Build system for the LCG software

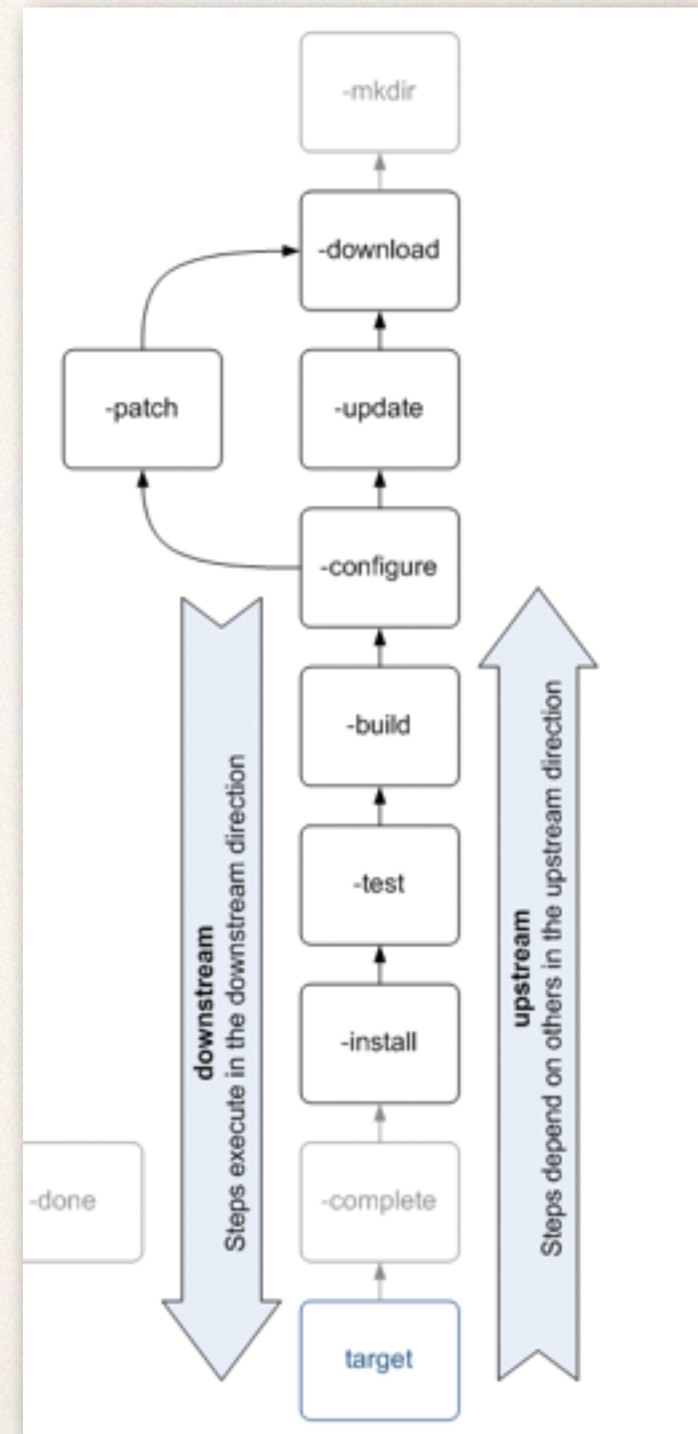
Benedikt Hegner, Pere Mato
HSF Packaging WG Meeting, 9th June 2015

LCG Configurations

- Configuring, building and deploying external libraries (~140) and MC Generators (~50) for all the supported platforms (~10) used by LHC experiments
  - Releasing full configurations. Content, versions and platforms discussed/agreed with experiments (LIM+AF)
  - We have been providing this service to the experiments successfully for the last N>10 years
  - Originally implemented with script-lets driven by CMT, now an implementation based on CMake
- With some special constrains, e.g. :
  - python packages installation into a reduced set of “wrapper packages”
  - several versions of same package (in particular for MC generators)
  - special naming conventions (package, platforms, …)
  - relocatability (e.g. easy moving installations from AFS to CVMFS)
  - …

CMake *ExternalProject* Module

- CMake comes with a standard module *ExternalProject* that creates custom targets to drive download, update/patch, configure, build, install and test steps of an external package
  - Fairly easy to add additional custom steps such as the creation of source and binary tarfiles, installation of logfiles, etc.
  - **Implemented a wrapper of *ExternalProject_Add*() to to inject all these extra features**
- CMake generates a Makefile (Ninja file) that at the end drives all the build process
  - *make -jN* works like a dream!



