Software Developement Tools

Introduction to software building

SED & friends

Outline

1. an example
2. what is software building?
3. tools
4. about CMake

1

an example

Example

- we want to build the parallel program solving heat equation:

Example

- we want to build the parallel program solving heat equation:
  - 0 level: hit the following line: `mpicc -Wall -o heat_par heat_par.c heat.c mat_utils.c -lm`

Example

- we want to build the parallel program solving heat equation:
  - 0 level: hit the following line: `mpicc -Wall -o heat_par heat_par.c heat.c mat_utils.c -lm`
  - 0.1 level: write a script (bash, csh, Zsh, ...)

Example

- we want to build the parallel program solving heat equation:
  - 0 level: hit the following line: `mpicc -Wall -o heat_par heat_par.c heat.c mat_utils.c -lm`
  - 0.1 level: write a script (bash, csh, Zsh, ...)
- drawbacks:

Example

- we want to build the parallel program solving heat equation:
  - 0 level: hit the following line: `mpicc -Wall -o heat_par heat_par.c heat.c mat_utils.c -lm`
  - 0.1 level: write a script (bash, csh, Zsh, ...)
- drawbacks:
  - non-portability: works only on a Unix systems, with `mpicc` shortcut and MPI libraries and headers installed in standard directories

Example

- we want to build the parallel program solving heat equation:
  - 0 level: hit the following line: `mpicc -Wall -o heat_par heat_par.c heat.c mat_utils.c -lm`
  - 0.1 level: write a script (bash, csh, Zsh, ...)
- drawbacks:
  - non-portability: works only on a Unix systems, with `mpicc` shortcut and MPI libraries and headers installed in standard directories
  - every build, we compile all files

Example

- 0.5 level: write a `Makefile`

```
MPICC = mpicc
CFLAGS = -Wall -O3
INCLUDE = -I. -I/usr/include/mpi
HFILES = heat.h
CFILES_PAR = heat.c heat_par.c mat_utils.c
LIB_PAR =-lm
EXEC_PAR= heat_par

.SUFFIXES: .c .o

${EXEC_PAR}: ${CFILES_PAR:.c=.o}
  ${MPICC} -o ${EXEC_PAR} ${CFILES_PAR:.c=.o}
.c.o:
  ${CC} ${CFLAGS} ${INCLUDE} -c $*.c -o $*.o
```

Example

- 0.5 level: write a `Makefile`

```
MPICC = mpicc
CFLAGS = -Wall -O3
INCLUDE = -I. -I/usr/include/mpi
HFILES = heat.h
CFILES_PAR = heat.c heat_par.c mat_utils.c
LIB_PAR =-lm
EXEC_PAR= heat_par

.SUFFIXES: .c .o

${EXEC_PAR}: ${CFILES_PAR:.c=.o}
  ${MPICC} -o ${EXEC_PAR} ${CFILES_PAR:.c=.o}
.c.o:
  ${CC} ${CFLAGS} ${INCLUDE} -c $*.c -o $*.o
```

- drawbacks:

Example

- 0.5 level: write a `Makefile`

```
MPICC = mpicc
CFLAGS = -Wall -O3
INCLUDE = -I. -I/usr/include/mpi
HFILES = heat.h
CFILES_PAR = heat.c heat_par.c mat_utils.c
LIB_PAR =-lm
EXEC_PAR= heat_par

.SUFFIXES: .c .o

${EXEC_PAR}: ${CFILES_PAR:.c=.o}
  ${MPICC} -o ${EXEC_PAR} ${CFILES_PAR:.c=.o}
.c.o:
  ${CC} ${CFLAGS} ${INCLUDE} -c $*.c -o $*.o
```

- drawbacks:
  - `/usr/include/mpi` is hardcoded ☹

Example

- 0.5 level: write a `Makefile`

```
MPICC = mpicc
CFLAGS = -Wall -O3
INCLUDE = -I. -I/usr/include/mpi
HFILES = heat.h
CFILES_PAR = heat.c heat_par.c mat_utils.c
LIB_PAR =-lm
EXEC_PAR= heat_par

.SUFFIXES: .c .o

${EXEC_PAR}: ${CFILES_PAR:.c=.o}
  ${MPICC} -o ${EXEC_PAR} ${CFILES_PAR:.c=.o}
.c.o:
  ${CC} ${CFLAGS} ${INCLUDE} -c $*.c -o $*.o
```

- drawbacks:
  - `/usr/include/mpi` is hardcoded ☹
  - use auto configuration tricks: `pkg-config`

Example

- 0.5 level: write a `Makefile`

```
MPICC = mpicc
CFLAGS = -Wall -O3
INCLUDE = -I. -I/usr/include/mpi
HFILES = heat.h
CFILES_PAR = heat.c heat_par.c mat_utils.c
LIB_PAR =-lm
EXEC_PAR= heat_par

.SUFFIXES: .c .o

${EXEC_PAR}: ${CFILES_PAR:.c=.o}
  ${MPICC} -o ${EXEC_PAR} ${CFILES_PAR:.c=.o}
.c.o:
  ${CC} ${CFLAGS} ${INCLUDE} -c $*.c -o $*.o
```

- drawbacks:
  - `/usr/include/mpi` is hardcoded ☹
  - use auto configuration tricks: `pkg-config`
  - How to build for non unix OSes ?

