From Ideas to Execution

Building Code on Supercomputers

ATPESC 2015
Aron Ahmadia
Continuum Analytics

building code is complex

Today's scientists regularly work with "stacks" of code instead of single libraries

- A minimal "stack" includes MPI and Fortran 77
- But many of today's applications are frequently more complicated

We manage complexity with abstractions

- This is no different when building software

Programming is simply another form of writing…

interpreted vs. compiled

- a high level language can be either interpreted or compiled into machine language before being executed on a computer
  - an interpreted program is indirectly executed through an interpreter, a separate program that usually runs on the same computer
  - a compiled program is instead translated to machine code by a compiler before being executed directly on the computer
- compiler - translates a high level language to object code
- linker - collects and merges object files into single executables
- assembler - assembly -> machine code, archiver - static objects -> libraries

Programming is simply another form of writing ELF binaries

elf: executable and linkable format

- very simple composition
  - ELF header (architecture, endianness)
  - run-time information (segments)
  - link/load-time information (sections)
  - program data
- tools for examining elf binaries
  - file, nm, ldd, readelf, objdump, patchelf

a basic taxonomy of programming languages

machine language - the native language executed by central processing units on computers, represented usually in hexadecimal

assembly language - an isomorphic mapping between a machine language and a set of human understandable text mnemonics

object code, a collection of separate, named sequences of machine language and associated data and metadata

high level language - a programming languages designed to be abstracted from the specifics of a particular computer architecture

scripting language - a programming language that controls one or more software applications

a basic taxonomy of elf binaries

relocatable - holds code and data suitable for linking with other object files to create an executable or a shared object file (.o)

executable - holds a program suitable for execution

shared object file - holds code and data suitable for linking either through the link editor (.so) or the dynamic linker

EVERY DEPENDENCY IS IMPLICIT UNTIL YOU MAKE IT EXPLICIT

is code sufficient?

An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.

David Donoho, 1998.

is code sufficient?

An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is **the complete software development environment** and the complete set of instructions which generated the figures.

David Donoho, 1998.

thirty years ago

“Here’s my .f77 file! I hope you’ve got a computer with 1 MB of RAM…”

ten years ago

"Here's a bunch of source code and a Makefile, don't worry I just included all of the dependencies in my code…"

today

"These results rely on a development LAMMPS, an unreleased version of NAMD, a patched version of PETSc, our ported LLVM stack, and the latest commit on this branch from my GitHub repository. Good luck!"

explicit and implicit dependencies

- explicit
  - compilation units (.f, .c, .cxx)
  - external library names
- implicit (by default)
  - header files / template libraries
  - compiler/linker versions, flags, settings
  - external library versions

including vs. linking

Includes are a pre-processing step in compiling relocatable objects

- -I flag
- Directly copies and pastes included files into compilation unit

Links are references to other relocatable object archives (libfoo.a) or shared object files (libfoo.so)

- -L flag (Also use the -R flag when linking dynamically)
- resolved either statically at link-time or dynamically at run-time

building scientific software

- most freely distributed scientific software is designed to be built with make
  - make [Feldman, 1978] - a software construction tool for building targets from their corresponding dependencies
    - dependencies are frequently hierarchical (e.g. program: objects: source)
    - builds can be partial (only some targets updated) or complete
- the most popular accompanying tool is the GNU build system
  - GNU build system [MacKenzie, 1991]- a suite of programming tools for portably building source-code based distributions

