Useful Script for Compilation

Hans Petter Langtangen*

March 14, 2004

1 Using the C Preprocessor in Fortran Codes

C and C++ compilers run a preprocessor[1] prior to the compilation. The preprocessor is a handy tool that is, unfortunately, not integrated with Fortran compilers. Nevertheless, the preprocessor can be executed as a stand-alone program, called `cpp`, or it can be run as a part of a C compiler (e.g., `gcc -E`). In this way, one can apply the preprocessor to any file, including Fortran source code files. We shall now develop a script that transforms a Fortran file with C preprocessor directives to standard Fortran syntax. This will allow writing Fortran programs with, e.g., include statements (`#include`), macros (`#define`), and C-style (possible multi-line) comments (`/* ... */`).

We let Fortran files containing C preprocessor directives have the extension `.fcp`. The script is to be invoked with the following command-line parameters:

```
[cpp options] file1.fcp
```

That is, standard `cpp` options can be present, followed by the name of a single Fortran file. Typical examples on `cpp` options are definitions of macros, like `-DMY_DEBUG=2`, and specification of directories with include files, like `-I../mydir`. The C preprocessor is run by the command

```
cpp [cpp options] file1.fcp > file1.f
```

if you have `cpp` available as a separate program. Since `cpp` is not always present as a separate program, we recommend to run the preprocessor as part of GNU's C compiler `gcc`, since `gcc` is a standard utility found on most machines. The relevant commands are then

```
cp file1.fcp tmp.c # gcc must work with a file with suffix .c
gcc -E -c [cpp options] tmp.c > file1.f
rm -f tmp.c
```

In Python this becomes

```
cpp_options = ' '.join(sys.argv[1:-1])
shutil.copy(fcp_file, 'tmp.c')
cmd = 'gcc -E %s -c tmp.c > %s.f' % (cpp_options,fcp_file[:-4])
os.system(cmd)
os.remove('tmp.c')
```

A fundamental problem with the macro expansions performed by the preprocessor is that code lines can easily exceed 72 characters, which is illegal according to the Fortran 77 standard. Although modern Fortran 77 compilers, and in particular Fortran 90/95 compilers, allow longer line lengths, buffer overflow is not unusual for long lines (longer than (say) 255 characters). Since macros are expanded to a single line, there is a danger of very long lines, and the script needs to split lines that are longer than a specified number of characters, which we here set to 72. Fortunately, whitespace is not significant in Fortran so one can split a line by just inserting newline, five blanks, and any character (indicating continuation of a line) in column six.

*Dept. of Scientific Computing, Simula Research Laboratory, and Dept. of Informatics, University of Oslo. hpl@simula.no.

[1]This section probably does not make much sense if you are not familiar with the C or C++ preprocessor.

```
# split lines that are longer than maxlen chars:
maxlen = 72
f = open(fcp_file[:-4]+'.f', 'r'); lines = f.readlines(); f.close()
for i in range(len(lines)):
    line = lines[i]
    if len(line) > maxlen:  # split line?
        nrest = len(line) - maxlen
        splitline = line[0:maxlen]
        start = maxlen
        while nrest > 0:
            splitline = splitline + '\n     &' + \
                        line[start:start+maxlen-6]
            start = start + maxlen-6
            nrest = nrest - (maxlen-6)
        lines[i] = splitline  # in-place list modification
```

Note that this split feature makes the script convenient for writing Fortran source code files without any restriction on the line length, besides allowing the use of any C preprocessor directive.

Newer C preprocessors preserve indentation, but minimize whitespace elsewhere such that labels like `10 CONTINUE` appear as `10 CONTINUE`, which is not valid Fortran 77 since `CONTINUE` starts before column 7. We therefore need to ensure that labels in columns 1-6 appear correctly:

```
c = re.compile(r'^(\s*)(\d+)(\s*)')  # label regex
for i in range(len(lines)):
    # remove lines starting with #
    lines[i] = re.sub(r'^#.*', '      ', lines[i])
    if len(lines[i]) >= 5:
        column1to5 = lines[i][0:5]
        if re.search(r'\w',column1to5):
            # letter after label?
            m = re.search(r'(\s*)(\d+)(\s+)\w+', column1to5)
            if m:
                # insert extra white space after group 3
                n = len(m.group(1))+len(m.group(2))+len(m.group(3))
                space = ''.join([' ']*(6 - n))
                lines[i] = m.group(1) + m.group(2) + m.group(3) + \
                           space + lines[i][n:]
```

Writing `lines` back to the Fortran file finishes the script. The complete script is called `fccp.py` and is available in `src/tools`.