Example

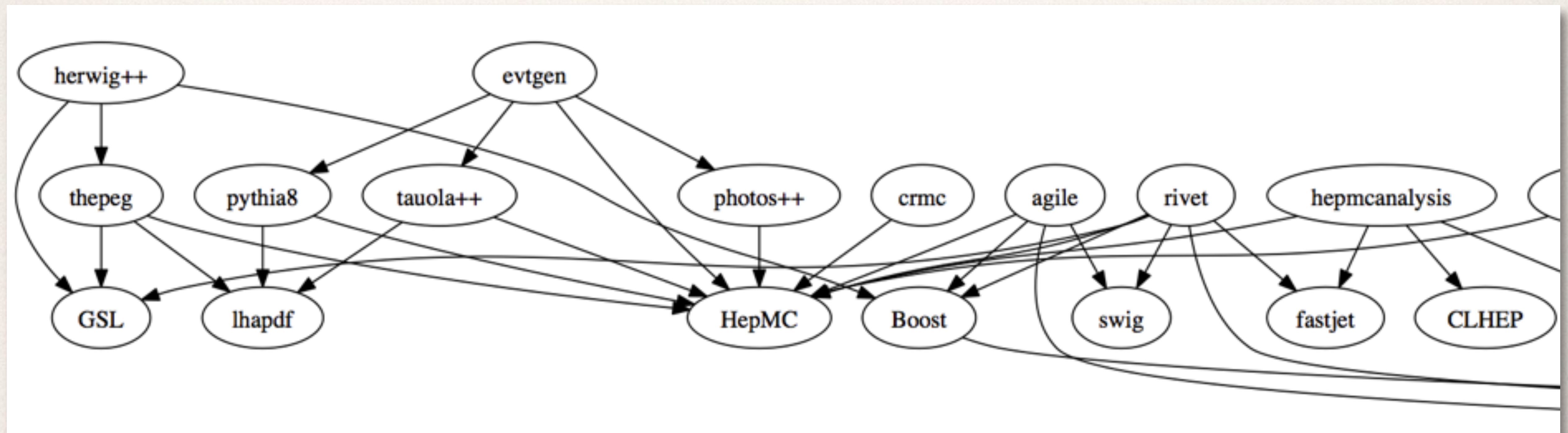
- Few lines are sufficient to describe the steps required for a given package
  - Dependencies to other packages are explicit
  - Variables such as ${XXX_home} point to the installation of package XXX

```
#---agile---------------------------------------------------------------------

LCGPackage_Add(
  agile
  URL http://www.hepforge.org/archive/agile/AGILe-${agile_native_version}.tar.bz2
  CONFIGURE_COMMAND ./configure --prefix=<INSTALL_DIR>
                                --with-hepmc=${HepMC_home}
                                --with-boost-incpath=${Boost_home_include}
                                --with-lcgtag=${LCG_platform}
                      PYTHON=${Python_home}/bin/python
                      LD_LIBRARY_PATH=${Python_home}/lib:$ENV{LD_LIBRARY_PATH}
                      SWIG=${swig_home}/bin/swig
  BUILD_COMMAND make all LD_LIBRARY_PATH=${Python_home}/lib:$ENV{LD_LIBRARY_PATH}
  INSTALL_COMMAND make install
                   LD_LIBRARY_PATH=${Python_home}/lib:$ENV{LD_LIBRARY_PATH}
  BUILD_IN_SOURCE 1
  DEPENDS HepMC Boost Python swig
)
```

