Software Engineering and Process for HPC Scientific Software

Anshu Dubey

With several slides from
Brian Van Straalen
Phil Colella

ATPSEC 2013
Why is Software Process Important

• Modern scientific computing is no longer a solo effort
  – Most interesting modeling questions that could be simulated by the heroic individual programming scientist have already been investigated
  – “Productivity language” that are meant to alleviate the complexity of programming high performance software have not delivered yet
  – Thus, coding is complicated and requires division of roles and responsibilities.
• Working together on a common code is difficult unless there is a software process
Software Process Components

• For All Codes
  – **Code Repository**
  – Build Process
  – Code Architecture
  – Coding Standards
  – Verification Process
  – Maintenance Practices

• If Publicly Distributed code
  – Distribution Policies
  – Contribution Policies
  – Attribution Policies
Code Repositories

• **Centralized Version Control**
  – **CVS** the first one to be heavily deployed
  – **Subversion** the most commonly used

• **Distributed Version Control**
  – Most popular ones are **Git** and **Mercurial**
  – Synchronization through exchange of patches
  – One can maintain multiple local branches
  – Makes for a much easier co-existence of production and development
  – Gate keeping can become challenging
Subversion: SVN

• Central Repository system.
  – There is one master version of the state of the code
• Users have “check outs” or “working copy” of the master repository
• Can access the master repository via several mechanisms
  – rsh connection
  – ssh connection
  – svnserver
  – All user interaction is considered a client-side operation
  – Transactional protocol
Working with Repositories

- Checkout
- update
  - Also a merging/concurrent process, as with CVS
- diff [filename|directory]
- add [filename|directory]
- commit [ filename|directory]
- delete [filename|directory]
- merge
- branches
Working with Repositories

- You check out the head or some branch of the repository
  - This is your working copy
  - When you have modified your working copy and you want to save your work you check in
- What is stored is the difference between versions
  - Minimization of information since the whole history must be maintained
  - When you do update the “diff” is merged into your working copy
- You can roll back as much as you like
  - Because the whole change history is maintained
  - Tools exist that translate the history and logs into web readable information

Example: FLASH repository
What Else Can You Do With Repositories

• Managing branches
  – Individuals working on some development that they don’t want to have colliding with other developers
  – Tag a stable branch
  – Separate production from development
  – Manage multiple production projects
• Also help with backtracking for verification
• Aid in reproducibility of results (within the limits of having the same software stack and hardware available)
• In short those of us who have been using it, wouldn’t live without it
Unusual Use

• Supporting multiple set of projects from different branches is more recent at FLASH
• A hierarchy of project and production branches
• A stringent merge and test schedule is important
• How we did it:
  – Turned one of the branches into main development branch
  – Turned trunk into the merge area
  – Enforced a merge schedule
  – Enforced a policy of prioritizing the fixing of whatever broke in the merge.
Software Process Components

• For All Codes
  – Code Repository
  – **Build Process**
  – Code Architecture
  – Coding Standards
  – Verification Process
  – Maintenance Practices

• If Publicly Distributed code
  – Distribution Policies
  – Contribution Policies
  – Attribution Policies
Build Process

• Multiple files, individual file compilation does not scale beyond a point
• If the code runs on many different platforms then each software stack will have its own peculiarities
• The code may want to use available libraries, getting them all built consistently may be challenging
• For all of these reasons it is worth investing in a managed build process
• Usually a combination of configuration and make
• Autoconf, perl scripts, python for configuration
• GNU Make for compilation
Configuration - FLASH Example : Setup Script

Python code links together needed physics and tools for a problem

- Traverse the FLASH source tree and link necessary files for a given application to the object directory
- Creates a file defining global constants set at build time
- Builds infrastructure for mapping runtime parameters to constants as needed
- Configures Makefiles properly
- Determine solution data storage list and create Flash.h
- Generate files needed to add runtime parameters to a given simulation.
- Generate files needed to parse the runtime parameter file.
Setup works with Config file and local makefile snippets

• FLASH-specific syntax
• Define dependencies at all levels in the source tree:
  – Lists required, requested, exclusive modules
• Declare solution variables, fluxes
• Declare runtime parameters
  – Sets defaults and allowable ranges – do it early!
  – Documentation – start line with “D”
• Variables, Units are additive down the directory tree
• Provides warnings to prevent dumb mistakes
• Consolidates makefile snippets into a complete makefile
# Configuration File for setup Stirring Turbulence

REQUIRES Driver
REQUIRES physics/sourceTerms/Stir/StirMain
REQUIRES physics/Eos
REQUIRES physics/Hydro
REQUIRES Grid
REQUIRES I0

# include IO routine only if IO unit included
LINKIF IO_writeIntegralQuantities.F90 IO/IOMain
LINKIF IO_writeUserArray.F90 IO/IOMain/hdf5/parallel
LINKIF IO_readUserArray.F90 IO/IOMain/hdf5/parallel

LINKIF IO_writeUserArray.F90.pnetcdf IO/IOMain/pnetcdf
LINKIF IO_readUserArray.F90.pnetcdf IO/IOMain/pnetcdf

D c_ambient reference sound speed
D rho_ambient reference density
D mach reference mach number
PARAMETER c_ambient REAL 1.e0
PARAMETER rho_ambient REAL 1.e0
PARAMETER mach REAL 0.3

GRIDVAR mvrt

USESETUPVARS nDim
IF nDim <> 3
  SETUPERROR At present Stir turb works correctly only in 3D. Use ./setup StirTurb -3d blah blah
ENDIF
Simple setup

