RWTH Aachen

Department of Computer Science
Technical Report

A Framework for Proving Correctness of Adjoint Message Passing Programs

Uwe Naumann, Laurent Hascoët, Chris Hill, Paul Hovland, Jan Riehme, and Jean Utke

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2008-06

RWTH Aachen · Department of Computer Science · April 2008

A Framework for Proving Correctness of Adjoint Message Passing Programs

Uwe Naumann[1], Laurent Hascoët[2], Chris Hill[3], Paul Hovland[4]
Jan Riehme[5], and Jean Utke[4]

[1] Corresponding Author: LuFG Informatik 12 (Software and Tools for Computational Engineering) Department of Computer Science
RWTH Aachen University, 52056 Aachen, Germany
www: `http://www.stce.rwth-aachen.de`
email: `naumann@stce.rwth-aachen.de`
[2] Projet TROPICS, INRIA Sophia-Antipolis, France
[3] Department of Earth, Atmospheric, and Planetary Sciences, Massachusetts Institute of Technology, Cambridge, MA, USA
[4] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
[5] Department of Computer Science, University of Hertfordshire, Hatfield, UK

Abstract. Adjoint programs play a central role in modern numerical algorithms such as large-scale sensitivity analysis, parameter tuning, and general nonlinear optimization. They can be generated automatically by compilers. In such cases, the data flow of the original program needs to be reversed. If message passing is used, then any communication needs to be reversed, too. Crucial properties of the original program such as deadlock-freeness and determinism must be preserved in the adjoint code. A formalism for proving the correctness of compiler-generated adjoints is required but has been missing so far, to the best of our knowledge.
To rectify this situation, we propose a proof technique that relies on data dependences in partitioned global address space versions of the adjoint message-passing program. If the original program is deadlock-free, the transformation rules can be shown to be correct in the sense that the automatically generated adjoint program is also deadlock free while implementing the mathematical mapping from given independent inputs onto their corresponding adjoints correctly. As an example we discuss asynchronous unbuffered send/receive using MPI.

1 Adjoint Numerical Programs

Numerical simulation and optimization in computational science and engineering have gained significant importance over the past few decades. For example, our ability to understand physical, chemical, and biological processes has improved with the increased power of computational resources as well as with the deepened insight into mathematical and algorithmic issues. Numerical simulation programs map n *independent* inputs onto m *dependent* outputs (also referred to as the objectives). Often n is very large in comparison to m. The classical numerical approach to quantifying the sensitivities of those objectives with respect to the inputs through finite-difference quotients yields a computational complexity of $O(n)$. Note that certain high-end applications such as the simulation of ocean circulation [15] may have a runtime of several days to produce physically relevant results on the latest high-performance computing platforms. The number of independent inputs may reach values of the order of $n = 10^9$. Hence, forward sensitivity analysis requiring n runs of the simulation program is simply not feasible.

Adjoint methods and corresponding program transformation techniques have been developed to replace the dependence on n with that on the number of objectives m. If $m = 1$, then adjoint programs deliver the sensitivities of the objective with respect to all independent inputs at $O(1)$. Adjoint programs can be generated from a given numerical simulation program by a semantic program transformation technique known as *automatic differentiation* (AD) [11]. A large number of successful applications of AD to real-world problems in science and engineering have been reported in the literature. Refer, for example, to the proceedings of the five international conferences on AD held in 1991 [8], 1996 [2], 2000 [7], 2004 [4], and 2008 [3].

Adjoint numerical programs consist of two parts. The *augmented forward section* is an instrumented version of the original program containing statements to memorize certain intermediate values that are required for the correct (and efficient) evaluation of the adjoint program variables. The *reverse section* propagates values of adjoint program variables in the opposite direction of the original data flow. Optimal data-flow reversal is NP-complete [16, 17]. It involves the reversing of the flow of control (which implies reversing the order of the statements within basic blocks) and generating the corresponding adjoint statements. Proofs of correctness of sequential adjoint programs are based on the chain rule of differential calculus and, in particular, on its associativity. Refer to [11] for a comprehensive discussion of the mathematical foundations of adjoint programs. The purpose of this paper is served best by introducing adjoint programs by means of an example.