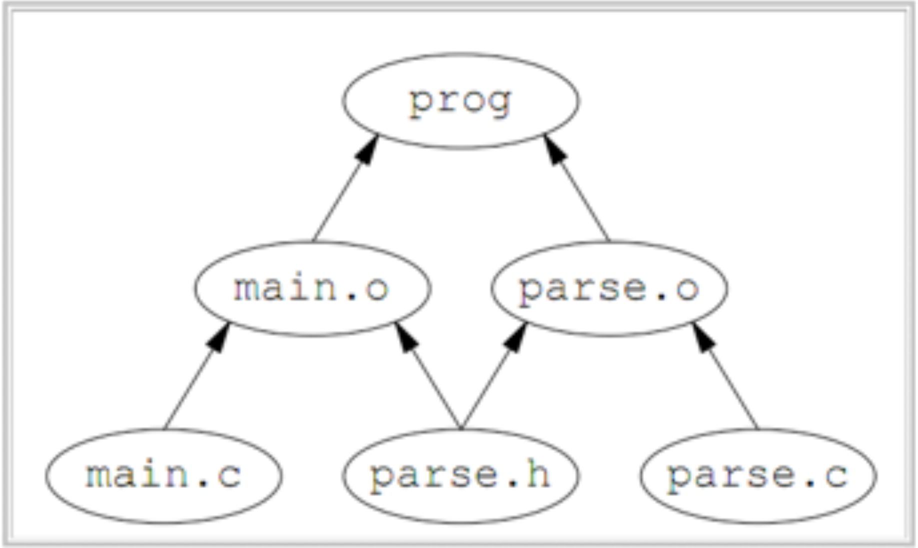
how 'make all' works

1. look in the current directory for a Makefile, a text file containing declarations of targets, their dependencies, and a set of rules for building targets from their respective dependencies
2. construct a directed acyclic graph (DAG) mapping targets to dependencies.
3. traverse the directed acyclic graph and, for each target, find all its dependencies, and update the target if it is older than any of its dependencies (out-of-date)

- a common idiom is to issue make recursively, with a Makefile in each subdirectory of the source tree -- this can be dangerous!

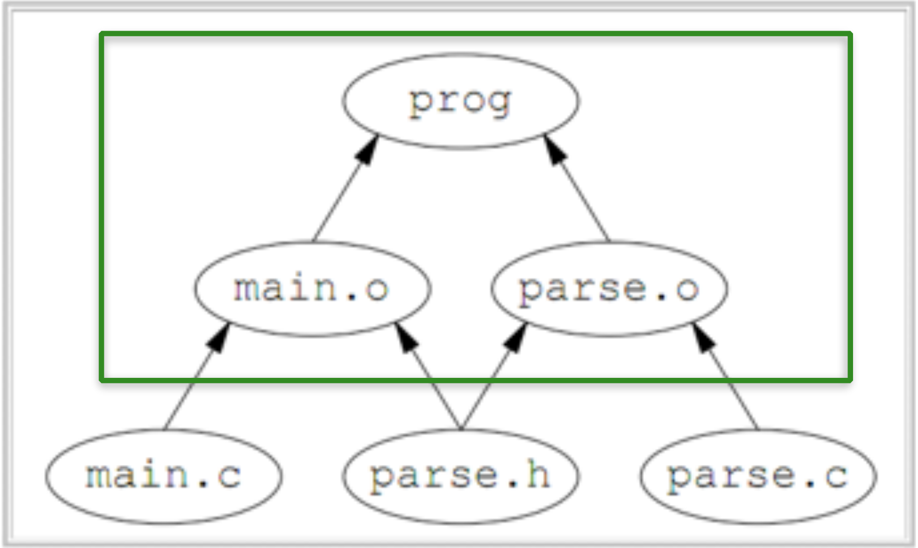
sample makefile/DAG

```
OBJ = main.o parse.o
prog: $(OBJ)
  $(CC) -o $@ $(OBJ)
main.o: main.c parse.h
  $(CC) -c main.c
parse.o: parse.c parse.h
  $(CC) -c parse.c
```



