WW3 Tutorial 4.1: compile and run with MPI

Purpose

In this tutorial exercise we will go through the steps of compiling WAVEWATCH III® for both single- and multi-processor (MPI) compute environments. It expands on the day 1 exercise where the code is compiled for single-processor applications only. We will then go through some interactive model runs to see how the code works in the different environments.

Tutorial material

Part of this exercise will be run from the standard work directory of WW3. If the code is installed in the directory `~/wwatch3`, then this work directory is `~/wwatch3/work`. In this directory, links to the `comp` and `link` scripts are already available. Also used are the `w3_make` and `make_MPI` scripts from the wwatch3/bin directory. Links to these scripts in the work directory can easily be made by executing

```
cd ~/wwatch3/work
ln3 w3_make
ln3 make_MPI
```

Additional test are using the example input files in this directory. For convenience, copies of these files are provided in the directory `day_4/tutorial_MPI`. These files represent standard input files, as well as a script to clean up the directory, and `comp` and `link` scipts for MPI on the machine available for the winter school.

```
clean.sh
switch
ww3_grid.inp
ww3_prep.inp
ww3_strt.inp
ww3_shel.inp
ww3_outf.inp
ww3_outp.inp
comp.UMD
link.UMD
```

Basic compilation

The basic compilation of WW3 has already been addressed in the tutorials of day 1. Whenever a version of WW3 is installed, the `comp` and `link` scripts need to be adapted to the actual compiler, and the `switch` file identifies compile-time model options, including switches for shared and distributed memory options. As a preparation for setting up the model for distributed computations using MPI, we will first compile and test all codes in a shared-memory environment. In the switch file, the shared memory environment is identified using the `SHRD` switch (see switch file contents on the following lines).

```
F90 NOGRB LRB4 SHRD NOPA PR3 FLX2 LN1 ST2 STAB2 NL1 BT1 DB1 MLIM
TR0 BS0 XX0 WNX1 WNT1 CRX1 CRT1 O0 O1 O2 O3 O4 O5 O6 O7 O11 O14
```

After copying the file to the proper location expected by the compile scripts, the standard compile command

```
cp switch ~/wwatch3/bin/.
w3_make
```

will compile all WW3 executables for shared-memory execution. After all codes have been compiled correctly (output not reproduced here again), go to the `day_4/tutorial_MPI` directory to test the codes and to generate some benchmark results to test the MPI implementation against. Program outputs are not reproduced here, and the output post-processors are run twice, once to see the output on the screen, and once to generate a baseline results file that can be used for later comparison with the results from the MPI model run. Similarly, the log file from `ww3_shel` is saved.

```
cd ~/day_4/tutorial_MPI
ww3_grid
ww3_prep
ww3_strt
ww3_shel
cp log.ww3 log.tst
ww3_outp
ww3_outp > ww3_outp.tst
ww3_outf
ww3_outf > ww3_outf.tst
```

MPI basics

MPI is a standard way to communicate between processors in a distributed memory compute environment. Whereas MPI as a `language' is highly standardized, MPI implementations on various computers are not. Compiling with MPI and executing MPI codes is, therefore, unfortunately rather system dependent. Here we will cover the three basic steps needed to compile and run using MPI:

1) Let WW3 know that you want to use MPI.
2) Compile with the proper MPI libraries.
3) Execute WW3 in the proper parallel compute environment.

The first step is simple and independent of the compute environment. In the `switch` file, the switch `SHRD` for "shared" needs to be replaced by two switched, `DIST` for "distributed", and `MPI` for using standard MPI coded. The default switch file thus needs to be modified as follows (in the `~/wwatch3/bin` or `~/wwatch3/work` directory):

```
F90 NOGRB LRB4 DIST MPI NOPA PR3 FLX2 LN1 ST2 STAB2 NL1 BT1 DB1
MLIM TR0 BS0 XX0 WNX1 WNT1 CRX1 CRT1 O0 O1 O2 O3 O4 O5 O6 O7 O11
O14
```

The second step is depending on the way in which MPI is implemented on your computer, and generally requires modifications to the `comp` and `link` scripts. In these scripts, and environment variable $mpi_mod is set and used to point to the proper compiler wrappers, is such wrappers are available. An example of a small piece of code from the comp script using Portland and a MPI compile wrapper looks like this (this is the way it works at the UMD cluster)

```
if [ "$mpi_mod" = 'yes' ]
then
  comp=mpif90
else
  comp=pgf90
fi
```

If such a wrapper is not available, you will generally have to link the proper libraries "manually". An example is given here for the Lahey compiler on NOAA's Zeus R&D SGI cluster. Note that `$opt` represents the compiler options used in the `comp` script.

```
if [ "$mpi_mod" = 'yes' ]
then
  comp=lf95
  opt="$opt -L${MPI_ROOT}/lib -lmpi -I${MPI_ROOT}/include"
else
  comp=lf95
fi
```

Fortunately once you have figured this out, you will not have to update it, unless compilers and libraries are updated. The final step is to execute these codes in the proper parallel compute environment. This typically involves compute resource management, and is often done through batch processing. Many Linux implementations (like on UMD cluster) use a command like (sometimes needing full that on executable)

```
mpirun -np 12 ww3_shel
```

to execute the `ww3_shel` program under MPI, in this case using 12 processors. There are, however, many different syntaxes for many different implementations, so you will have to learn the proper way of doing this on your own hardware.