Example: Consider the following simple code fragment that is assumed to implement a function $y = f(x)$, depending on some condition c.

$$
\begin{array}{l}
y = \sin(x) \\
\text{if}(c) \\
\qquad x = y + 1 \\
\text{else} \\
\qquad x = y - 1 \\
y = \cos(x)
\end{array}
$$

As an input to the adjoint routine $\bar{f}(x, \bar{x}, \bar{y})$ shown in Figure 1, the variable $\bar{x}$ should be initialized to zero in order to obtain $\bar{x} = \bar{y} \cdot f'(x)$ on output. The gradient $f'(x)$ at point x (a single scalar partial derivative in this simple case) is obtained by initializing $\bar{y} = 1$ on input to $\bar{f}$.

We use two stacks for a store-all approach to data-flow reversal. The first stack, S_d, is used to store values that are required for evaluating the partial derivatives of some assignments and that are (possibly[6]) overwritten by some subsequently executed assignment. For example, the value of x at input is required to compute the partial derivative of the left-hand side of the first assignment with respect to x as the argument of the intrinsic call $\sin(x)$. Hence, it needs to be stored before being overwritten by the second or third assignment. The value of y right before the fourth assignment is not required for evaluating

[6] Substantial conservative static data-flow analysis is usually involved in deciding which values to store. See, e.g., [12].

partial derivatives of any preceding assignment and therefore does not need to be stored.

The second stack, S_c, contains information on the original flow of control that is to be reversed. For example, we need to remember which branch of the if-statement is executed. One solution is to push one or zero depending on the condition c being true or false. The augmented forward section is shown in Figure 1(a). The adjoint statements that correspond to a given original assignment

$$
\begin{array}{l}
y = \sin(x) \\
\text{if}(c) \\
\quad \text{push}(S_d, x) \\
\quad x = y + 1 \\
\quad \text{push}(S_c, 1) \\
\text{else} \\
\quad \text{push}(S_d, x) \\
\quad x = y - 1 \\
\quad \text{push}(S_c, 0) \\
y = \cos(x)
\end{array}
$$

(a)

$$
\begin{array}{l}
\bar{x} += -\sin(x) \cdot \bar{y};\ \bar{y} = 0 \\
\text{if}(\text{pop}(S_c)) \\
\quad \bar{y} += \bar{x};\ \bar{x} = 0 \\
\quad x = \text{pop}(S_d) \\
\text{else} \\
\quad \bar{y} += \bar{x};\ \bar{x} = 0 \\
\quad x = \text{pop}(S_d) \\
\bar{x} += \cos(x) \cdot \bar{y};\ \bar{y} = 0
\end{array}
$$

(b)

Fig. 1. Adjoint Code = Augmented Forward Section (a) + Reverse Section (b)

(e.g., the last one) increment the adjoints ($\bar{x}$) of all program variables (x) on the original right-hand side with the product of the adjoint ($\bar{y}$) of the program variable (y) on the original left-hand side with the corresponding local partial derivative ($\cos(x)$). The adjoint of the left-hand side needs then to be reset to zero. Correctness of these rules follows immediately from the chain rule applied to program variables that can represent various instances due to overwrites. The order of the statements is reversed in the reverse section. Correct reversal of the flow of control is achieved through S_c. The reverse section of the example code is shown in Figure 1 (b).

This paper is motivated by the need for automatically generated adjoint versions of parallel programs that use message passing. Related work comprises [5, 6, 10, 13, 14, 20]. We describe a proof technique that allows us to show the correctness of adjoint message-passing programs. Usually a number of semantically equivalent adjoint versions can be generated for a given message-passing program. As developers of adjoint code compilers, we consider the scenario of a given transformation algorithm that needs to be proved right or wrong in the sense that correct adjoints are computed for arbitrary inputs.

2 Correctness of Adjoint Communication Patterns

We consider the *partitioned global address space (PGAS)* [9] version P_s of a message-passing program P involving n processes $p_1, \ldots, p_n$. In order for P_s to operate on the union of the n memory spaces all program variables are augmented with an additional dimension of length n. Communications are translated into *x-assignments* between augmented program variables belonging to disjoint address spaces. Auxiliary variables are introduced for buffered communication. Barriers in asynchronous communication yield a set of PGAS versions for a given message-passing program.

