LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-738243

A Note on Compiling Fortran

L. E. Busby

September 8, 2017

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

A Note on Compiling Fortran

L.E. Busby, Livermore National Lab

1. *The Key Idea*[1]

Fortran *modules* tend to serialize compilation of large Fortran projects, by introducing dependencies among the source files. If file ***A*** depends on file ***B***, (***A*** *uses* a module defined by ***B***), you must finish compiling ***B*** before you can begin compiling ***A***.

Some Fortran compilers (Intel *ifort*, GNU *gfortran* and IBM *xlf*, at least) offer an option to ''verify syntax'', with the side effect of also producing any associated Fortran module files. As it happens, this option usually runs much faster than the object code generation and optimization phases. For some projects on some machines, it can be advantageous to compile in two passes: The first pass generates the module files, quickly; the second pass produces the object files, in parallel. We achieve a 3.8× speedup in the case study below.

2. *A Case Study*

Miranda[2] is a radiation hydrodynamics code under development at LLNL. The primary language is Fortran 2003, with about 45,000 SLOC spread across 130± source files. In part to simplify dependency analysis, we follow a strict rule that each Fortran file defines exactly one module, with a simple pattern: `abc.f` → `xxx_abc.mod`, where *xxx* is a constant prefix.

Miranda is still on the small side, as multi-physics simulation codes go. Even so, it would already be daunting to reduce inter-file dependencies by analysis of the dependency graph (Fig. 1).[3]

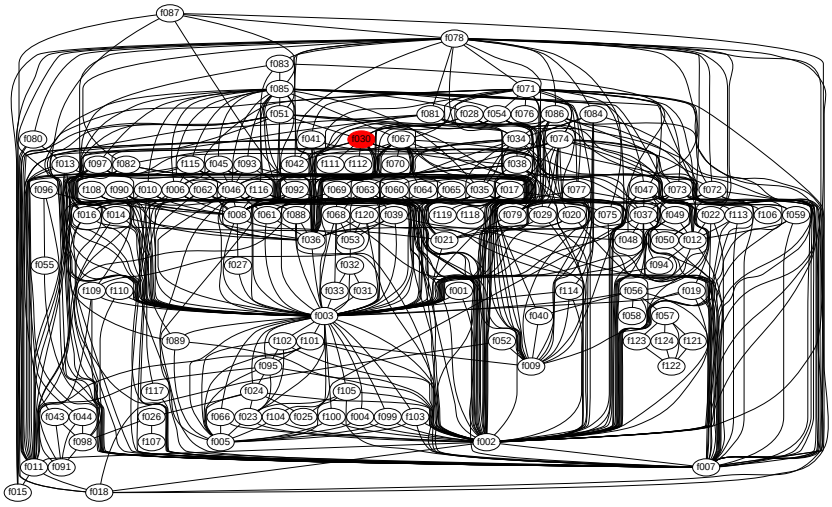

Fig. 1 — Dependency Graph for Miranda Modules

Our build system finds the dependencies automatically, thank goodness, but a full compilation has to satisfy them all, in the order they occur. Fig. 2 shows how this works out in the case of a standard one-pass compile. Time is on the horizontal axis; each bar shows the beginning and

---

1. J. VandeVondele independently experimented with the same idea in 2011, as perhaps have others. See https://gcc.gnu.org/bugzilla/show_bug.cgi?id=47495, comments 9, 12.

2. https://wci.llnl.gov/simulation/computer-codes/miranda

3. The node in red happens to be the single file that takes the longest to compile — about 60 seconds. How would you restructure that file to reduce its build time? How would you restructure (any subset of) files in the system to simplify the dependency graph, and how long would that take?

ending time of the compilation of one file.

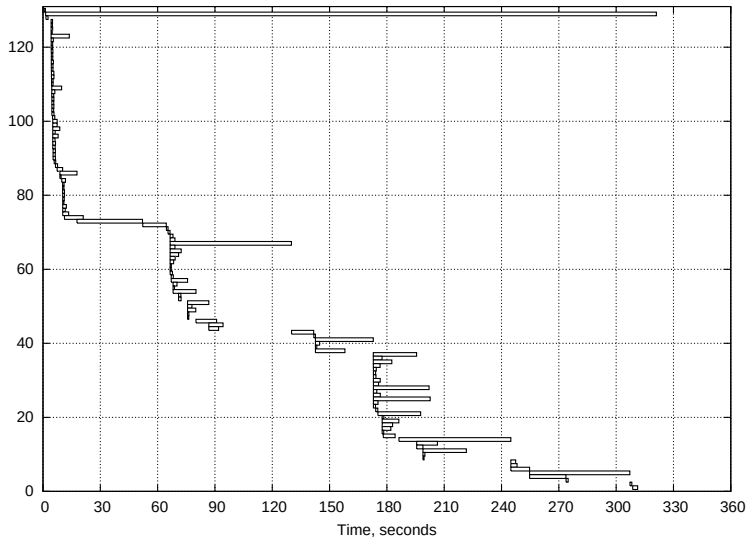

Fig. 2 — One Pass Compilation

Running a parallel make on a node with 36 cores, notice that there are long stretches where only one process is running. The build completes in a little under 330 seconds overall (top bar.)