There is no way that we can cover all possibilities here. Therefore, we have provided a `comp` and `link` script that work on the test system available for the workshop. We will work from these focusing on functionality and basic principles, but not on details of MPI implementations. Note that these script were already used on day 1.

MPI compilation (setup)

Above, we have already compiled all code for shared memory (single processor) execution. To prepare for compilation using MPI, we need to modify the `switch`, `comp` and `link` scripts in the WW3 directories. We will start with replacing the `comp` and `link` scripts with those provided with the tutorial.

```
cd ~/wwatch3/bin
cp ~/day_4/tutorial_MPI/comp.UMD comp
cp ~/day_4/tutorial_MPI/link.UMD link
cd ../work
```

This sets up the compiler correctly.

MPI compilation (manual)

In the `switch` file in the ~/`wwatch3/work` directory, we now need to replace the `SHRD` switch with the `DIST` and `MPI` switches, as shown above. With this, `w3_make` will automatically recompile all subroutines used, as they may contain MPI code or includes. Here it is sufficient to cat the compile command for a single code (`ww3_shel`) only. Other parallel codes cloud be `ww3_multi`, `ww3_strt` and `ww3_sbs1`.

```
w3_make ww3_shel
```

Part of the output of this command is duplicated below. Note the highlighted script output.

```
                  ****************************
                ***  compiling WAVEWATCH III  ***
                  ****************************

Scratch directory : . . . . . .
Save source codes : yes
Save listings     : yes

Making makefile ...
   new shared / distributed memory
   new message passing protocol
   Checking all subroutines for modules (this may take a while) ...

Processing ww3_shel
--------------------
ad3 : processing w3servmd
ad3 : processing w3gsrumd

........

ad3 : processing w3fldsmd
ad3 : processing ww3_shel
      Linking ww3_shel


                    **********************
                  *** end of compilation ***
                    **********************
```

MPI compilation (automated)

Because going back and forth between shared and distributed codes happens a lot during model development, the MPI compilation has been automated in a single script

```
make_MPI
```

This script manipulates the `switch` file, and uses `w3_new` and `w3_make` to get all codes compiled as needed. Note that minor editing at the end of the script defied the status of the `switch` file at the end of running this script. This is fairly short script. Edit it to take a look inside. Note the file lists, which we regularly edit as needed for the development work we are doing. The output of this script is mainly output from `w3_make`, and is not reproduced here.

Testing and MPI model setup

After having run `make_MPI` above, most codes are compiled for traditional execution, whereas `ww3_shel` and `ww3_multi` are set up for MPI execution. We can now redo previous interactive computations. First we will go to the proper directory, and clean up all files except for the benchmark (`.tst`) files.

```
cd day_4/tutorial_MPI
clean.sh
```

Now we will run the same sequence of codes are run above to generate the benchmark data, with the exception of running `ww3_shel` under MPI. As mentioned above, the syntax of executing the latter program is high system dependent. The `diff` command should show that there are no differences in results between running the code on 1 or more processors.

```
ww3_grid
ww3_prep
ww3_strt
mpirun -np 12 ww3_shel
ww3_outf
ww3_outf > ww3_outf.out
diff ww3_outf.tst ww3_outf.out
ww3_outp
ww3_outp > ww3_outp.out
diff ww3_outp.tst ww3_outp.out
```

It is also interesting to compere the `log.ww3` and `log.tst` files. The files should be identical with the exception of the elapsed time at the bottom. As this is a very small problem, do not be surprised if running under MPI does not give much benefit here. After playing with this , the tutorial directory can be cleaned up by executing

```
clean.sh all
```

Pitfalls and common errors.

Running an MPI code without a parallel environment:
When attempting to run a code compiled for MPI without using a parallel environment usually leads to a terminal error in MPI identifying that the parallel environment is not properly set. The error message may have many different forms, often mentioning that node resource data is not available.

Running an single processor code in a parallel (MPI) environment:
This will not always give an error or program abort. The code will just run as slow as a single processor code, or even much slower due to IO conflicts. The way in which this situation is easily identified, is that it represents a case where the entire code is run on each processor individually, and hence produces output for each processor individually. If this is done on 12 processors, each output line will be reproduced 12 times, in a somewhat scrambled format. On some of the NOAA machines, this will not execute at all, but give a convoluted error message.

Compile looks OK, but code does not run / freezes:
If this is a developmental code from the svn server, there may well be an error in the code. If this is a well-tested distributed version of the code, it is also likely that there is a problem with the MPI implementation on your machine. Case and point, several years ago IBM managed to break our operational wave models with a "transparent" upgrade to MPI. Last moth, we needed a special MPI environment set up to get our well established MPI codes to run on our new operational machines. If something like this happens, contact both your IT support and us. It is good to try and debug, but it may well be outside your (and our) capability to repair . . .

References

No references for this tutorial, other than the manual.

More information:
Hendrik Tolman (Hendrik.Tolman@NOAA.gov)