Installing and using TAF on Stampede2

Patrick Heimbach

May 29, 2019

Contents

1 Installing TAF

TAF stands for Transformation of Algorithms in Fortran. It was created and is distributed by the company FastOpt (http://fastopt.de).

For the purpose of the ECCO summer school 2019, FastOpt has kindly agreed to provide free access to TAF for all participants.

FastOpt asks that no code generated during the summer school shall be used beyond the end of the school.

1.1 Preliminaries

All resources (files and fields) are shared in the ECCO community space on Stampede2. We assign a variable name to the directory trunk in which they can be found:

```
communitydir = /work/projects/aci/ECCO/community/
```

It is useful to make either your $HOME directory or a dedicated sub-dir. (e.g., `$HOME/bin/`) part of your search path in your `.bashrc`. In the following we will assume that you have no `$HOME/bin/`.

1.2 The script `staf`

TAF on Stampede2 is enabled via a script, `staf`. The script gathers the code to be transformed, sends it to a server at fastopt.com where the source-to-source transformation is performed, and returns the transformed code back to your local directory from which you have invoked it.

You can obtain the `staf` script by copying it from the community space:

```
# copy the staf script to your home dir.
cp $communitydir/bin/staf $HOME/.
```

The script uses the same arguments as TAF itself. To familiarize yourself with possible arguments, type

```
staf -help
```

You can use `staf` to download a TAF manual via command

```
staf -get manual
```

A very good paper stepping through the essentials of source-to-source code AD is that by Giering and Kaminski, TOMS, 1998, http://doi.org/10.1145/293686.293695.

1.3 Getting access

Use of TAF requires a license. A valid license is enabled via setting up a secure connection to the Fastopt server. To do so, Fastopt issues a public key that is placed in your folder `$HOME/.ssh/`. For the duration of the ECCO Summer School, Fastopt has issued a single key that will be shared by all students. The key will expire after the end of the school.

To set up and enable `staf` on Stampede2, follow these steps:

```
# copy the TAF public keys to your $HOME/.ssh/ dir.
cp $communitydir/bin/taf_keys/* $HOME/.ssh/.
```

Assuming that your `$HOME` is in your search path, you are now ready to use TAF via the `staf script`. To test this, do the following:

```
staf -test
```

The result should be a message

```
 Transformation of Algorithms in Fortran (TAF)
 Copyright 2000-2019 FastOpt GmbH, Hamburg, Germany
 All rights reserved.
 URL: http://www.FastOpt.de, Email: info@FastOpt.de
 script to access TAF remotely version 4.0

Your access to TAF is enabled
```

2 Application: Stommel's 3-box model

This application uses Stommel's 3-box model, consisting of a surface equator-to-subpolar cell, a deep equator-to-subpolar cell, and a polar (top-to-bottom) cell.

2.1 Obtaining the code

We have provided a Fortran code of the 3-box model in the community space. To obtain it, do the following on your terminal.

```
# go to your work directory
cd $WORK/../
# anticipating more adjoint examples, we create a base directory
mkdir adjoint_examples/
cd adjoint_examples/
# now get a local copy of the box model code
cp -r $communitydir/adjoint_examples/boxmodel_stommel_2019 .
```

2.2 Derivative code generation

The two basic modes of algorithmic differentiation (AD) for generating first derivatives of the model's dependent variable (output, cost function, quantity of interest) to the model's independent variables (inputs, control variables, initial/surface boundary conditions, internal model parameters) are

- *forward mode AD*: the tangent linear model
- *reverse mode AD*: the adjoint model

We will cover both in the following.

2.2.1 Adjoint (or reverse) mode of AD

We now invoke TAF to generate the adjoint code based on the original model

```
# we chdir to the directory where we will compile the model
cd $HOME/../adjoint_examples/boxmodel_stommel_2019/box_model_030118_scalar_cost/bin/
# make dependencies, in particular linking all source code to the compile dir.
make depend
# we separate the compile directory from the directory where we generate the adjoint
# this is purely for convenience and clarity
# so we need to chdir
cd ../adjoint/
# let's clean, i.e. remove earlier generated codes
make clean
# NOW ... invoke TAF (via staf)
make adtaf
```

You now see a number of new files in your directory:

- `taf_ad_out.log`:
  detailed output by TAF, documenting what it has done
- `taf_command`:
  the `staf` command that was issued, in particular containing the list of arguments provided to `staf`
- `taf_output`:
  short output; would contain ERROR messages, if there are any
- `tamc_codead.f`:
  Contains *both* the adjoint code generated by TAF as well as the original code