Example of `CMakeLists.txt`

```
cmake_minimum_required (VERSION 3.1)
project (Heat)

find_package(MPI)

set(HEAT_PAR_SOURCES heat_par.c heat.c mat_utils.c)
add_executable(HeatPar ${HEAT_PAR_SOURCES})
target_include_directories(HeatPar PRIVATE ${MPI_C_INCLUDE_DIRS})
target_link_libraries(HeatPar ${MPI_C_LIBRARIES} m)

# enable testing
enable_testing()

add_test(patcHeatPar4 ${MPIEXEC} ${MPIEXEC_NUMPROC_FLAG} 4
         ./HeatPar 10 10 200 2 2 0)
```

2

what is software building?

Why use a building tool ?

- developer's POV:

Why use a building tool ?

- developer's POV:
  - enforce software portability

Why use a building tool ?

- developer's POV:
  - enforce software portability
  - detect the building environment by auto configuration

Why use a building tool ?

- developer's POV:
  - enforce software portability
  - detect the building environment by auto configuration
  - use native tools to build the software (best integration to the OS)

Why use a building tool ?

- developer's POV:
  - enforce software portability
  - detect the building environment by auto configuration
  - use native tools to build the software (best integration to the OS)
  - add tests to check the build

Why use a building tool ?

- developer's POV:
  - enforce software portability
  - detect the building environment by auto configuration
  - use native tools to build the software (best integration to the OS)
  - add tests to check the build
- user's POV:

Why use a building tool ?

- developer's POV:
  - enforce software portability
  - detect the building environment by auto configuration
  - use native tools to build the software (best integration to the OS)
  - add tests to check the build
- user's POV:
  - install program must be an **easy** process

Why use a building tool ?

- developer's POV:
  - enforce software portability
  - detect the building environment by auto configuration
  - use native tools to build the software (best integration to the OS)
  - add tests to check the build
- user's POV:
  - install program must be an **easy** process
  - possible use of binaries packages could accelerate and simplify installation

Requirements

- building is more than compilation

Requirements

- building is more than compilation
- an effective building tool must manage different:

Requirements

- building is more than compilation
- an effective building tool must manage different:
  - languages

Requirements

- building is more than compilation
- an effective building tool must manage different:
  - languages
  - compilers

Requirements

- building is more than compilation
- an effective building tool must manage different:
  - languages
  - compilers
  - libraries implementations

Requirements

- building is more than compilation
- an effective building tool must manage different:
  - languages
  - compilers
  - libraries implementations
  - operating systems or distributions of an OS

Requirements

- building is more than compilation
- an effective building tool must manage different:
  - languages
  - compilers
  - libraries implementations
  - operating systems or distributions of an OS
  - programming tools

Requirements

- building is more than compilation
- an effective building tool must manage different:
  - languages
  - compilers
  - libraries implementations
  - operating systems or distributions of an OS
  - programming tools
- possibly embed some testing process

Requirements

- building is more than compilation
- an effective building tool must manage different:
  - languages
  - compilers
  - libraries implementations
  - operating systems or distributions of an OS
  - programming tools
- possibly embed some testing process
- integrate some packaging systems

3

tools

Tools

- `make`: basic, robust tool but no cross-platform

Tools

- `make`: basic, robust tool but no cross-platform
- Maven for Java

Tools

- `make`: basic, robust tool but no cross-platform
- Maven for Java
- GNU autotools: efficient and powerful; most widely used tool in free software. Limited to Unix (POSIX) environments

Tools

- `make`: basic, robust tool but no cross-platform
- Maven for Java
- GNU autotools: efficient and powerful; most widely used tool in free software. Limited to Unix (POSIX) environments
- CMake: complete tools suite. Cross-platform (Unix, MacOS, Windows)

Tools

- `make`: basic, robust tool but no cross-platform
- Maven for Java
- GNU autotools: efficient and powerful; most widely used tool in free software. Limited to Unix (POSIX) environments
- CMake: complete tools suite. Cross-platform (Unix, MacOS, Windows)
- Scons: smart tool written in Python, also cross-platform

4

about CMake

CMake: features

- created in 1999 by ITK (mimic pcmake, a build tool for VTK)

CMake: features

- created in 1999 by ITK (mimic pcmake, a build tool for VTK)
- integrated tools suite for building software