Now compare Fig. 3, run on the same machine with the same *make -j* command, but using the two-pass compilation procedure.

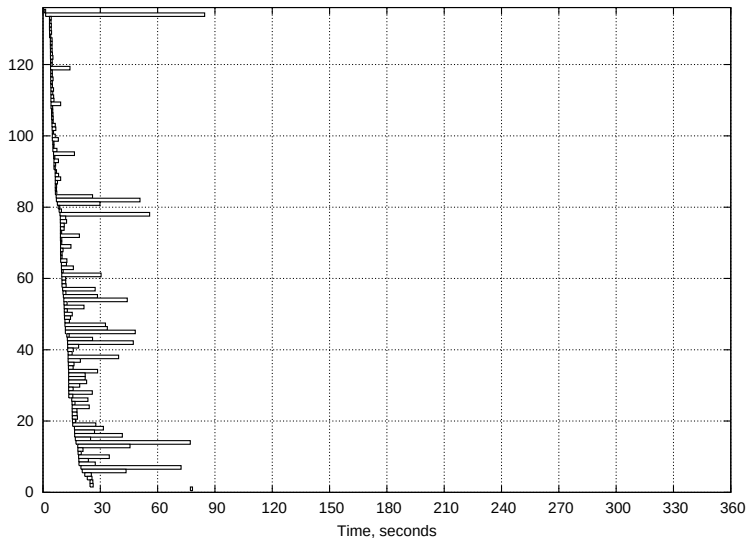

Fig. 3 — Two Pass Compilation

The first pass (not shown above, but included in the total time) constructs just modules, then the second pass constructs the object files, but now with greatly improved parallelism. The two-pass build completes overall in slightly less than 90 seconds, about 3.8× faster than the first run.

Finally, Fig. 4 zooms in on the first few seconds of the two-pass build, showing details of pass one.[4]

4. We have experimented with interleaving, or not, passes one and two. The interleaved version is usually slightly quicker, which I find non-intuitive. It may be case-specific.

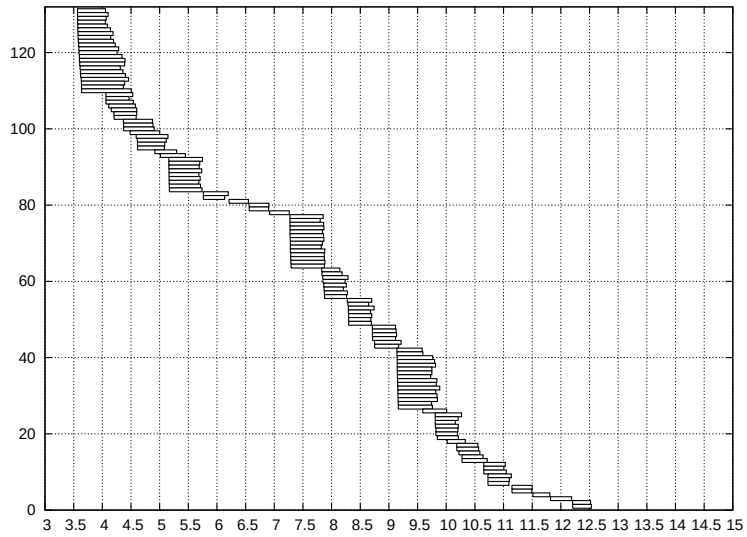

Fig. 4 — Pass One of the Two Pass Run

Note that pass one of a two pass build is still limited by the same dependency graph as the original one pass compile. It's just 30 times faster overall, more or less.

No code changes are required; you can choose 1-pass or 2-pass compiles at build time, which must be the case for us, since not all Fortran compilers can presently support this approach. The two pass compile saves quite a lot of time immediately, but it also makes it much easier to restructure the code to make the job even faster: Split a big file into two parts using any plausible rationale. Even if your new *part_1* depends upon new *part_2*, you can now do most of the work in parallel after pass one completes.

