# A Duality Between Forward and Adjoint MPI Communication Routines

Benny N. Cheng
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA, U.S.A.

**Abstract** ***In this article,we explore a natural duality that exist between MPI communication routines in parallel programs, and show the ease of its adjoint implementation via pointers.***

*Keywords:* ocean modeling,MPI, parallel programming,automatic differentiation,adjoint

## 1 Introduction

Parallel ocean models are widely used in the oceanographic community as a research tool. Forecasting and estimating the state of the ocean at a particular time is a main staple of ocean modeling. One of the recent major developments in this subject area is the use of the so-called adjoint inverse model, as opposed to the forward simulation that is the usual way to run a model. The adjoint model, which can be derived either manually by hand coding or through automatic differentiation tools such as TAF [1] , is a kind of reverse simulation that has proven useful for tracing water flows backwards in time[2] , or for measuring sensitivity of specific oceanic variables such as temperature and salinity [3]. Automatic differentiation tools nowadays have few problems adjointing sequential forward codes, however, they still have some ways to go toward adjointing parallel communication routines inside a parallel forward model [4]. We demonstrate that with the right coding structure, parallel communication routines can be easily adjointed with ease.

## 2 MPI calls setup

The ocean model that we selected for adjointing is the so-called Modular Ocean Model version 4 (MOM4) [5], originally developed at the Geophysical Fluid Dynamics Laboratory (GFDL) in Princeton university. This model is written in the modern F90 language, with built-in parallel communication routines , written primarily by V. Balaji. The following code snippet illustrates the message exchanges occurring between different CPUs:

```
do m=1,ncpus
if( send_w .AND. domain%list(m)%send_w%overlap )then
is1 = domain%list(m)%send_w%is1; ie1 = domain%list(m)%send_w%ie1
js1 = domain%list(m)%send_w%js1; je1 = domain%list(m)%send_w%je1
...
buffer(pos) = field(is1:ie1,js1:je1)
...
endif
call mpp_send( buffer(pos), plen=msgsize, to_pe=to_pe )
if( recv_e .AND. domain%list(m)%recv_e%overlap )then
is2 = domain%list(m)%recv_e%is2; ie2 = domain%list(m)%recv_e%ie2
js2 = domain%list(m)%recv_e%js2; je2 = domain%list(m)%recv_e%je2
endif
call mpp_recv( buffer(pos), glen=msgsize, from_pe=from_pe )
field(is2:ie2,js2:je2) = buffer(pos)
enddo
```

In the above code segment, MPP_SEND and MPP_RECV are simply wrapper routines for standard MPI calls MPI_SEND and MPI_RECV, and we consider them as equivalent for the purpose of this article. The above code signal each cpu to send the appropriate slab of the array 'field' to its west neighbor, which then placed this slab into its proper destination. Similar types of communication exist in practically all parallel programs. The key point to note in the code is the efficient characterization of the relationships between domain, cpu, communication task to be performed, and array indices, in their respective order as a chain of pointers. Once this chain of relationships is established, adjointing the code is relatively simple.

```
do m=1,ncpus
if( recv_e .AND. domain%list(m)%recv_e%overlap )then
is1 = domain%list(m)%recv_e%is1; ie1 = domain%list(m)%recv_e%ie1
js1 = domain%list(m)%recv_e%js1; je1 = domain%list(m)%recv_e%je1
...
buffer(pos) = field(is1:ie1,js1:je1)
field(is1:ie1,js1:je1) = 0.
...
endif
call mpp_send( buffer(pos), plen=msgsize, to_pe=to_pe )
if( send_w .AND. domain%list(m)%send_w%overlap )then
is2 = domain%list(m)%send_w%is2; ie2 = domain%list(m)%send_w%ie2
js2 = domain%list(m)%send_w%js2; je2 = domain%list(m)%send_w%je2
endif
call mpp_recv( buffer(pos), glen=msgsize, from_pe=from_pe )
field(is2:ie2,js2:je2) = field(is2:ie2,js2:je2) + buffer(pos)
enddo
```

Now the duality between MPP_SEND and MPP_RECV calls is now apparent. In the adjoint code, we send where we recieved before , and received where we send previously.

# 3 Conclusions

Adjointing a complicated piece of code is normally a tedious task, especially if done manually. Recent developments in automatic differentiation tools have aided greatly in reducing the coding time. However, as of the moment, there are still some major shortcomings with regards to the adjointing of parallel codes. We have shown that with the proper coding structure and setup, such a task can be done with ease and may be incorporated into future versions of automatic code transformation programs.

# Acknowledgement

This work is performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Agency. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology. We are also grateful to NASA AMES Research Center for the use of their SGI Altix Columbia supercomputers in running the ocean model code.

# References

[1] Transformation of Algorithms in Fortran
http://www.fastopt.com/topics/products.html

[2] Fukumori I., T. Lee, B. Cheng, D. Menemenlis, *The origin, pathway, and destination of Nino3 water estimated by a simulated passive tracer and its adjoint*, 2004, J. Phys. Oceanogr., 34, 582-604

[3] Errico, R., *What is an Adjoint Model?*, 1997, Bull. of Amer. Met. Soc., Vol. 78, 11, pp. 2577-2591.

[4] Heimbach P., C. Hill and R. Giering (2002). *Automatic generation of efficient adjoint code for a parallel Navier-Stokes solver*, Computational Science-ICCS 2002, PT II, Proceedings, Vol. 2330, pp.1019-1028.

[5] http://www.gfdl.noaa.gov/fms/