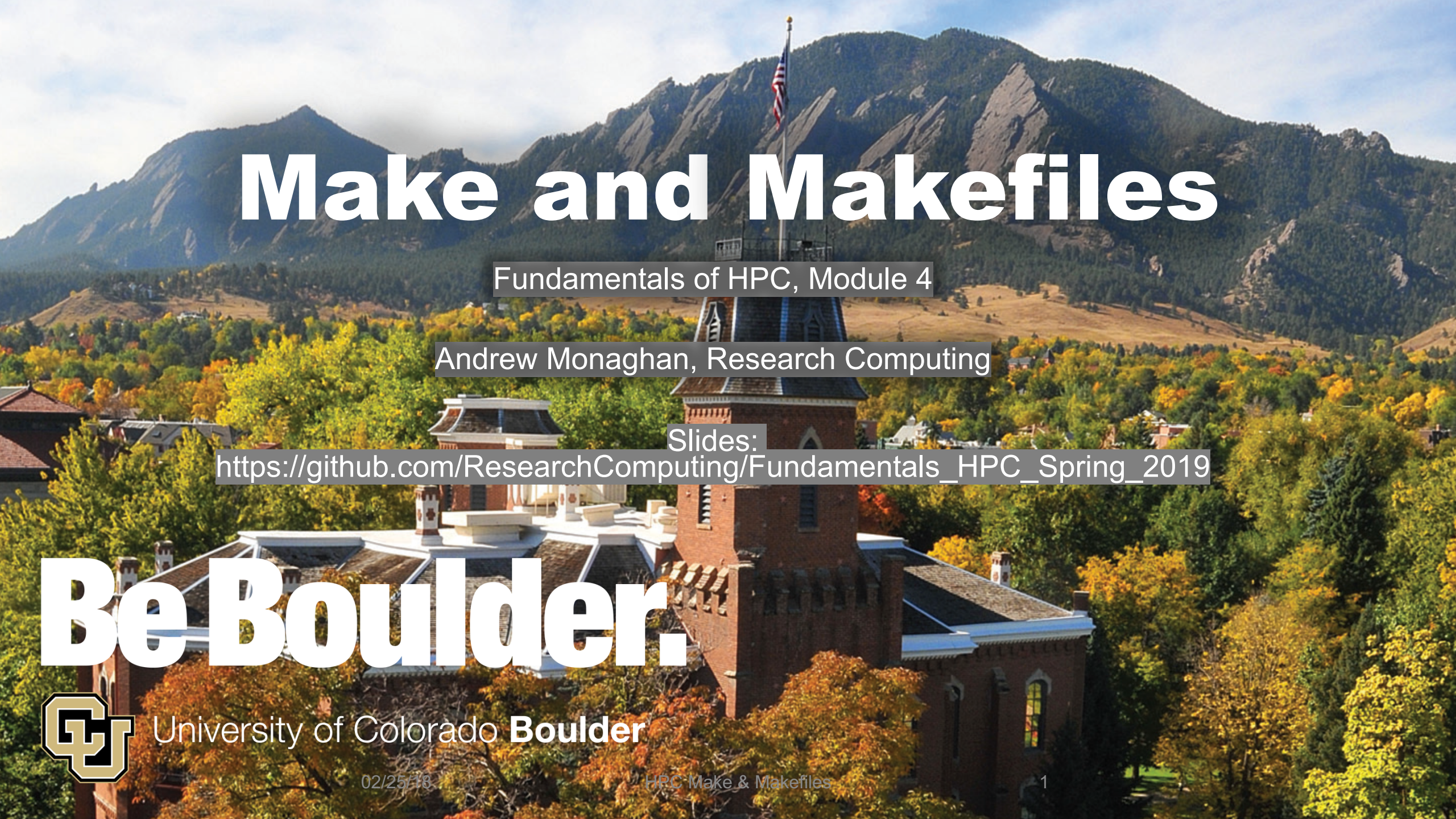
Make and Makefiles
Fundamentals of HPC, Module 4
Andrew Monaghan, Research Computing
Slides:
https://github.com/ResearchComputing/Fundamentals_HPC_Spring_2019
Be Boulder.
University of Colorado Boulder