- `tamc_code.f`:
  All forward code assembled into a single file (not required, just for convenience)

Finally, we go ahead and compile the code:

```
# we chdir back to the directory where we will compile the model
cd $HOME/../adjoint_examples/boxmodel_stommel_2019/box_model_030118_scalar_cost/bin/
# now we make the executable
make
```

2.2.2 Tangent linear (or forward) mode of AD

We now invoke TAF to generate the adjoint code based on the original model

2.3 Running the model

We are now ready to run the model. The Makefile was set up such that the executable (file `modelexe`) obtained is placed into a dedicated directory
`$HOME/../adjoint_examples/boxmodel_stommel_2019/box_model_030118_scalar_cost/exe/`
Let's go there and run it:

```
# chdir to model execution directory
cd $HOME/../adjoint_examples/boxmodel_stommel_2019/box_model_030118_scalar_cost/exe/
# run the model
./runexe
```

2.4 interpreting the output

...

3 Some comments on the AD-generated code

3.1 An example code line and its adjoint

In the lecture, we covered the advection equation for T_3:

$$\begin{aligned} \frac{dT_3}{dt} &= U(T_3 - T_2), \quad \text{for } U \geq 0 \\ \texttt{diffT3} &= \texttt{u} * (\texttt{T3} - \texttt{T2}) \end{aligned}$$

Its total derivative is:

$$\delta\texttt{diffT3} = \frac{\partial\texttt{diffT3}}{\partial\texttt{U}}\delta\texttt{U} + \frac{\partial\texttt{diffT3}}{\partial\texttt{T}_2}\delta\texttt{T}_2 + \frac{\partial\texttt{diffT3}}{\partial\texttt{T}_3}\delta\texttt{T}_3$$

In the box model code provided, the corresponding line is in file `box_timestep.F`. (Unfortunately, some variable names have changed, but it should be straightforward to attribute the code.)

```
         dFldDt(3) = velsign*
     &        uVelLoc*( fldNow(2) - fldNow(3) ) / vol(3)
```

The corresponding adjoint code is contained in file `box_adjoint.F`, S/R `box_timestepad`:

```
        fldnowad(3) = fldnowad(3)-dflddtad(3)*(velsign*uvelloc/vol(3))
        fldnowad(2) = fldnowad(2)+dflddtad(3)*(velsign*uvelloc/vol(3))
        uvellocad = uvellocad+dflddtad(3)*(velsign*(fldnow(2)-fldnow(3))
     $/vol(3))
        dflddtad(3) = 0.d0
```

Compare this to the matrix expression we derived in the lecture.

3.2 IF-statements

The S/R `box_timestep.F` also contains an example of an IF-statement:

```
      if ( uVelLoc .GE. 0. ) then
        ...
      else
        ...

      end if
```

The corresponding adjoint code in S/R `box_timestepad` documents how AD generates conditional derivative code.

3.3 Reversal of S/R call sequence

The "top-level routine" of the box model is S/R `box_model_body`. The bare-bones calling sequence within the time stepping loop is:

```
c-- calculate densities:
            call box_density( tNow, sNow, rho )
```

```
c-- calculate transport:
            call box_transport( rho, uVel )

CADJ STORE uvel = comlev1, key = ikey, byte = isbyte

c-- leap frog time stepping:
            call box_timestep(
     &           'T', gamma_t, tStar, nullforce,
     &           uVel, tNow, tOld, tNew )

            call box_timestep (
     &           'S', gamma_s, sStar,        FW,
     &           uVel, sNow, sOld, sNew )

c-- Robert filter:
            call box_robert_filter( tNow, tOld, tNew )
            call box_robert_filter( sNow, sOld, sNew )

c-- cycle fields
            call box_cycle_fields
```

The corresponding call sequence of the adjoint code is in S/R `box_model_bodyad`:

```
              call box_cycle_fieldsad
              call box_robert_filterad( snowad,soldad,snewad )
              call box_robert_filterad( tnowad,toldad,tnewad )
              call box_timestepad( gamma_s,uvel,uvelad,snow,snowad,
     $soldad,snewad )
              call box_timestepad( gamma_t,uvel,uvelad,tnow,tnowad,
     $toldad,tnewad )
              call box_transportad( rhoad,uvelad )
              call box_densityad( tnowad,snowad,rhoad )
```

3.4 Store directive and multi-level checkpointing

TBD ...