Package Dependencies

- From the dependencies we can generate dependency graphs
  - Useful for documentation
  - Full package dependency versions for binary compatibility (hash number generation)



Defining the Configuration

- A single file lists all the packages and their required versions

```
# Application Area Projects
LCG_AA_project(COOL  COOL_2_8_17)
LCG_AA_project(CORAL CORAL_2_3_26)
LCG_AA_project(RELAX RELAX_1_3_0k)
LCG_AA_project(ROOT  5.34.05)
LCG_AA_project(LCGCMT LCGCMT_${heptools_version})

# Externals
LCG_external_package(4suite            1.0.2p1                                  )
LCG_external_package(AIDA              3.2.1                                    )
LCG_external_package(blas              20110419                                 )
LCG_external_package(Boost             1.50.0                                   )

...

# Generators
LCG_external_package(starlight         r43            MCGenerators/starlight    )
LCG_external_package(herwig            6.520          MCGenerators/herwig       )
LCG_external_package(herwig            6.520.2        MCGenerators/herwig       )
LCG_external_package(crmc              v3400          MCGenerators/crmc         )
LCG_external_package(cython            0.19           MCGenerators/cython       )
LCG_external_package(yaml_cpp          0.3.0          MCGenerators/yaml_cpp     )
LCG_external_package(yoda              1.0.0          MCGenerators/yoda         )
```

Build instructions are fairly simple

- get or setup **cmake**
- checkout **lcgcmake** package from SVN
- setup C/C++/Fortran compilers
- create workspace area
- configure with **cmake**
- build with **make**

1. On **lxplus** set PATH to use one of latest CMake versions (default is 2.6)
   `export PATH=/afs/cern.ch/sw/lcg/external/CMake/2.8.9/Linux-i386/bin:${PATH}`
2. Checkout the lcgcmake package from lcgsoft SVN repository
   `svn co svn+ssh://svn.cern.ch/reps/lcgsoft/trunk/lcgcmake`
3. Create a workspace area in which to perform the builds
   `mkdir lcgcmake-build`
   `cd lcgcmake-build`
4. You may need at this moment to define the compiler to use if different from the native compiler
   `source /afs/cern.ch/sw/lcg/external/gcc/version/platform/setup.(c)sh`
5. Configure the build of all externals with cmake
   `cmake -DCMAKE_INSTALL_PREFIX=../lcgcmake-install ../lcgcmake`
6. In order to build against the existing external repository use the option
   `-DLCG_INSTALL_PREFIX=/afs/cern.ch/sw/lcg/external`
   to tell the system to look for packages in the LCG area.
7. Build and install all external packages
   `make -j`
8. Or to build a single external package
   `make -j <package>` (use `make help`to see the list of all available packages)
9. You may need to restart de build of a package from beginning in case of obscure errors. The best is to clean a specific package
   `make clean-<package>`

