the forward run of the overloaded program execution. We therefore do not overload the original routines to avoid killing cache performance. Due to the reasonable amount of different routines, we provide hand-written adjoint routines of the BLAS routines used in PETSc, which rely on the original BLAS implementation whenever possible. This includes the computation of the function values during the forward run. We expect this to produce at least binary identical results to the non-overloaded function run, which is desirable for verification purposes. Additionally we expect a better runtime compared to a reimplementation of the BLAS routines.

3.2 LAPACK

LAPACK (Linear Algebra PACKage) [9] is also used by PETSc for non-parallel tasks. In contrast to BLAS this library implements algorithms for general linear algebra problems like linear systems (called, e.g., by PETSc's SNES) or eigenvalue problems. The implementation of LAPACK includes calls to BLAS routines for all basic vector and matrix operations. Because LAPACK comes with a large number of different specialized functions (in total ca. 1600 routines) we chose this time to differentiate those routines in a black box way; change the datatypes in LAPACK to our dco/c++ datatype. Again, hand-written and optimized derivative code would yield better performing code. We therefore aim to provide those optimized adjoint routines of LAPACK step by step in a project-driven fashion. As in BLAS, those hand-written routines should of course reuse the original LAPACK ones. This is useful for efficiency reasons as well as for keeping the code up-to-date with the current LAPACK implementation. This will include also the possibility of switching to GPU-enabled LAPACK (see e.g. MAGMA [10]) or other LAPACK implementations.

4 Adjoint MPI and PETSc

The reversal of MPI [11] communication in AD is directly related to the implied data flow reversal. The adjoint MPI library is specifically designed to reverse the MPI communication. It is separated from the actual AD tool involved (here `dco/c++`) and therefore only traces the communication itself. The actual data (values and adjoints) is stored through an AMPI interface in the data structures of the AD tool.

Tracing communications amounts to tracing data dependence in between the processes' address space. As a formalism, we apply the PGAS (Partitioned Global Address Space) notation to the MPI communication.

Table 1. Adjoint communication of an MPI_Send and MPI_Recv between two processes P1 and P2

MPI	MPI_Send(x,P2), MPI_Recv(x,P1)
PGAS	P2.x=P1.x
Adjoint PGAS	P1.$\bar{x}$+=P2.$\bar{x}$; P2.$\bar{x}$=0
Adjoint MPI	MPI_Recv($\bar{x}$,P2), MPI_Send($\bar{x}$,P1)

Sending and receiving data are actual assignments in PGAS that need to be adjoined according to the incremental adjoint model (2). Every communication is transformed into an *incremental communication*. The adjoint MPI library itself as well as the consequences of the incremental communication and the data flow reversal have been subject to various papers covering blocking, non-blocking, collective [12] and one-sided communication [13]. 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 adjoined. 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. And second collective communication in the shape of a MPI_Allreduce.

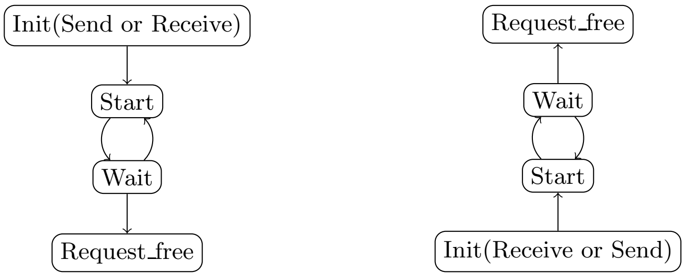
4.1 Persistent Communication

The reversal of non-blocking communication has already been subject of several publications [14,15]. They cover the reversal of Isend/Wait and Irecv/Wait pairs. In a nutshell, the reversal logic of the blocking send and receive is preserved. The send becomes a receive and the receive becomes a send. However, the ordering of the communication pairs Isend/Wait, Irecv/Wait is reversed. In the reverse section the Wait becomes either an Isend or Irecv whereas the Isend and Irecv both become a Wait. There are other effects which will not be discussed in this paper.