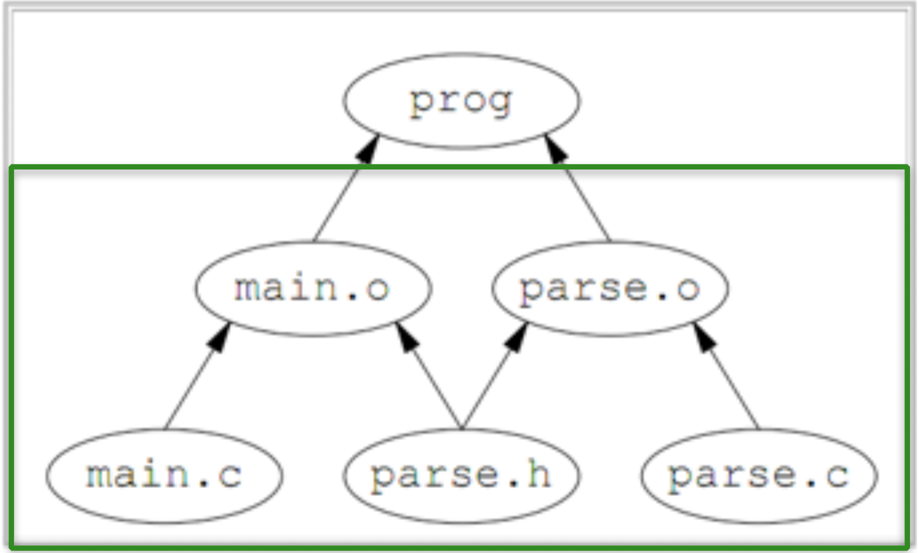
targets

```
OBJ = main.o parse.o
prog: $(OBJ)
  $(CC) -o $@ $(OBJ)
main.o: main.c parse.h
  $(CC) -c main.c
parse.o: parse.c parse.h
  $(CC) -c parse.c
```



dependencies

```
OBJ = main.o parse.o
prog: $(OBJ)
  $(CC) -o $@ $(OBJ)
main.o: main.c parse.h
  $(CC) -c main.c
parse.o: parse.c parse.h
  $(CC) -c parse.c
```



A WELL DESIGNED MAKEFILE CAN CAPTURE A COMPLETE SCIENTIFIC WORKFLOW

pop quiz: scalability

assume a makefile with defined N targets and M independent dependent source files

make considers a target to be 'out-of-date' when the time stamp of any of its dependencies are more recent than the target

what is the minimum number of source files to be considered when evaluating whether to rebuild a single target? all N targets?

pop quiz: scalability

what is the minimum number of source files to be considered when evaluating whether to rebuild a single target?

0. a real target with no dependencies is never rebuilt (but make will still check to see if it exists when you try to build it)

all N targets?

M. All source files must be considered, if any of the dependencies are more recent than their corresponding targets, then the targets must be rebuilt.

deficiencies in standalone make

doesn’t know anything about your system

cannot determine your dependencies for you

recursive make can improperly fail to update targets

scales poorly to very large builds

tools that use make

GNU build system (autoconf, automake, and libtool)	provide configuration, makefile-generation, and library management, implicit dependency tracking
CMake	provides cross-platform configuration and build setup, can generate makefiles in UNIX as well as project files for IDEs such as Visual Studio, implicit dependency scanner

don’t ignore the shell

Automate with scripts

Refactor, organize, and control with shell functions

Manage the build and run context through environment variables

Use subshells to safely execute commands without modifying the current shell’s working directory or environment variables

QUESTIONS?

further reading/sources

See the previous slides for direct links to the software packages listed
GNU Make

[*] Python for Software Design - How to Think Like a Computer Scientist by Allen B. Downey

[*] Program Library HOWTO by David A. Wheeler

[*] How to Write Shared Libraries by Ulrich Drepper

[*] Grace Hopper by Wikipedia Community

[*] Build System Rules and Algorithms by Mike Shal

photo credits

Slides 9, 10 (Grace Hopper) from the US Department of the Navy Naval Historical Center, NH 96919-KN, by James S. Davis, 1984

Slides 20, 21, and 22 (Makefile DAG example) used diagrams lifted from "Recursive Make Considered Harmful", by Peter Miller, 1987