http://ph-dep-sft.web.cern.ch/document/using-lcgcmake

Conditional Declarations

- Often we need to change the build instructions depending on the platform, version, etc.
  - Introduced ‘embedded conditional declarations’
- The example of ROOT is probably the most complicated one

```
LCGPackage_Add(
    ROOT
    IF <VERSION> MATCHES "^v.*-patches|HEAD" THEN
      GIT_REPOSITORY http://root.cern.ch/git/root.git GIT_TAG <VERSION>
      UPDATE_COMMAND <VOID>
    ELSE
      URL ftp://root.cern.ch/root/root_v${ROOT_author_version}.source.tar.gz
    ENDIF
    CMAKE_CACHE_ARGS -DCMAKE_PREFIX_PATH:STRING=${Python_home} ${Davix_home}
                     ${fftw_home} ${mysql_home} ${xrootd_home} ${graphviz_home}
                     ${GSL_home} ${Qt_home} ${CASTOR_home} ${dcap_home}
    CMAKE_ARGS -DCMAKE_BUILD_TYPE=${CMAKE_BUILD_TYPE}
               -DCMAKE_INSTALL_PREFIX=<INSTALL_DIR>
               -Dpython=ON
               -Dbuiltin_pcre=ON
               -Dcintex=ON
               IF DEFINED Davix_native_version THEN
                 -Ddavix=ON
               ENDIF
               -Dgdml=ON
               -Dgsl_shared=ON
               -Dkrb5=ON
               -Dgenvector=ON
               IF <VERSION> MATCHES "^v6-|^6[.]" THEN
                 -Dvc=ON
               ENDIF
              ...
               IF LCG_CPP11 THEN
                   -Dcxx11=ON
               ENDIF
               IF LCG_TARGET MATCHES x86_64-slc THEN
                   -Dcastor=ON
                   -Ddcache=ON
                   -Dgfal=ON -DGFAL_DIR=${gfal_home}
                             -DSRM_IFCE_DIR=${srm_ifce_home}
               ENDIF
               IF LCG_TARGET MATCHES slc THEN
                   -Doracle=ON -DORACLE_HOME=${oracle_home}
                   -Dqt=ON
               ENDIF
   DEPENDS Python fftw graphviz GSL mysql xrootd
            IF DEFINED Davix_native_version THEN
                Davix
            ENDIF
            IF LCG_TARGET MATCHES x86_64-slc THEN
                CASTOR dcap gfal srm_ifce
            ENDIF
            IF LCG_TARGET MATCHES slc THEN
                oracle Qt
            ENDIF
)
```

Incremental Builds

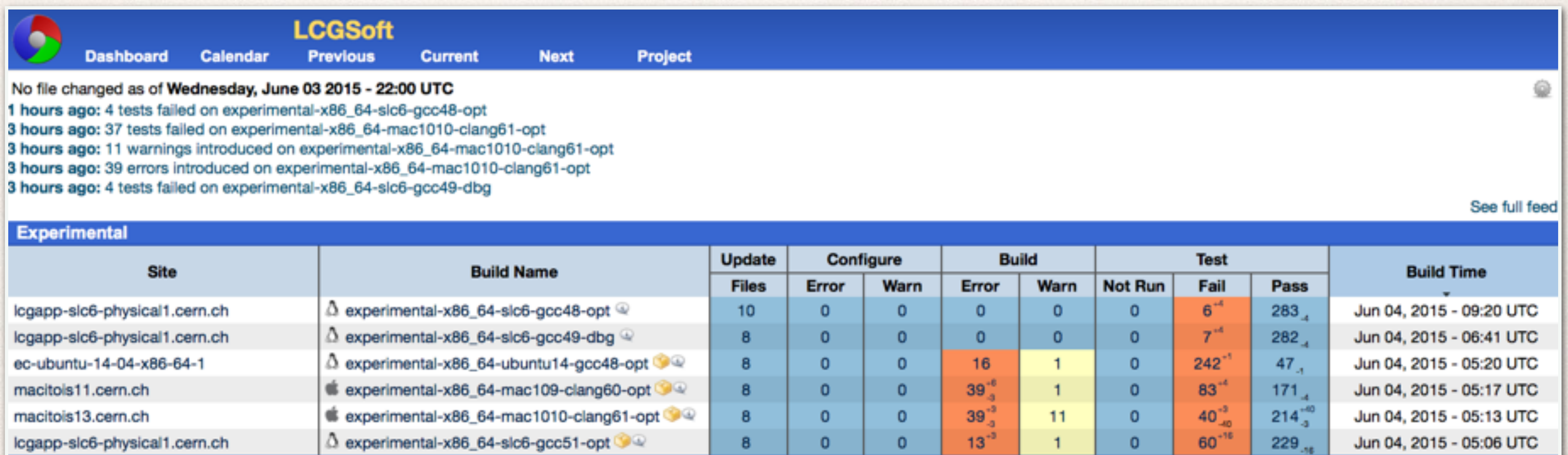
- Package binaries are installed in:
  - <prefix>/<package>/<version>_<hash>/<platform_tag>/…
    - The <platform_tag> is a combination of processor architecture, os version, compiler version and build type (e.g. x86_64-slc6-gcc48-dbg, aarch64-ubuntu14-gcc49-opt)
    - The <hash> value is calculated taking into account the full list of package dependencies and their versions
- When building a package, the user can tell the system to take existing builds from a given <prefix>
  - The match will take into account <version>_<hash>/<platform_tag>
  - The actual 'target' build will consists of creating a soft-link to the existing installation