In our PETSc code we do not have pairs of non-blocking Isend/Wait and Irecv/Wait, but triplets of persistent Recv_init/Start/Wait and Send_init/Start/Wait calls. Persistent communication is similar in logic to the non-blocking pairs. 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 non-blocking communication. That logic should be preserved in adjoint MPI. In particular, we want to correctly and efficiently adjoin the multiple MPI_Start calls.



(a) Forward Communication (b) Reverse Communication

Fig. 1. Adjoint communication of a Init, Start, Wait and Request_free

As mentioned in the previous section, the core principle of AD by overloading is that we tape each step of the forward section in order to generate a correct adjoint section. The same holds for adjoint MPI. We trace each of the involved routines in the forward section along with its arguments. 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:

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. We allocate the adjoint buffer here and call MPI_Send/Recv_init 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 we will start here the adjoint communication 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 non-blocking communication) at the MPI_Start.

Start. MPI_Start marked the start of actual data communication. In the adjoint case this amounts to a Wait. We have to make 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. We may release the requests with an MPI_Request_free.

4.2 Collective Communication

For the reduction of residuals and certain simulation parameters, several reduction operations are involved in PETSc. They are all of the Allreduce kind where the result is accessible on all the processes. The operations are limited to MPI_MAX, MPI_SUM.

Maximum Operation

Value	Adjoint
$(y, m) = \max_{i=1}^{p} x_i,$ with $x_m = y$	$\bar{x}_m = \bar{y};\ \bar{y} = 0$

For the maximum operation we need to save the process rank m of the element that is the maximum. Only this element has a non zero partial derivative with respect to the output. Hence, in the reverse section the adjoint of the result on all processes is summed up and only sent to the process that had the maximum value.

Sum Operation

Value	Adjoint
$y = \sum_{i=1}^{p} x_i$	$\bar{x}_i += \bar{y};\ \bar{y} = 0$

In case of a sum, the adjoint has to be propagated to all processes involved. The adjoint communication amounts to a broadcast.

5 Results

As has been mentioned before, this paper is meant as a proof of concept that fully discrete adjoints of PETSc are possible with the available tools. However, a conclusion on the performance of discrete adjoint must be dismissed at this time. To understand why, we provide the following benchmarks made on the RWTH Aachen University cluster. All tests were conducted on the MPI nodes of the Bull HPC-Cluster. They are 2-socket systems equipped with Intel Westmere EP processors with 24GB memory each.

	Memory Usage	Original RT (s)	Adjoint Mode RT (s)
1 Process	16GB	0.22	3.8
2 Processes	8GB	0.14	3.45
4 Processes	4GB	0.09	2.3

With a grid of only 128x128 we reach a memory usage of 16GB in adjoint mode. The time it takes to trace the forward section as well as the adjoint computation is 3.8s. The problem size is a hard limit. However, with a runtime of only 3.8s seconds we are far from any real world application. Increasing the number of processes for the 128x128 to more than 4 is not reasonable either, since the overhead due the communication takes its toll. Therefore we are quite limited in the possible benchmarks. In the end they should prove that there is some speedup and that our method is on the right track.

To remedy the situation, one has two choices. Either apply a checkpointing scheme [16] or replace the computationally most expensive part, namely the linear solver. Replacing the linear solver with a continuous computation of the adjoints similar to the BLAS routines in this paper will considerably reduce the memory hit. This is the next step outlined in the coming outlook

6 Summary

We proved that the combination of `dco/c++` and Adjoint MPI is robust enough to compute semi-automatic discrete adjoints of PETSc. Four distinct steps were necessary. First, `dco/c++` had to be applied to PETSc by overloading all variables of type PetscReal and PetscScalar. Second, BLAS had to be continuously adjoined by writing adjoint BLAS functions by hand. Third, the LAPACK routines were adjoined using again `dco/c++`. Finally, the adjoint MPI library was used to adjoin all the MPI communication.