Here is a simple test example that `fccp.py` can handle. Two macros are defined in a file `macros.i`, stored in (say) `/home/hpl/f77macros`,

```
#define DDx(u, i, j, dx) \
  (u(i+1,j) - 2*u(i,j) + u(i-1,j))/(dx*dx)
#define DDy(u, i, j, dy) \
  (u(i,j+1) - 2*u(i,j) + u(i,j-1))/(dy*dy)
```

A Fortran 77 file `wave1.fcp` with C macros, `#ifdef` directives, and C-style comments has the following form:

```
#include <macros.i>

C234567 column numbers 1-7 are important in F77!
      SUBROUTINE WAVE1(SOL, SOL_PREV, SOL_PREV2, NX, NY,
     &                 DX, DY, DT)
C     variable declarations:
      INTEGER NX, NY     /* no of points in x and y dir */
      REAL*8 DX, DY, DT  /* cell and time increments */
      REAL*8 SOL(NX,NY), SOL_PREV(NX,NY), SOL_PREV2(NX,NY)

C     update SOL:
      DO 20 J=1, NY
        DO 10 I=1, NX
        /*
          a 2nd-order time difference combined with
          2nd-order differences in space results in
          the standard explicit finite difference scheme
          for the wave equation:
        */
          SOL(I,J) = 2*SOL_PREV(I,J) - SOL_PREV2(I,J) +
     &               DT*DT*(DDx(SOL_PREV, I, J, DX) +
```

```
     &                       DDy(SOL_PREV, I, J, DY))
#ifdef DEBUG
          WRITE(*,*) 'SOL(',I,',',J,')=',SOL(I,J)
#endif
 10     CONTINUE
 20   CONTINUE

      RETURN
      END
```

We may then run

```
fccp.py -I/home/hpl/f77macros wave1.fcp
```

and get a valid Fortran 77 file `wave1.f`, which looks like this:

```
C234567 column numbers 1-7 are important in F77!
      SUBROUTINE WAVE1(SOL, SOL_PREV, SOL_PREV2, NX, NY,
     &                 DX, DY, DT)
C     variable declarations:
      INTEGER NX, NY
      REAL*8 DX, DY, DT
      REAL*8 SOL(NX,NY), SOL_PREV(NX,NY), SOL_PREV2(NX,NY)

C     update SOL:
      DO 20 J=1, NY
        DO 10 I=1, NX

          SOL(I,J) = 2*SOL_PREV(I,J) - SOL_PREV2(I,J) +
     &              DT*DT*(( SOL_PREV (  I +1,  J ) - 2* SOL_PREV (  I
     &,  J ) +  SOL_PREV (  I -1,  J ))/(  DX *  DX )  +
     &                     ( SOL_PREV (  I ,  J +1) - 2* SOL_PREV (  I
     &,  J ) +  SOL_PREV (  I ,  J -1))/(dy*dy) )

 10   CONTINUE
 20   CONTINUE

      RETURN
      END
```

If you are a Fortran 77 programmer and start using `fccp.py`, never forget that changes in the source code must be performed in files with suffix `.fcp`!

2 Experimenting with Optimization Flags

Experimenting with a compiler's optimization flags is a frequently encountered task in high-performance computing. Measuring the efficiency of a wide range of flags, possibly on different platforms and with different compilers, requires accurate work. This is should not be left as a manual job for a human being. Automating the work in a script makes it easy to repeat the experiments, extend or modify them, try out new compilers and hardware, etc. In this section we shall develop a quite general script for running a benchmark problem with different compilers and compiler flags.

A completely general tool for compiler experimentation would in some sense require us to reimplement a `make` program, which is far beyond scope. However, with hardly no extra work we can generalize a specific example and provide a tool that with minor modifications can be adapted to a wide range of problems. This is typical for scripting: even a short script can be made quite generic, and although the completely generic counterpart is beyond scope, the script can meet surprisingly many demands if you allow for some tuning of the statements in a new application.