2.1 Example

The program

s_0
if $(myrank == 1)$ isend(a, r); s_1; if $(myrank == 2)$ irecv(b, r); s_2; wait(r)
s_3

with unspecified sequences of statements s_i for $i = 0, \ldots, 3$ yields the following set of constraints for the placement of the x-assignment χ:

$$s_0^1 < \chi; \quad s_1^2 < \chi; \quad \chi < s_3^1; \quad \chi < s_3^2 \quad .$$

These constraints lead to the following six PGAS codes:

$s_0;\ s_1^2;\ b^2 = a^1;\ s_1^1;\ s_2;\ s_3$
$s_0;\ s_1;\ b^2 = a^1;\ s_2;\ s_3$
$s_0;\ s_1;\ s_2^2;\ b^2 = a^1;\ s_2^1;\ s_3$
$s_0;\ s_1;\ s_2^1;\ b^2 = a^1;\ s_2^2;\ s_3$
$s_0;\ s_1^2;\ s_2^2;\ b^2 = a^1;\ s_1^1;\ s_2^1;\ s_3$
$s_0;\ s_1;\ s_2;\ b^2 = a^1;\ s_3$

The statements executed in section i by processor j are denoted by s_i^j. In this example we assume two processors. Note that $(s_i^1;\ s_i^2) = (s_i^2;\ s_i^1)$ as a result of the disjoint address spaces. Hence, the PGAS code $s_i;\ s_{i+1}$ yields the following six semantically equivalent sequential codes:

$s_i^1;\ s_{i+1}^1;\ s_i^2;\ s_{i+1}^2$
$s_i^2;\ s_{i+1}^2;\ s_i^1;\ s_{i+1}^1$
$s_i^1;\ s_i^2;\ s_{i+1}^1;\ s_{i+1}^2$
$s_i^1;\ s_i^2;\ s_{i+1}^2;\ s_{i+1}^1$
$s_i^2;\ s_i^1;\ s_{i+1}^1;\ s_{i+1}^2$
$s_i^2;\ s_i^1;\ s_{i+1}^2;\ s_{i+1}^1$

The partial order of the statements is induced by $s_i^j < s_{i+1}^j$. Any two statements from s_i^j and s_{i+1}^k can be executed in arbitrary order for $j \neq k$. Further combinations resulting from feasible (wrt. data dependence) switches of the x-assignment and statements in certain s_i^j lead to an exponential number of possible actual execution orders that need to be taken into account when proving properties of PGAS programs. For this example we observe that the original program must satisfy the restriction for *isend* that a^1 is not written by s_1 nor s_2.[7] Similarly, for *irecv* it must satisfy that b^2 is neither read nor written by s_2.

To prove the correctness of an adjoint of a message-passing program, we need to show that its adjoint PGAS versions are semantically equivalent to the PGAS versions of its adjoint. We do so by looking at all possible actual execution orders.

[7] a^1 must not be written by s_1^1 nor s_2^1. It is not written by s_1^2 nor s_2^2 due to the separate address spaces.

2.2 Case Study: Asynchronous Unbuffered Send/Receive

We present here a case study to illustrate the use of the proposed formalism. Similar proofs are required for a large number of communication patterns. We are analyzing all communication patterns used by our main target applications, including MITgcm (`mitgcm.org`) as well as ICON (`www.icon.enes.org`).

Proposition: *Let P be a message-passing program involving processes p_1 and p_2, and let the integer variable myrank contain the respective process identifiers. The communication pattern*

$$\begin{array}{l} s_{i-1};\ \text{if } (myrank == 1)\ \text{isend}(a, r);\ s_{i+1} \\ \dots \\ s_{j-1};\ \text{if } (myrank == 2)\ \text{irecv}(b, r);\ s_{j+1} \\ \dots \\ s_{k-1};\ \text{wait}(r);\ s_{k+1} \end{array}$$

in the forward section of the adjoint code yields

$$\begin{array}{l} \bar{s}_{k+1} \\ \text{if } (myrank == 2)\ \text{isend}(\bar{b}, r) \\ \text{if } (myrank == 1)\ \text{irecv}(t, r) \\ \bar{s}_{k-1} \\ \dots \\ \bar{s}_{j+1};\ \text{if } (myrank == 2)\ \text{wait}(r);\ \bar{b} = 0;\ \bar{s}_{j-1} \\ \dots \\ \bar{s}_{i+1};\ \text{if } (myrank == 1)\ \text{wait}(r);\ \bar{a} \mathrel{+}= t;\ \bar{s}_{i-1} \end{array}$$

in the reverse section, where $\bar{s}_k$ are the adjoint statements corresponding to s_k.