7 Outlook

While relying on AD tools, discrete adjoints are the straightforward way of adjoining a given code. However, they do not always match the desired derivative information of the simulated phenomenon. They only represent a differentiated algorithm that models a given phenomenon. Hence, applying for example a continuously differentiated linear solver in PETSc might yield different results and runtime behaviour. However these results may better fit the actual simulated phenomenon. This requires formulating the adjoint linear system that computes the adjoints of the original linear system. Unfortunately, this requires considerable changes to the PETSc code base, but definitely worth further investigations. For the time being only the BLAS routines are adjoined continuously.

The current case study should be superseded by a real world application in the future. The driving application behind this project are discrete adjoints of PADGE, an adaptive discontinuous Galerkin solver for 3D turbulent flow developed by the German Aerospace Center (DLR).

References

1. Balay, S., Brown, J., Buschelman, K., Gropp, W.D., Kaushik, D., Knepley, M.G., McInnes, L.C., Smith, B.F., Zhang, H.: PETSc Web page (2012), `http://www.mcs.anl.gov/petsc`
2. Naumann, U.: The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation. Software, Environments, and Tools. SIAM, Philadelphia (2012)
3. MPI Forum: MPI: A Message-Passing Interface Standard. Version 2.2 (September 4, 2009), `http://www.mpi-forum.org` (December 2009)
4. Hovland, P.D., McInnes, L.C.: Parallel Simulation of Compressible Flow using Automatic Differentiation and PETSc. Parallel Computing 27(4), 503–519 (2001); Parallel Computing in Aerospace
5. Sagebaum, M., Gauger, N.R., Naumann, U., Lotz, J., Leppkes, K.: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries. Technical Report AIB-2013-04, RWTH Aachen (February 2013)
6. Averick, B.M., Carter, R.G., Mor, J.J.: The Minpack-2 Test Problem Collection (1991)
7. Ungermann, J., Blank, J., Lotz, J., Leppkes, K., Hoffmann, L., Guggenmoser, T., Kaufmann, M., Preusse, P., Naumann, U., Riese, M.: A 3-D Tomographic Retrieval Approach with Advection Compensation for the Air-Borne Limb-Imager GLORIA. Atmos. Meas. Tech. 4(11), 2509–2529 (2011)
8. Lawson, C.L., Hanson, R.J., Kincaid, D.R., Krogh, F.T.: Basic Linear Algebra Subprograms for Fortran Usage. ACM Transactions on Mathematical Software (TOMS) 5(3), 308–323 (1979)
9. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S.: Lapack users' guide, release 2.0, vol. 326, p. 327. SIAM, Philadelphia (1995)
10. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA projects. Journal of Physics: Conference Series 180, 012037 (2009)
11. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message Passing Interface. MIT Press (1994)
12. Hovland, P., Bischof, C.: Automatic Differentiation for Message-Passing Parallel Programs. In: IPPS 1998: Proceedings of the 12th International Parallel Processing Symposium on International Parallel Processing Symposium. IEEE Computer Society, Washington, DC (1998)

13. Schanen, M., Naumann, U.: A Wish List for Efficient Adjoints of One-Sided MPI Communication. In: Träff, J.L., Benkner, S., Dongarra, J.J. (eds.) EuroMPI 2012. LNCS, vol. 7490, pp. 248–257. Springer, Heidelberg (2012)
14. Schanen, M., Naumann, U., Hascoët, L., Utke, J.: Interpretative Adjoints for Numerical Simulation Codes using MPI. Procedia Computer Science 1(1), 1819 –1827 (2010); ICCS 2010
15. Utke, J., Hascoët, L., Heimbach, P., Hill, C., Hovland, P., Naumann, U.: Toward Adjoinable MPI. In: Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium. IEEE Computer Society, Washington, DC (2009)
16. Griewank, A., Walther, A.: 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)