Runtime Environment

- During the build, files for providing the runtime environment setting will be generated
  - By default, /lib[64] will go to LD_LIBRARY_PATH, /bin will go to PATH, etc.
  - **<package>-env.sh** is generated for each package, which executes similar scripts for the dependent packages
  - This is an area that will be improved in the next few weeks with custom variables

Nightly Integration

- The full software stack can be built regularly on several configurations and all supported platforms/buildtypes and tests run
- Easy integration with Jenkins (scheduler) and CDash (dashboard)



Site	Build Name	Update	Configure		Build		Test			Build Time
		Files	Error	Warn	Error	Warn	Not Run	Fail	Pass	
lcgapp-slc6-physical1.cern.ch	experimental-x86_64-slc6-gcc48-opt	10	0	0	0	0	0	6	283	Jun 04, 2015 - 09:20 UTC
lcgapp-slc6-physical1.cern.ch	experimental-x86_64-slc6-gcc49-dbg	8	0	0	0	0	0	7	282	Jun 04, 2015 - 06:41 UTC
ec-ubuntu-14-04-x86-64-1	experimental-x86_64-ubuntu14-gcc48-opt	8	0	0	16	1	0	242	47	Jun 04, 2015 - 05:20 UTC
macitois11.cern.ch	experimental-x86_64-mac109-clang60-opt	8	0	0	39	1	0	83	171	Jun 04, 2015 - 05:17 UTC
macitois13.cern.ch	experimental-x86_64-mac1010-clang61-opt	8	0	0	39	11	0	40	214	Jun 04, 2015 - 05:13 UTC
lcgapp-slc6-physical1.cern.ch	experimental-x86_64-slc6-gcc51-opt	8	0	0	13	1	0	60	229	Jun 04, 2015 - 05:06 UTC

Comparing with Worch

Worch Overview

Worch = **W**af + **Orch**estration:

- **software suite builder** used to build large suites of software composed of many packages from all different sources.
- **configuration manager** using a simple declarative language in order to precisely and concisely assert all build parameters.
- **workflow manager** using Waf to run interdependent tasks in parallel
- **software build features** "batteries included" for exercising many common package-level build methods
- **bootstrap aggregation** packaged using Python's `setuputils` with support for developing domain-specific extensions to easily create the build environment.
- **policy-free** leaving issues such as installation layout, target package formats, suite content, build parameters up to the end user.

Yes, the same

Very similar, declarative and short, specially for 'standard' packages

Yes, as good as make -j (ninja)

ExternalProject comes with their batteries

Bootstrap very simple requirements: cmake, svn/git, make

Policies encoded in CMake code easy to change :-)

Comparing with Homebrew

Why (Not) Homebrew?

- Works out the box on Mac and Linux
- *Extremely* easy to use and add new packages
- Good support for build variants and C++ Standards
- Only provides a single rolling release
- Doesn't directly support git tags or rollback on versions
- Binary packages not completely relocatable(*)

Yes, and also Windows?

Similar or perhaps even simpler

Yes, using CMAKE_BUILD_TYPE and CXX/C global flags

Many many concurrent releases are supported

Yes

Yes

Conclusions

- Very simple setup and instructions, adding a new package is really trivial
  - ~ 13 <LOC>/package (including comments and blank lines)
- Many concurrent configurations, many platforms supported
- Very easy customizable
  - New build steps (e.g. installation of log files, RPM creation, etc.) can be added very easily and applicable to all 150 packages
  - All customizations and policies in ~800 lines of CMake code
- Results are relocatable, and be installed in several ways appreciate to the users