Imagine we have some files to be compiled and linked by a set of compilers. The compilers have some common flags and some flags that are specific to a certain compiler. We want to experiment with different settings of the compiler-specific flags, i.e., for each compiler we want to run through a set of different flag alternatives. The resulting executable is to be run in a specified benchmark problem. We need to measure the CPU time, and if possible, grab results from a profiler such as `gprof` or `prof`. The results should of course be nicely formatted for easy inspection. It should be easy to repeat tests on different platforms. The purpose is now to accomplish these tasks in a Python script.

We restrict the attention to source code files written in the Fortran 77 language. Modifying the resulting script to treat C or C++ files is a trivial task. Although most applications are compiled and linked using a makefile, we will in the script issue the commands directly without using any make utility[2]. We introduce a set of common options for the compilation and for the linking step as well as a set of libraries to link with the application. A minimal specification of these options is

```
compile_flags = '-c'
link_flags = '-o %s' % programname
libs = ''
```

More advanced applications might need specifications of, e.g., include and library directories, like in this example:

```
compile_flags = '-c -I %s/include' % os.environ['PREFIX']
link_flags = '-o %s -L %s/lib -L /usr/share/some/lib % \
             (programname, os.environ['PREFIX'])
libs = '-ladvanced_lib -lmylib'
```

The information about a specific compiler is stored in a dictionary with keys reflecting the name of the compiler, a description of the compiler, the common compile and link options, and a list of variable compile options. The latter data are subject to experimentation. Here is a definition of such a dictionary for GNU's Fortran 77 compiler `g77`:

```
g77 = {
    'name'          : 'g77',
    'description'   : 'GNU f77 compiler, v2.95.4',
    'compile_flags' : compile_flags + ' -pg',
    'link_flags'    : link_flags + ' -pg',
    'libs'          : libs,
    'test_flags' :
    ['-O0', '-O1', '-O2', '-O3','-O3 -ffast-math -funroll-loops',],
    'platform_specific_compile_flags' : {},
    'platform_specific_link_flags' : {},
    'platform_specific_libs' : { c1 : '-lf2c' },
    }
```

According to the `test_flags` key, we want to experiment with different levels of optimization (`-O0`, ..., `-O3`) and special optimization flags (e.g., `-ffast-math`). We will typically loop over the `test_flags` values and compile and run the benchmark problem for each value.

On a Sun system, we may want to test Sun's native F77 compiler:

```
# Sun f77 compiler:
Sunf77 = {
    'name' : 'f77',
    'description' : 'Sun f77 compiler, v5.2',
    'compile_flags' : compile_flags,
    'link_flags'    : link_flags,
    'libs'          : '',
    'test_flags' :
    ['-O0', '-O1',  '-fast',],
    'platform_specific_compile_flags' : {},
    'platform_specific_link_flags' : {},
    'platform_specific_libs' : {},
    }
```

The next step is to attach a list of compilers, where each compiler is represented by a dictionary as exemplified above, to a dictionary holding the various platforms where we want to perform the tests. To this end, we declare a dictionary structure `cd` (compiler data), whose keys are the name of specific machines. For example,

[2]Apart from checking a file's date and time, and thereby avoiding unnecessary recompilation, `make` does not perform much else than straightforward operating system commands. These are simpler to deal with in a script written in an easy-to-read language like Python. Avoiding recompilation is not a major issue anymore on today's fast machines.

```
cd = {}

c1 = 'basunus.ifi.uio.no'  # computer 1
cd[c1] = {}
cd[c1]['data'] = 'Linux 2.2.15 i686, 500 MHz, 128 Mb'

c2 = 'skidbladnir.ifi.uio.no'  # computer 2
cd[c2] = {}
cd[c2]['data'] = 'SunOS 5.7, sparc Ultra-5_10'


cd[c1]['compilers'] = [g77]
cd[c2]['compilers'] = [g77, Sunf77]
```

The machine names are taken to be identical to the contents of the `HOST` environment variable. In this way, we can easily extract the name of the current computer inside the script.