Sample Units File

INCLUDE Driver/DriverMain/TimeDep
INCLUDE Grid/GridMain/paramesh/Paramesh3/PM3_package=headers
INCLUDE Grid/GridMain/paramesh/Paramesh3/PM3_package/mpi_source
INCLUDE Grid/GridMain/paramesh/Paramesh3/PM3_package/source
INCLUDE Grid/localAPI
INCLUDE IO/IOMain/hdf5/serial/PM
INCLUDE PhysicalConstants/PhysicalConstantsMain
INCLUDE RuntimeParameters/RuntimeParametersMain
INCLUDE Simulation/SimulationMain/Sedov
INCLUDE flashUtilities/general
INCLUDE physics/Eos/EosMain/Gamma
INCLUDE physics/Hydro/HydroMain/split/PPM/PPMKernel
INCLUDE physics/Hydro/HydroMain/utilities
GNU Make

• Main purpose: turn a set of source code into a library or executable.

• Only two kinds of objects in a Makefile
  – Variables (lists of strings)
  – Rules

• Only a few kinds of flow control
  – ifeq/ifneq/else/endif
  – No forms or looping available, no jumps, no recursion.

• Most difficulties arising from make are related to
  – Non-trivial variable parsing of the makefile(s)
  – Rules can fire and trigger in non-obvious ways
The Two type of Variables in GNU Make

- **Recursively Expanded Variables** “=“
  
  ```
  foo = $(bar)
  bar = $(ugh)
  ugh = Huh?
  all:;echo $(foo)
  > make all
  Huh?
  ```

- Variable is executed at the time it is used in a command
- = means build up a symbol table for this name
- Notice $. Like in shell, there is the value ‘bar’ and the variable named ‘bar’
• Good points:
  – Order doesn’t matter!
  – Can declare a variable as the composite of many other variables that can filled in by other parts of the Makefile
  – CFLAGS = $(DEBUG_FLAGS) $(OPT_FLAG) $(LIB_FLAGS)
  – Lets a makefile build up sophisticated variables when you don’t know all the suitable inputs, or what parts of the Makefile they will come from
    • >make all DIM=3

• Bad points:
  – Future = declarations can clobber what you specified
  – The last = declaration in the linear parsing of a Makefile is the only one that matters
• Simply Expanded Variables “:=“
  – Immediate mode variable.
  – The variable is assigned it’s value based on the current state of the Makefile parsing
  – No symbol chain is created.
  – Specific to GNU Make
• Often just an easier to understand variable.
  – It acts like variables you know in other languages.
  – can use for appending
    • CFLAGS := $(CFLAGS) –c –e –mmx
Rules

targets : prerequisites
[TAB] recipe

• prerequisites are also called “sources”

• Simple example

clobber.o : clobber.cpp clobber.h config.h
[TAB] g++ -c -o clobber.o clobber.cpp

clob.ex : clobber.o killerApp.o
[TAB] g++ -o clob.ex cobber.o killerApp.o
More powerful rules

• Pattern Rules
  
  
  %.o : %.cpp

  $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@

  #Gives a pattern that can turn a .cpp file into a .o file

• Multitarget Rules

  %.f %.H : %.ChF

• Suffix Rules

  - .c.o:

    • $(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
Other Makefile commands

- include
- $(MAKE)
  - calling a makefile from inside a recipe
  - $(MAKELEVEL) can be looked at to see how deep the call stack is
- export
  - send variables from this level of make to lower makelevels
- subst
  - CFLAGS:= $(CFLAGS) $(subst FALSE,,$(subst TRUE,-DCH_MPI $(mpicppflags),$(MPI)))
- foreach
  - libincludes = $(foreach i,${LibNames},-I$(CHOMBO_HOME)/src/$i)
What the “make” program does

• Much mental confusion about make comes from thinking that the Makefile is the make program
  – Remember: Makefile is only Variables & Rules
• make:
  – parses all of your Makefile, builds up variable chains (overriding variables defined on command line)
  – builds up rules database, then looks at what target the user has specified
  – then attempts to create a chain of rules from the files that exist to the targets specified.
    • recursive “=“ variables in source-target expressions are evaluated
  – Using the date stamp on files discovered in the chain make executes recipes to deliver the target.
    • “=“ variables are evaluated in recipes.
Demonstration of the pervasive Make ‘error’

FooBar = trendy
F:= fashion
vars:
   @echo $(FooBar) $(F)

ifeq ($(F),fashion)
   FooBar=tragic
endif
F:= comedy
>make vars
tragic comedy
>


FLASH Example : Makefile

• Each supported site has a specific Makefile.h
  – Variable defined for library locations
  – Variables for compiler being used
  – Flags for using in “debug”, “test” or “opt” mode
  – Other necessary flags

• Every directory can have a makefile snippet
  – Exploits the recursively expanded variables
  – Makes sure to include the source files defined at that level unless they are inherited
  – Specified local dependencies

• The file snippets are consolidated into Makefile.Unit for every unit

• The Makefile.h and Makefile.Unit are “included” in the generated Makefile
Software Process Components

• For All Codes
  – Code Repository
  – Build Process
  – **Code Architecture**
  – Coding Standards
  – Verification Process
  – Maintenance Practices

• If Publicly Distributed code
  – Distribution Policies
  – Contribution Policies
  – Attribution Policies

Hal with HACC architecture next