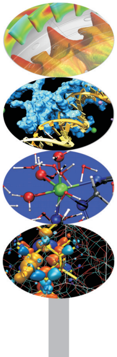
Programmazione Avanzata / 2

Francesco Salvadore

CINECA Roma - SCAI Department



Roma, 2014

Outline

Librerie scientifiche

Makefile

HPC development tools

- What do I need to develop my HPC application? At least:
  - A compiler, e.g. GNU, Intel, PGI, PathScale, Sun
  - Code editor
- Several tools may help you (even for non HPC applications)
  - Debugger, e.g. gdb and DDD, TotalView, Allinea DDT
  - Profiler, e.g. gprof, Scalasca, Tau, Vampir
  - Project management, e.g. make, projects
  - Revision control, e.g. svn, git, cvs, mercurial
  - Generating documentation, e.g. doxygen
  - Source code repository, e.g. sourceforge, github, google.code
  - Data repository, currently under significant increase
  - and more . . .

Code editor

- You can select the code editor among a very wide range
  - from the light and fast text editors, e.g. VIM, emacs
  - to the more sophisticated Integrated development environment (IDE), e.g. Visual Studio, Kdevelop, Ecplise
  - or you have intermediate options, e.g. Geany
- The choice obviously depends on the complexity and on the software tasks
- ...but also on your personal taste

Project management

- Non trivial programs are hosted in several source files and link libraries
- Different types of files require different compilation
  - different optimization flags
  - different languages may be mixed, too
  - compilation and linking require different flags
  - and the code could work on different platforms
- During development (and debugging) several recompilations are needed, and we do not want to recompile all the source files but only the modified ones
- How to deal with it?
  - use the IDE (with plug-ins) and their project files to manage the content (e.g. Eclipse)
  - use language-specific compiler features
  - use external utilities, e.g. Make or CMake

GNU Make

- “Make is a tool which controls the generation of executables and other non-source files of a program from the program’s source files”
- Make gets its knowledge from a file called the makefile, which lists each of the non-source files and how to compute it from other files
- When you write a program, you should write a makefile for it, so that it is possible to use Make to build and install the program
- GNU Make has some powerful features for use in makefiles, beyond what other Make versions have

Preparing and Running Make

- To prepare to use make, you have to write a file that describes:
  - the relationships among files in your program
  - commands for updating each file
- Typically, the executable file is updated from object files, which are in turn made by compiling source files
- Once a suitable makefile exists, each time you change some source files, the command

```
make -f <makefile_name>
```

  suffices to perform all necessary recompilations
- If **`-f`** option is missing, the default names **`makefile`** or **`Makefile`** are used

Rules

- A simple makefile consists of “rules”:

```
target ... : prerequisites ...
        recipe
        ...
        ...
```

  - a **`target`** is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as clean
  - a **`prerequisite`** is a file that is used as input to create the target. A target often depends on several files.
  - a **`recipe`** is an action that make carries out. A recipe may have more than one command, either on the same line or each on its own line. Recipe commands must be preceded by a tab character.
- By default, make starts with the first target (default goal)

My first rule

- A simple rule:

```
    foo.o : foo.c defs.h
            gcc -c -g foo.c
```

- This rule says two things
  - how to decide whether foo.o is out of date: it is out of date if it does not exist, or if either foo.c or defs.h is more recent than it
  - how to update the file foo.o: by running gcc as stated. The recipe does not explicitly mention defs.h, but we presume that foo.c includes it, and that is why defs.h was added to the prerequisites.
- Remember the tab character before starting the recipe lines!