A typical experiment with the compilers and flags on a computer can be sketched as follows.

```
# run through the various compiler and options for the
# present host:
host = os.environ['HOST']
for compiler in cd[host]['compilers']:
    for optimization_flags in compiler['test_flags']:
        # construct compilation command from
        #    compiler['name'],
        #    compiler['compile_flags'],
        #    optimization_flags,
        #    compiler['platform_specific_compile_flags'][host],
        #    source code filenames

        <compile...>
        <link (similar construction as the compile command)...>
        <run problem...>
        <report timing results...>
```

The CPU-time measurement can be performed by calling `os.times` before and after running the benchmark program. More detailed information about the efficiency of the code can be obtained from a profiler, such as `gprof` or `prof`. Here we demonstrate how to run `gprof` or `prof` and grab the sorted table of the CPU time spent in each of the program's functions. If the table is long, we display only the first 10 functions:

```
def run_profiler(programname):
    """grab data from gprof/prof output and format nicely"""

    # gprof needs gmon.out (from the last execution of programname)
    if os.path.isfile('gmon.out'):
        # run gprof:
        if not findprograms(['gprof']):
            print 'Cannot find gprof'
            return
        res = os.popen('gprof ' + programname)
        lines = res.readlines()
        failure = res.close()
        if failure:
            print 'Could not run gprof'; return
        # grab the table from the gprof output:
        for i in range(len(lines)):
            if re.search(r'\%\s+cumulative\s+self', lines[i]):
                startline = i
                break
        try:
            # we are interested in the 10 first lines of the table,
            # but if there is a blank line, we stop there
            stopline = 10
            i = 0
            for line in lines[startline:startline+stopline]:
                if re.search(r'^\s*$', line):
                    stopline = i;  break
                i = i + 1
            table = ''.join(lines[startline:startline+stopline])
            print table
            os.remove('gmon.out') # require new file for next run...
        except:
            print 'Could not recognize a table in gmon.out...'; return
    elif os.path.isfile('mon.out'):
```

```
        # run prof:
        if not findprograms(['prof']):
            print 'Cannot find gprof'
            return
        res = os.popen('prof ' + programname)
        lines = res.readlines()
        failure = res.close()
        if failure:
            print 'Could not run prof'; return
        for line in lines[0:10]: print line,

    else:  # no gmon.out or mon.out, cannot run gprof or prof
        print programname,\
              'was not compiled in profiling mode (-pg or -p?)'
        return
```

The `findprograms` functions are found in the module `funcs` in the `py4cs` package.

A possible command-line interface to such a script can have the following items

```
programname file1.f file2.f ... inputfile comment
```

This implies compiling and linking `file1.f`, `file2.f`, and so, then running `programname < inputfile`, and finally reporting the CPU time in an output line containing the `comment` about what type of test we perform. Extracting the command-line information is trivial using Python's convenient subscripting syntax:

```
programname = sys.argv[1]
inputfile   = sys.argv[-2]
comment     = sys.argv[-1]
f77files    = sys.argv[2:-1]
```

A specific application of a script of the type of script described above is found in

```
src/app/wavesim2D/F77/compile.py
```

The Fortran 77 code in the `src/app/wavesim2D/F77` directory solves the two-dimensional wave equation

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(\lambda(x,y)\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\lambda(x,y)\frac{\partial u}{\partial y}\right)$$

by an explicit finite difference scheme over a uniform, rectangular grid. We can think of this equation as modeling 2D water waves. Then u is the water surface elevation, and $\lambda(x, y)$ represents the bottom topography. The `README` file in this directory contains an overview of the code files and the documentation of the involved mathematics and numerics. The finite difference scheme is coded in a separate file, using a C preprocessor macro to simplify the coding and future modifications. A script from the previous section transforms an F77 file with preprocessor directives to standard F77 code.

In the subdirectory `versions` there are several different versions of the code, aimed at testing various high-performance computing aspects:

- file writing versus pure number crunching,
- row-wise versus column-wise traversal of arrays,
- representing λ by an array versus calling functions,
- the effect of if-tests inside long do-loops.

A complete implementation of the type of script explained in this section is found in the file `compile.py`. This script is central for testing the efficiency of the different coding techniques used in the files in the `versions` subdirectory. A simple Bourne shell script `runall.sh` calls up `compile.py` for the different versions of the code. This makes it trivial to test the efficiency of all versions on different platforms, compilers, and optimization flags. The `ranking.py` script extracts the CPU time measurements from the output of `runall.sh` and writes out the relevant lines in sorted order. This acts as a kind of summary of the tests.