Outline

- Overview

- Makefile examples:
  - Simple Makefile (1) (targets, dependencies)
  - Simple Makefile (2) (variables)
  - Simple Makefile (3) (wildcards, portability)

- ./configure: downloading, configuring & compiling code with make

Before we begin

- ssh user00NN@tlogin1.rc.colorado.edu
  - Use password provided
- ssh scompile
- ml intel
- cd /scratch/summit/$USER/makefiles

Editing files: Nano survival guide

- Ctrl+o save (need to confirm filename)
- Ctrl+x exit
- Ctrl+k cut
- Ctrl+u paste

GNU Make

- Helps manage compilation of source code.
- Examine the files in the makefiles subdirectory
- Compile:
  - **icc hello.c hellofunc.c -I. -o hello.exe**
- Drawbacks?
  - Have to remember compilation syntax
  - Possibly machine-specific (no Intel on my laptop?)

Simple Makefile (1)

- Create a file named Makefile in the makefiles directory
- Add these lines (use tab, not 4 spaces):

```
hello.exe: hello.c hellofunc.c
    icc -I . hello.c hellofunc.c -o hello.exe
```

TAB

- This is a **rule** for making the **target file** hello.exe
- Type ***make***
- Checks target and dependencies for:
  - existence
  - modifications (based on timestamp)

Simple Makefile (1)

- Make executes every command indented below the target/dependency line.
- Can use any Linux commands

```
hello.exe: hello.c hellofunc.c
    TAB  echo "Compiling hello.exe"
    TAB  icc -I . hello.c hellofunc.c -o hello.exe
```

- Make displays the command executed on the screen
- Suppress by prepending @ symbol: echo → @echo

Simple Makefile (1)

- Standard to include a target named clean:

```
hello.exe: hello.c hellofunc.c
    @echo "Compiling hello.exe"
    icc -I . hello.c hellofunc.c -o hello.exe
clean:
    rm –f  *.o
    rm –f  hello.exe
```

Simple Makefile (1)

- Complication. Run these commands:
  - touch clean
  - make clean
- A file named clean will conflict with our target named clean.
- Solution – add .PHONY target:

```
.PHONY: clean
```

- Understood that targets in .PHONY list are not files

Simple Makefile (1): A Good Start

```
.PHONY: clean

hello.exe: hello.c hellofunc.c
	@echo "Compiling hello.exe"
	icc -I . hello.c hellofunc.c -o hello.exe

clean:
	rm –f  *.o
	rm –f  hello.exe
```

Variables in Make

- Compilers and optimization flags may differ between machines
- Use variables to avoid rewriting compilation commands for each new machine
- Best to do at top of Makefile

Define variables via **=**

```
CC = icc
```

Evaluate variables via **$( )**

```
echo $(CC)
```

Simple Makefile (2)

- Modify your makefile to include these changes
- (keep clean and .PHONY as they were)

```
CC = icc
INCLUDE_FLAGS = -I .
OPT_FLAGS = -O2
CFLAGS  = $(INCLUDE_FLAGS) $(OPT_FLAGS)

hello.exe: hello.c hellofunc.c
        @echo "Compiling hello.exe"
        $(CC) $(CFLAGS)  hello.c hellofunc.c -o hello.exe
```

Simple Makefile (2): Special Variables

- These variables are useful within rules/recipes
- The variable **$@**
  - Refers to the rule target (e.g., hello.exe)
- The variable **$<**
  - Refers to the 1st dependency (e.g., hello.c)
- The variables **$^**
  - Refers to all dependencies (e.g., hello.c hellofunc.c)
- Exercise:
  - Rewrite the rule for hello.exe using **$@** and **$^**
    - And create a variable called 'PROG' that equals 'hello.exe'