Proof. The forward PGAS codes are given as

$$\begin{array}{l} s_{i-1};\ s_{i+1};\ \dots\ s_{j-1};\ s^2_{j+1};\ b^2 = a^1;\ s^1_{j+1};\ \dots\ s_{k-1};\ s_{k+1} \\ s_{i-1};\ s_{i+1};\ \dots\ s_{j-1};\ s_{j+1};\ b^2 = a^1;\ \dots\ s_{k-1};\ s_{k+1} \\ \dots \\ s_{i-1};\ s_{i+1};\ \dots\ s_{j-1};\ s_{j+1};\ \dots\ s_{k-1};\ b^2 = a^1;\ s_{k+1} \end{array}$$

The reverse sections of the adjoint PGAS codes become

$$\begin{array}{l} \bar{s}_{k+1};\ \bar{s}_{k-1};\ \dots\ \bar{s}^1_{j+1};\ \bar{a}^1 \mathrel{+}= \bar{b}^2;\ \bar{b}^2 = 0;\ \bar{s}^2_{j+1};\ \bar{s}_{j-1};\ \dots\ \bar{s}_{i+1};\ \bar{s}_{i-1} \\ \bar{s}_{k+1};\ \bar{s}_{k-1};\ \dots\ \bar{a}^1 \mathrel{+}= \bar{b}^2;\ \bar{b}^2 = 0;\ \bar{s}_{j+1};\ \bar{s}_{j-1};\ \dots\ \bar{s}_{i+1};\ \bar{s}_{i-1} \\ \dots \\ \bar{s}_{k+1};\ \bar{a}^1 \mathrel{+}= \bar{b}^2;\ \bar{b}^2 = 0;\ \bar{s}_{k-1};\ \dots\ \bar{s}_{j+1};\ \bar{s}_{j-1};\ \dots\ \bar{s}_{i+1};\ \bar{s}_{i-1} \end{array}$$

The variable a^1 is not written by any of the statements in s_{i+1}; ... s_{k-1} because the original message-passing program is assumed to satisfy the restrictions on *isend*. Similarly, b^2 is neither read nor written by $s_{j+1}; \ ... \ s_{k-1}$. However, the value of a^1 may be read by statements in $s_{i+1}; \ ... \ s_{k-1}$, implying that while $\bar{a}^1$ may be incremented by $\bar{s}_{k-1}; \ ... \ \bar{s}_{i+1}$, it is not read or written otherwise. The order of two successive increment operations can be switched if the incremented variable is neither read nor written in between the two increment operations.[8] Moreover, the placement of theses increment operations is arbitrary as long as the values of the increments do not change. The value of $\bar{b}^2$ is neither read nor written by $\bar{s}_{k-1}; \ ... \ \bar{s}_{j+1}$. Hence, the statement $\bar{a}^1$+=$\bar{b}^2$ can be inserted at any position between $\bar{s}_{k+1}$ and $\bar{s}_{j-1}$. In other words, the adjoints of all PGAS versions of the given message-passing program are equivalent.

The adjoint message passing program yields the following set of constraints for the placement of the adjoint x-assignment $\bar{\chi} \equiv$ "$t = \bar{b}^2$" :

$$\bar{s}^1_{k+1} < \bar{\chi}; \quad \bar{s}^2_{k+1} < \bar{\chi}; \quad \bar{s}^2_{j-1} > \bar{\chi}; \quad \bar{s}^1_{i-1} > \bar{\chi} \quad .$$

Hence, the PGAS versions of the adjoint message-passing program are the following:

$$\bar{s}_{k+1}; \ \bar{s}_{k-1}; \ \dots \ \bar{s}_{j+1}; \ t = \bar{b}^2; \ \bar{b}^2 = 0; \ \bar{s}_{j-1}; \ \dots \ \bar{s}_{i+1}; \ \bar{a}^1{+}{=}t; \ \bar{s}_{i-1}$$
$$\dots$$
$$\bar{s}_{k+1}; \ t = \bar{b}^2; \ \bar{s}_{k-1}; \ \dots \ \bar{s}_{j+1}; \ \bar{b}^2 = 0; \ \bar{s}_{j-1}; \ \dots \ \bar{s}_{i+1}; \ \bar{a}^1{+}{=}t; \ \bar{s}_{i-1}$$

As a compiler-generated auxiliary variable, t can be guaranteed not to be read or written by any of the statements $\bar{s}_{k-1}; \ ... \ \bar{s}_{i+1}$. From our previous argument we recall that $\bar{a}^1$ may be incremented by $\bar{s}_{k-1}; \ ... \ \bar{s}_{i+1}$ but it is not read or written otherwise. Hence, the increment operation of $\bar{a}^1$ with t can be placed in between $\bar{s}_{i+1}$ and $\bar{s}_{i-1}$. As the value of $\bar{b}^2$ is neither read nor written by $\bar{s}_{k-1}; \ ... \ \bar{s}_{j+1}$, the fixed placement of $\bar{b}^2 = 0$ in between $\bar{s}_{j+1}$ and $\bar{s}_{j-1}$ does not change the program's semantics either. The auxiliary variable t can be removed as the result of copy-propagation [1], yielding a possible replacement of the first assignment in $t = \bar{b}^2; \ \dots \ \bar{a}^1{+}{=}t$ with $\bar{a}^1{+}{=}\bar{b}^2$. Consequently, the adjoint PGAS versions of the message-passing program are semantically equivalent to the PGAS versions of the adjoint message-passing program. ■