A simple example in C

- The main program is in laplace2d.c file
  - includes two header files: timing.h and size.h
  - calls functions in two source files: update_A.c and copy_A.c
- update_A.c and copy_A.c includes two header files: laplace2d.h and size.h
- A possible (naive) Makefile

```
laplace2d_exe: laplace2d.o update_A.o copy_A.o
	gcc -o laplace2d_exe laplace2d.o update_A.o copy_A.o

laplace2d.o: laplace2d.c timing.h size.h
	gcc -c laplace2d.c

update_A.o: update_A.c laplace2d.h size.h
	gcc -c update_A.c

copy_A.o: copy_A.c laplace2d.h size.h
	gcc -c copy_A.c

.PHONY: clean
clean:
	rm -f laplace2d_exe *.o
```

How it works

- The default goal is (re-)linking laplace2d_exe
- Before make can fully process this rule, it must process the rules for the edited files that depends on it, which in this case are the object files
- The object files, according to their own rules, are recompiled if the source files, or any of the header files named as prerequisites, is more recent than the object file, or if the object file does not exist
- Note: in this makefile .c and .h are not the targets of any rules, but this could happen it they are automatically generated
- After recompiling whichever object files need it, make decides whether to relink according to the same “updating” rules.
- Try to follow the path: what happens if, e.g., laplace2d.h is modified?

A simple example in Fortran

- The main program is in laplace2d.f90 file
  - uses two modules named prec and timing
  - calls subroutines in two source files: update_A.f90 and copy_A.f90
- update_A.f90 and copy_A.f90 use only prec module
- sources of prec and timing modules are in the prec.f90 and timing.f90 files
- Beware of the Fortran modules:
  - program units using modules require the mod files to exist
  - a target may be a list of files: e.g., both timing.o and timing.mod depend on timing.f90 and are produced compiling timing.f90
- Remember: the order of rules is not significant, except for determining the default goal

A simple example in Fortran / 2

```
laplace2d_exe: laplace2d.o update_A.o copy_A.o prec.o timing.o
	gfortran -o laplace2d_exe prec.o timing.o laplace2d.o update_A.o copy_A.o

prec.o prec.mod: prec.f90
	gfortran -c prec.f90

timing.o timing.mod: timing.f90
	gfortran -c timing.f90

laplace2d.o: laplace2d.f90 prec.mod timing.mod
	gfortran -c laplace2d.f90

update_A.o: update_A.f90 prec.mod
	gfortran -c update_A.f90

copy_A.o: copy_A.f90 prec.mod
	gfortran -c copy_A.f90

.PHONY: clean
clean:
	rm -f laplace2d_exe *.o *.mod
```

Phony Targets and clean

- A phony target is one that is not really the name of a file; rather it is just a name for a recipe to be executed when you make an explicit request.
  - avoid target name clash
  - improve performance
- **`clean`**: an ubiquitous target

```
.PHONY: clean
clean:
        rm *.o temp
```

- Another common solution: since `FORCE` has no prerequisite, recipe and no corresponding file, make imagines this target to have been updated whenever its rule is run

```
clean: FORCE
        rm *.o temp
FORCE:
```

Variables

- The previous makefiles have several duplications
  - error-prone and not expressive
- Use variables!
  - define
    `objects = laplace2d.o update_A.o copy_A.o`
  - and use as **`$(objects)`**

```
objects := laplace2d.o update_A.o copy_A.o

laplace2d_exe: $(objects)
        gcc -o laplace2d_exe $(objects)

laplace2d.o: laplace2d.c timing.h size.h
        gcc -c laplace2d.c

update_A.o: update_A.c laplace2d.h size.h
        gcc -c update_A.c

copy_A.o: copy_A.c laplace2d.h size.h
        gcc -c copy_A.c

.PHONY: clean
clean:
        rm -f laplace2d_exe *.o
```

More Variables

- Use more variables to enhance readability and generality
- Modifying the first four lines it is easy to modify compilers and flags

```
CC       := gcc
CFLAGS   := -O2
CPPFLAGS :=
LDFLAGS  :=

objects  := laplace2d.o update_A.o copy_A.o

laplace2d_exe: $(objects)
        $(CC) $(CFLAGS) $(CPPFLAGS) -o laplace2d_exe $(objects) $(LDFLAGS)

laplace2d.o: laplace2d.c timing.h size.h
        $(CC) $(CFLAGS) $(CPPFLAGS) -c laplace2d.c

update_A.o: update_A.c laplace2d.h size.h
        $(CC) $(CFLAGS) $(CPPFLAGS) -c update_A.c

copy_A.o: copy_A.c laplace2d.h size.h
        $(CC) $(CFLAGS) $(CPPFLAGS) -c copy_A.c

.PHONY: clean
clean:
        rm -f laplace2d_exe *.o
```