CMake: features

- created in 1999 by ITK (mimic pcmake, a build tool for VTK)
- integrated tools suite for building software
  - CMake: makefiles generator

CMake: features

- created in 1999 by ITK (mimic pcmake, a build tool for VTK)
- integrated tools suite for building software
  - CMake: makefiles generator
  - CTest: testing utility

CMake: features

- created in 1999 by ITK (mimic pcmake, a build tool for VTK)
- integrated tools suite for building software
  - CMake: makefiles generator
  - CTest: testing utility
  - CPack: package builder

CMake: features

- created in 1999 by ITK (mimic pcmake, a build tool for VTK)
- integrated tools suite for building software
  - CMake: makefiles generator
  - CTest: testing utility
  - CPack: package builder
  - CDash: web interface to present a summary of builds/tests

CMake: features

- created in 1999 by ITK (mimic pcmake, a build tool for VTK)
- integrated tools suite for building software
  - CMake: makefiles generator
  - CTest: testing utility
  - CPack: package builder
  - CDash: web interface to present a summary of builds/tests
- cross-platform, and works with IDEs (Eclipse, Visual, etc.)

CMake: features

- created in 1999 by ITK (mimic pcmake, a build tool for VTK)
- integrated tools suite for building software
  - CMake: makefiles generator
  - CTest: testing utility
  - CPack: package builder
  - CDash: web interface to present a summary of builds/tests
- cross-platform, and works with IDEs (Eclipse, Visual, etc.)
- support for C, C++, Fortran

CMake: features

- created in 1999 by ITK (mimic pcmake, a build tool for VTK)
- integrated tools suite for building software
  - CMake: makefiles generator
  - CTest: testing utility
  - CPack: package builder
  - CDash: web interface to present a summary of builds/tests
- cross-platform, and works with IDEs (Eclipse, Visual, etc.)
- support for C, C++, Fortran
- graphical user interface (`cmake-gui`) and text interface (`ccmake`)

CMake: philosophy

- developer writes one or several `CMakeLists.txt` files in a language which abstracts:

CMake: philosophy

- developer writes one or several `CMakeLists.txt` files in a language which abstracts:
  - compiler supports (GCC, ICC, MSVC, XCode, etc.)

CMake: philosophy

- developer writes one or several `CMakeLists.txt` files in a language which abstracts:
  - compiler supports (GCC, ICC, MSVC, XCode, etc.)
  - file operations (copy, move, delete, mkdir)

CMake: philosophy

- developer writes one or several `CMakeLists.txt` files in a language which abstracts:
  - compiler supports (GCC, ICC, MSVC, XCode, etc.)
  - file operations (copy, move, delete, mkdir)
  - system inspection (headers, libraries)

CMake: philosophy

- developer writes one or several `CMakeLists.txt` files in a language which abstracts:
  - compiler supports (GCC, ICC, MSVC, XCode, etc.)
  - file operations (copy, move, delete, mkdir)
  - system inspection (headers, libraries)
- Run: `mkdir build && cd build && cmake ..`
  CMake processes all subdirectory and their `CMakeLists.txt` and generates `Makefiles`

CMake: philosophy

- developer writes one or several `CMakeLists.txt` files in a language which abstracts:
  - compiler supports (GCC, ICC, MSVC, XCode, etc.)
  - file operations (copy, move, delete, mkdir)
  - system inspection (headers, libraries)
- Run: `mkdir build && cd build && cmake ..`
  CMake processes all subdirectory and their `CMakeLists.txt` and generates `Makefiles`
- you can tweak some options in the cache
  `make edit_cache`

CMake: philosophy

- developer writes one or several `CMakeLists.txt` files in a language which abstracts:
  - compiler supports (GCC, ICC, MSVC, XCode, etc.)
  - file operations (copy, move, delete, mkdir)
  - system inspection (headers, libraries)
- Run: `mkdir build && cd build && cmake ..`
  CMake processes all subdirectory and their `CMakeLists.txt` and generates `Makefiles`
- you can tweak some options in the cache
  `make edit_cache`
- compiles your project with color and percentages: `make`

CMake: philosophy

- developer writes one or several `CMakeLists.txt` files in a language which abstracts:
  - compiler supports (GCC, ICC, MSVC, XCode, etc.)
  - file operations (copy, move, delete, mkdir)
  - system inspection (headers, libraries)
- Run: `mkdir build && cd build && cmake ..`
  CMake processes all subdirectory and their `CMakeLists.txt` and generates `Makefiles`
- you can tweak some options in the cache
  `make edit_cache`
- compiles your project with color and percentages: `make`
- run some test: `make test` or `ctest --parallel 9`