3 Conclusion and Outlook

A formalism for proving the correctness of adjoint message-passing programs has been illustrated by means of an asynchronous unbuffered send/receive communication between two processes. This method is applied to a large number of transformation rules currently being implemented in OpenAD [21] and the differentiation-enabled NAGWare Fortran compiler [18]. It is based on analyzing the data dependences in the PGAS versions of the original message-passing program. Rigorous proofs can thus be constructed that rely only on program analysis techniques used in classical compiler construction. We intend to consider ideas presented in [19] in order to investigate a potential automatization of this proof technique.

[8] For a given use of a variable we distinguish between reads, writes, and increment operations as a special case of a read-write combination.

One of our long-term goals is to build an adjoint message-passing library on top of MPI. Such an extension is desirable for achieving satisfactory efficiency. The ability to prove the correctness of given communication patterns is a fundamental ingredient of this ambitious research and development project.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.
2. M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors. *Computational Differentiation: Techniques, Applications, and Tools*, Proceedings Series. SIAM, 1996.
3. C. Bischof, M. Bücker, P. Hovland, U. Naumann, and J. Utke, editors. *Advances in Automatic Differentiation*, LNCSE, Berlin, 2008. Springer. To appear.
4. M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors. *Automatic Differentiation: Applications, Theory, and Tools*, number 50 in LNCSE, Berlin, 2005. Springer.
5. A. Carle and M. Fagan. Automatically differentiating MPI-1 datatypes: The complete story. In *[7]*, chapter 25, pages 215–222. Springer, 2002.
6. C.Faure and P.Dutto. Extension of Odyssée to the MPI library – reverse mode. Rapport de recherche 3774, INRIA, Sophia Antipolis, Oct. 1999.
7. G. Corliss, C. Faure, A. Griewank, L. Hascoët, and U. Naumann, editors. *Automatic Differentiation of Algorithms – From Simulation to Optimization*, New York, 2002. Springer.
8. G. Corliss and A. Griewank, editors. *Automatic Differentiation: Theory, Implementation, and Application*, Proceedings Series. SIAM, 1991.
9. T. El-Ghazawi. Partitioned Global Address Space (PGAS) programming languages. Tutorial at SC07. See `http://sc07.supercomputing.org/`.
10. C. Faure, P. Dutto, and S. Fidanova. Odysée and parallelism: Extension and validation. In *Procceedings of the 3rd European Conference on Numerical Mathematics and Advanced Applications, Jyväskylä, Finland, July 26-30, 1999*, pages 478–485. World Scientific, 2000.
11. A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, 2000.
12. L. Hascoët, U. Naumann, and V. Pascual. To-be-recorded analysis in reverse mode automatic differentiation. *Future Generation Computer Systems*, 21:1401–1417, 2005.
13. P. Heimbach, C. Hill, and R. Giering. Automatic generation of efficient adjoint code for a parallel Navier-Stokes solver. In P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, editors, *Proceedings of ICCS 2002*, volume 2330 of *LNCS*, pages 1019–1028, Berlin, 2002. Springer.
14. 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.
15. J. Marotzke, R. Giering, K. Zhang, D. Stammer, C. Hill, and T. Lee. Construction of the adjoint MIT ocean general circulation model and application to Atlantic heat transport variability. *J. Geophysical Research*, 104, C12:29,529–29,547, 1999.
16. U. Naumann. Call tree reversal is NP-complete. In *[3]*. 2008. To appear.
17. U. Naumann. DAG reversal is NP-complete. *J. Discr. Alg.*, 2008. To appear.
18. U. Naumann and J. Riehme. A differentiation-enabled Fortran 95 compiler. *ACM Transactions on Mathematical Software*, 31(4):458–474, 2005.
19. D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.
20. M. Mills Strout, B. Kreaseck, and P. Hovland. Data-flow analysis for MPI programs. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 175–184, Washington, DC, USA, 2006. IEEE Computer Society.
21. J. Utke, U. Naumann, C. Wunsch, C. Hill, P. Heimbach, M. Fagan, N. Tallent, and M. Strout. OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software*, 34(4), 2008.