Implicit rules

- There are still duplications: each compilation needs the same command except for the file name
  - immagine what happens with hundred of files!
- What happens if Make does not find a rule to produce one or more prerequisite (e.g., and object file)?
- Make searches for an implicit rule, defining default recipes depending on the processed type
  - C programs: n.o is made automatically from n.c with a recipe of the form

```
$(CC) $(CPPFLAGS) $(CFLAGS) -c
```

  - C++ programs: n.o is made automatically from n.cc, n.cpp or n.C with a recipe of the form

```
$(CXX) $(CPPFLAGS) $(CXXFLAGS) -c
```

  - Fortran programs: n.o is made automatically from n.f, n.F (`$(CPPFLAGS)` only for .F)

```
$(FC) $(FFLAGS) $(CPPFLAGS) -c
```

Pattern rules

- Implicit rules allow for saving many recipe lines
  - but what happens is not clear reading the Makefile
  - and you are forced to use a predefined structure and variables
  - to clarify the types to be processed, you may define **`.SUFFIXES`** variable at the beginning of Makefile

```
.SUFFIXES:
.SUFFIXES: .o .f
```

- You may use re-define an implicit rule by writing a pattern rule
  - a pattern rule looks like an ordinary rule, except that its target contains one character `%`
  - usually written as first target, does not become the default target

```
%.o : %.c
       $(CC) -c $(OPT_FLAGS) $(DEB_FLAGS) $(CPP_FLAGS) $< -o $@
```

- Automatic variables are usually exploited
  - `$@` is the target
  - `$<` is the first prerequisite (usually the source code)
  - `$^` is the list of prerequisites (useful in linking stage)

Running make targets

- It is possible to select a specific target to be updated, instead of the default goal (remember **`clean`**)

```
        make copy_A.o
```

  - of course, it will update the chain of its prerequisite
  - useful during development when the full target has not been programmed, yet
- And it is possible to set target-specific variables as (repeated) target prerequisites
- Consider you want to write a Makefile considering both GNU and Intel compilers
- Use a default goal which is a help to inform that compiler must be specified as target

C example

```
CPPFLAGS  :=
LDFLAGS   :=

objects   := laplace2d.o update_A.o copy_A.o

.SUFFIXES :=
.SUFFIXES := .c .o

%.o: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $<

help:
        @echo "Please select gnu or intel compilers as targets"

gnu: CC       := gcc
gnu: CFLAGS   := -O3
gnu: $(objects)
        $(CC) $(CFLAGS) $(CPPFLAGS) -o laplace2d_gnu $(objects) $(LDFLAGS)

intel: CC     := icc
intel: CFLAGS := -fast
intel: $(objects)
        $(CC) $(CFLAGS) $(CPPFLAGS) -o laplace2d_intel $(objects) $(LDFLAGS)

laplace2d.o: laplace2d.c timing.h size.h
update_A.o : update_A.c laplace2d.h size.h
copy_A.o   : copy_A.c laplace2d.h size.h

.PHONY: clean
clean:
        rm -f laplace2d_gnu laplace2d_intel $(objects)
```

Fortran example

```
LDFLAGS :=
objects := prec.o timing.o laplace2d.o update_A.o copy_A.o
.SUFFIXES:
.SUFFIXES: .f90 .o .mod

%.o: %.f90
        $(FC) $(FFLAGS) -c $<
%.o %.mod: %.f90
        $(FC) $(FFLAGS) -c $<

help:
        @echo "Please select gnu or intel compilers as targets"
gnu: FC         := gfortran
gnu: FCFLAGS    := -O3
gnu: $(objects)
        $(FC) $(FCFLAGS) -o laplace2d_gnu $^ $(LDFLAGS)
intel: FC       := ifort
intel: FCFLAGS := -fast
intel: $(objects)
        $(FC) $(CFLAGS) -o laplace2d_intel $^ $(LDFLAGS)

prec.o prec.mod:     prec.f90
timing.o timing.mod: timing.f90
laplace2d.o:         laplace2d.f90 prec.mod timing.mod
update_A.o:          update_A.f90 prec.mod
copy_A.o:            copy_A.f90 prec.mod

.PHONY: clean
clean:
        rm -f laplace2d_gnu laplace2d_intel $(objects) *.mod
```