3. A Small Example

An example will hopefully make the process more accessible. Three short Fortran files are given below, followed by a (*gmake*) makefile that carries out two-pass compilation for *aa.f90* and *bb.f90*. The Fortran code is entirely prosaic, but the makefile has several interesting details.

```
! file "prog.f90" ----------------------------------
program foo
  use xx_aa
  call sub_aa()
end

! file "aa.f90" --------------------------------------
module xx_aa
contains
  subroutine sub_aa
    use xx_bb
    integer :: a
    call sub_bb(a)
    print *, 'a is:', a
  end subroutine
end module
```

```
! file "bb.f90" ------------------------------------
module xx_bb
  integer, parameter :: b1=3
contains
  subroutine sub_bb(b)
    integer, intent(out) :: b
    b = b1
  end subroutine
end module

# file "makefile" ---------------------------------
prefix = xx_

# Run as ‘‘make compiler=gnu’’, or intel, or ibm.
gnu.fc = gfortran
gnu.pass1 = -fsyntax-only -c
gnu.pass2 = -J./tmpmod -O2 -c

intel.fc = ifort
intel.pass1 = -syntax-only -c
intel.pass2 = -module ./tmpmod -O2 -c

ibm.fc = xlf
ibm.pass1 = -qnoobject -c
ibm.pass2 = -qmoddir=./tmpmod -O2 -c

pass1 = $($(compiler).fc) $($(compiler).pass1)
pass2 = $($(compiler).fc) $($(compiler).pass2)

prog: prog.o aa.o bb.o; $($(compiler).fc) -o $@ $^
prog.o : prog.f90 aa.o bb.o; $($(compiler).fc) -c $<

$(prefix)%.mod : %.f90
       mkdir -p ./tmpmod
       $(pass1) $< && { test -f $@ || touch $@; ln -fs $< tmpmod/S_$@; }

%.o : $(prefix)%.mod
       $(pass2) ‘readlink tmpmod/S_$<‘

.PHONY: clean
clean: ;rm -rf tmpmod *.mod *.o prog

# Dependencies usually are computed automatically.
xx_aa.mod : xx_bb.mod
```

In the *makefile*, the *pass1* variable adds *-fsyntax-only*, etc., which causes the compiler to generate only the *.mod* file. The *pass2* variable adds *-J./tmpmod*, etc., which causes the compiler to place the second copy of the *.mod* file into a temporary subdirectory (*tmpmod*), to avoid over-writing the first copy, thereby preserving the timestamp from pass one.

The first pass creates the work directory *tmpmod*,[5] the *.mod* file itself, and in addition, a symbolic link that points to the original source file. The *test -f || touch* sequence allows for the case where the source file contains no modules. In that case, the *.mod* file is zero-length, and exists only to

5. There are many other ways to create a temporary work directory, of course.

carry a timestamp.

The symbolic link is used by the second pass to find its source file. This complication is a requirement of breaking the compile into two separate steps: We are effectively declaring that (first) the module file depends on the source file, then (second) the object file depends on the module file. The ordering is correct, but the *.mod* file doesn't contain the location of the original source[6], so the build system separately carries that information from *pass1* to *pass2*.

4. Conclusion

This problem is simple, really, and it has a simple solution. We have a pile of big jobs that all depend on a pool of small, easy to build modules. Make the modules first, and the big pile becomes embarassingly parallel. It's obvious, and the only reason it may seem odd is because our thinking is stuck on the ''fact'' that Fortran compilers must make modules and objects together. It doesn't have to be that way, and for some situations, it is much better to do them one at a time. Intel, IBM, and Gnu fortran have, apparently by accident, given us — barely — the tools needed to do a two pass build. The *make* recipe is a little clumsy, but not unduly so. In practice, it seems quite robust; the example given is very similar to our production build system.

In the second pass, we use a command line option (*-J./tmpmod*, etc.) to place the second copy of a given module ''out-of-the-way''. However, the same option may cause the compiler to also look first in *tmpmod* for generated modules, which raises the possibility of a data race. We have not yet observed this issue, if it is real. Also, the *-fsyntax-only*, etc. option in the first pass does not guarantee that generated modules will be identical to those created by the second pass. Again, we have not observed any issues, yet.

To benefit from this approach, you need a Fortran code with a fair amount of inter-dependency among its files. You need a machine that has more than a few cores available for parallel compiles. And you need to build the code frequently enough so that you care how fast that happens. If you find yourself in that situation, I hope these ideas can be helpful. I would be happy and grateful to receive information about other compilers, corrections, questions or comments about this note: *busby1@llnl.gov*.

---

6. Three elements would be useful to conveniently implement a two pass compile:

    1. An option to generate just the *.mod* files;
    2. An option to (given *.mod* files) generate just the object files;
    3. A means to store (in pass one) the full path to the original source file in the *.mod* files, then recover it for the second pass. Pass two (object) depends on pass one (module), but we still need to access the original source to build the object.

    Compiler writers: Please take note.