CMake: philosophy

- developer writes one or several `CMakeLists.txt` files in a language which abstracts:
  - compiler supports (GCC, ICC, MSVC, XCode, etc.)
  - file operations (copy, move, delete, mkdir)
  - system inspection (headers, libraries)
- Run: `mkdir build && cd build && cmake ..`
  CMake processes all subdirectory and their `CMakeLists.txt` and generates `Makefiles`
- you can tweak some options in the cache
  `make edit_cache`
- compiles your project with color and percentages: `make`
- run some test: `make test` or `ctest --parallel 9`
- install the software from sources: `make install`

CMake: philosophy

- developer writes one or several `CMakeLists.txt` files in a language which abstracts:
  - compiler supports (GCC, ICC, MSVC, XCode, etc.)
  - file operations (copy, move, delete, mkdir)
  - system inspection (headers, libraries)
- Run: `mkdir build && cd build && cmake ..`
  CMake processes all subdirectory and their `CMakeLists.txt` and generates `Makefiles`
- you can tweak some options in the cache
  `make edit_cache`
- compiles your project with color and percentages: `make`
- run some test: `make test` or `ctest --parallel 9`
- install the software from sources: `make install`
- create a package: `make package`

A few words about the syntax

- CMake is a 'scripting language'; test it with
  `cmake -P script.cmake`

A few words about the syntax

- CMake is a 'scripting language'; test it with
  `cmake -P script.cmake`
- it can define variables: `set(var value); message(STATUS ${var})`

A few words about the syntax

- CMake is a 'scripting language'; test it with
  `cmake -P script.cmake`
- it can define variables: `set(var value); message(STATUS ${var})`
- CMake has control statements: `foreach`, `if`, `while`, `function`

A few words about the syntax

- CMake is a 'scripting language'; test it with
  `cmake -P script.cmake`
- it can define variables: `set(var value)`; `message(STATUS ${var})`
- CMake has control statements: `foreach`, `if`, `while`, `function`
- support some data structures: `list`, `string`

A few words about the syntax

- CMake is a 'scripting language'; test it with `cmake -P script.cmake`
- it can define variables: `set(var value); message(STATUS ${var})`
- CMake has control statements: `foreach`, `if`, `while`, `function`
- support some data structures: `list`, `string`
- support for functions! beware of scope of variables relative to directory

A few words about the syntax

- CMake is a 'scripting language'; test it with `cmake -P script.cmake`
- it can define variables: `set(var value); message(STATUS ${var})`
- CMake has control statements: `foreach`, `if`, `while`, `function`
- support some data structures: `list`, `string`
- support for functions! beware of scope of variables relative to directory
- support modularity trough `CMakeModules` files

A few words about the syntax

- CMake is a 'scripting language'; test it with
  `cmake -P script.cmake`
- it can define variables: `set(var value); message(STATUS ${var})`
- CMake has control statements: `foreach`, `if`, `while`, `function`
- support some data structures: `list`, `string`
- support for functions! beware of scope of variables relative to directory
- support modularity trough `CMakeModules` files
- system inspection:
  find_library , find_program, find_package,

A few words about the syntax

- CMake is a 'scripting language'; test it with
  `cmake -P script.cmake`
- it can define variables: `set(var value); message(STATUS ${var})`
- CMake has control statements: `foreach`, `if`, `while`, `function`
- support some data structures: `list`, `string`
- support for functions! beware of scope of variables relative to directory
- support modularity trough `CMakeModules` files
- system inspection:
  find_library , find_program, find_package,
- case insensitive, except for constant argument of commands

```
MESSAGE(STATUS "peekaboo")
```

Bonus: doing maths with CMake :-)

- we want to compute

$$\sum_{i=1}^{10} i^2 = \frac{(10+1)\cdot(2\cdot 10+1)\cdot 10}{6} = 385$$

Bonus: doing maths with CMake :-)

- we want to compute

$$\sum_{i=1}^{10} i^2 = \frac{(10+1)\cdot(2\cdot 10+1)\cdot 10}{6} = 385$$

- write the following `somme.cmake`

```
cmake_minimum_required(VERSION 2.8)
set(somme 0)
set(n $ENV{N})
foreach(i RANGE 1 ${n})
  math(EXPR somme "${somme}+${i}*${i}")
endforeach()
message(STATUS ${somme})
```

Bonus: doing maths with CMake :-)

- we want to compute

$$\sum_{i=1}^{10} i^2 = \frac{(10+1)\cdot(2\cdot 10+1)\cdot 10}{6} = 385$$

- write the following `somme.cmake`

```
cmake_minimum_required(VERSION 2.8)
set(somme 0)
set(n $ENV{N})
foreach(i RANGE 1 ${n})
  math(EXPR somme "${somme}+${i}*${i}")
endforeach()
message(STATUS ${somme})
```

- run CMake: `env N=10 cmake -P somme.cmake`

```
-- 385
```