Defining variables

- Another way to support different compilers or platforms is to include a platform specific file (e.g., make.inc) containing the needed definition of variables

```
    include make.inc
```

- Common applications feature many make.inc.<platform_name> which you have to select and copy to make.inc before compiling

- When invoking `make`, it is also possible to set a variable

```
    make OPTFLAGS=-O3
```

  - this value will override the value inside the Makefile
  - unless `override` directive is used
  - but override is useful when you want to add options to the user defined options, e.g.

```
               override CFLAGS += -g
```

Variable Flavours

- The variables considered until now are called *simply expanded* variables, are assigned using `:=` and work like variables in most programming languages.
- The other flavour of variables is called *recursively expanded*, and is assigned by the simple **=**
  - recursive expansion allows to make the next assignments working as expected

```
CFLAGS       = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

  - but may lead to unpredictable substitutions or even impossibile circular dependencies

```
CFLAGS       = $(CFLAGS) -g
```

- You may use **+=** to add more text to the value of a variable
  - acts just like normal **=** if the variable in still undefined
  - otherwise, exactly what **+=** does depends on what flavor of variable you defined originally
- Use recursive variables only if needed

Wildcards

- A single file name can specify many files using wildcard characters: `*`, `?` and `[...]`
- Wildcard expansion depends on the context
  - performed by make automatically in targets and in prerequisites
  - in recipes, the shell is responsible for
  - what happens typing **`make print`** in the example below? (The automatic variable **`$?`** stands for files that have changed)

```
print: *.c
       lpr -p $?
       touch print
```

  - if you define

```
objects = *.o
foo : $(objects)
       gcc -o foo $(objects)
```

    it is expanded only when is used and it is not expanded if no .o file exists: in that case, foo depends on a oddly-named `.o` file
  - use instead the `wildcard` function:

```
objects := $(wildcard *.o)
```

Conditional parts of Makefile

- Environment variables are automatically transformed into make variables
- Variables could be not enough to generalize rules
  - e.g., you may need non-trivial variable dependencies
- Immagine your application has to be compiled using GNU on your local machine mac_loc, and Intel on the cluster mac_clus
- You can catch the hostname from shell and use a conditional statement (**`$SHELL`** is not exported)

```
SHELL := /bin/sh
HOST  := $(shell hostname)
ifeq ($(HOST),mac_loc)
   CC     := gcc
   CFLAGS := -O3
endif
ifeq ($(HOST),mac_clus)
   CC     := icc
   CFLAGS := -fast
endif
```

- Be careful on Windows systems!

Directories for Prerequisites

- For large systems, it is often desirable to put sources and headers in separate directories from the binaries
- Using Make, you do not need to change the individual prerequisites, just the search paths
- A **`vpath`** pattern is a string containing a `%` character.
  - **`%.h`** matches files that end in **`.h`**

```
    vpath %.c foo
    vpath %   blish
    vpath %.c bar
```

will look for a file ending in `.c` in foo, then blish, then bar

- using `vpath` without specifying directory clears all search paths associated with patterns

Directories for Prerequisites / 2

- When using directory searching, recipe generalizing is mandatory

```
      vpath %.c src
      vpath %.h ../headers
      foo.o : foo.c defs.h hack.h
              gcc -c $< -o $@
```

- Again, automatic variables solve the problem
- And implicit or pattern rules may be used, too
- Directory search also works for linking libraries using prerequisites of the form `-lname`
- `make` will search for the file libname.so and, if not found, for libname.a first searching in vpath and then in system directory