My Solution:

I defined a new variable…

```
…
CFLAGS = $(OPT_FLAGS) $(INCLUDE_FLAGS)
PROG = hello.exe
```

Now hello.exe appears in only 1 place.

```
$(PROG): hello.c hellofunc.c
        @echo "Compiling “ $(PROG) “."
        $(CC) $(CFLAGS) $^ -o $@

clean:
        rm -f *.o
        rm -f $(PROG)
```

Simple Makefile (3)

- We often have a number of object files needed to build the program.
- Make this small modification:

```
$(PROG): hello.c hellofunc.c
```

```
$(PROG): hello.o hellofunc.o
```

- By default, Make uses CFLAGS and CC to build the .o files

- We can control this behavior ourselves

Simple Makefile (3): Wildcards

- The % symbol acts a wildcard.
- Add the following target/rule:

```
%.o: %.c
	@echo "Compling "$@ " using " $<
	$(CC) $(CFLAGS) -c $< -o $@
```

- This tells Make:
  - Use this rule if a file ending in .o is needed
  - Before file.o can be created, file.c must exist
  - Build file.o via: $(CC) $(CFLAGS) -c file.c -o file.o

Simple Makefile (3): Portability

- If we move to a new machine, we may need to modify:
  - CC, OPT_FLAGS, INCLUDE_FLAGS
- To do so, we must edit the Makefile
- OK enough for small projects; really bad otherwise
- Good practice:
  - Create machine-specific definitions file
  - Include in Makefile

Simple Makefile (3): Portability

- Create a file named machine.def with these three lines

```
CC = icc
OPT_FLAGS = -O2
INCLUDE_FLAGS = -I .
```

- In your Makefile, make this replacement:

```
CC = icc
OPT_FLAGS = -O2
INCLUDE_FLAGS = -I .
```

```
include machine.def
```

- Makefile works on any machine now
- only machine.def is modified

./configure

- ./configure is another way source code is made to be portable
- User then runs "./configure <options>" to create machine-specific Makefile, or to create a supplementary file with machine-specific options, which is included in a pre-existing generic Makefile.
  - E.g., to create a Makefile that specifies the compiled code will be installed in a specified directory:

```
./configure --prefix=/projects/$USER
```

  - …and also specify optimization flags for the C and C++ compilers

```
./configure --prefix=/projects/$USER CFLAGS ='-g -O2' CXXFLAGS='-g -O2'
```

  - …let's try an example…

./configure

- Let's download and compile '*samtools'* using make and a Makefile.

```
cd /scratch/summit/$USER/makefiles/example_make
cat commands_reference
```

- The *commands_reference* file contains a list of commands to run to compile and install the *samtools* code, and to explore what happens along the way. Let's look at the file and try the example….

```
# This file contains the commands you would need to download and
# Install samtools, an open source set of tools for manipulating
# large nucleotide sequence alignments in SAM format

#Download the source code
wget https://github.com/samtools/samtools/releases/download/1.9/samtools-1.9.tar.bz2

#Untar/untip the source code directory
tar -xvjf samtools-1.9.tar.bz2

#Go into the source code directory
cd samtools-1.9

#Look around:
ls -l
vi Makefile

#Load the Intel compiler and run ./configure
ml intel
./configure --prefix=/projects/$USER/samtools CFLAGS='-g -O2 -xCORE-AVX2'

#examine what was created by running ./configure
ls -ltr
vi configure.mk

#Now make and install the code using 4 processors (-j 4)
make -j 4
make install
```

…Next week you'll learn more about the process of compiling and linking code. Knowing about make and makefiles provides a foundation.

Thank You!

- Survey: http://tinyurl.com/curc-survey18
- Make documentation: https://www.gnu.org/software/make/manual/make.html
- Slides: https://github.com/ResearchComputing/HPC_Short_Course_Fall_2018
- Contact: Andrew.Monaghan@colorado.edu