```
foo : foo.c -lcurses
        gcc $^ -o $@
```

Advanced topics

- Functions, also user-defined
  - e.g., define objects as the list of file which will be produced from all .c files in the directory

```
objects := $(patsubst %.c,%.o,$(wildcard *.c))
```

  - e.g., sorts the words of list in lexical order, removing duplicate words

```
headers := $(sort math.h stdio.h timer.h math.h)
```

- Recursive make, i.e. make calling chains of makes
  - MAKELEVEL variable keeps the level of invocation

Standard Targets (good practice)

- all $\rightarrow$ Compile the entire program. This should be the default target
- install $\rightarrow$ Compile the program and copy the executables, libraries, and so on to the file names where they should reside for actual use.
- uninstall $\rightarrow$ Delete all the installed files
- clean $\rightarrow$ Delete all files in the current directory that are normally created by building the program.
- distclean $\rightarrow$ Delete all files in the current directory (or created by this makefile) that are created by configuring or building the program.
- check $\rightarrow$ Perform self-tests (if any).
- install-html/install-dvi/install-pdf/install-ps $\rightarrow$ Install documentation
- html/dvi/pdf/ps $\rightarrow$ Create documentation

Much more...

- Compiling a large application may require several hours
- Running make in parallel can be very helpful, e.g. to use 8 processes

```
make -j8
```

  - but not completely safe (e.g., recursive make compilation)
- There is much more you could know about **`make`**
  - this should be enough for your in-house application
  - but probably not enough for understanding large projects you could encounter

`http://www.gnu.org/software/make/manual/make.html`

Hands-on

- Write a Makefile managing a simple mixed C/Fortran project
- Sources are in `c_src` and `f90_src` directories
- How to compile and build is described in the README file:

```
target all_c ==> Main and functions in C:
gcc avg_main.c avg_fun.c

target all_f90 ==> Main and functions in Fortran:
gfortran avg_fun.f90 avg_main.f90

target main_f90_fun_c ==> Fortran Main and C functions:
gcc -c avg_fun.c
gfortran avg_fun.o mod_wrapC.f90 avg_main.f90

target main_c_fun_f90 ==> C Main and Fortran functions:
gcc -c avg_main.c
gfortran avg_main.o avg_fun.f90 mod_wrapF.f90
```

- Use `vpath`, implicit rules, a default help target, and be careful about Fortran modules

Hands-on: solution

```
SHELL     := /bin/sh
FC        := gfortran
CC        := gcc
CFLAGS    := -O3
FCFLAGS   := -O3

.SUFFIXES :=
.SUFFIXES := .c .f90 .o

vpath %.c   c_src
vpath %.f90 f90_src

%.o: %.c
        $(CC) $(CFLAGS) $(CPPFLAGS) -c $<
%.o: %.f90
        $(FC) $(FFLAGS) -c $<

help:
        @echo "Please specify a valid target:"
        @echo "all_c/all_f90/main_f90_fun_c/main_c_fun_90"

..............
```

Hands-on: solution / 2

```
..............

all_c: avg_main_c.o avg_fun_c.o
        $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

all_f90: avg_main_f90.o avg_fun_f90.o
        $(FC) $(FCFLAGS) -o $@ $^ $(LDFLAGS)

main_f90_fun_c: avg_fun_c.o mod_wrapC.o avg_main_f90.o
        $(FC) $(FCFLAGS) -o $@ $^ $(LDFLAGS)

main_c_fun_f90: avg_main_c.o mod_wrapF.o avg_fun_f90.o
        $(FC) $(FCFLAGS) -o $@ $^ $(LDFLAGS)

avg_main_f90.o: statistics.mod
mod_wrapF.o: statistics.mod
statistics.mod: avg_fun_f90.o mod_wrspC.o

.PHONY: clean
clean:
        rm -f all_c all_f90 main_f90_fun_c main_c_fun_f90
rm -f *.o *.mod
```

Rights & Credits