The WTCHG Research Computing Core

Talk 6: Doing your own thing (compiling and customizing)

Robert Esnouf (robert@strubi.ox.ac.uk; robert@well.ox.ac.uk),

Jon Diprose (jon@well.ox.ac.uk) & Colin Freeman (cfreeman@well.ox.ac.uk)

Generic emails: support rescomp@well.ox.ac.uk

users rescomp-users@well.ox.ac.uk

A series of introductory talks…

A set of six talks of about 45 minutes each (plus questions)

Talk 1: What is the ResComp Core?
(Mon 23/1 10:00 Room B; Robert Esnouf)

Talk 2: A basic introduction to Linux
(Wed 25/1 10:00 Room B; Robert Esnouf)

Talk 3: Submitting jobs to the cluster
(Thu 26/1 10:00 Room B; Robert Esnouf)

Talk 4: Monitoring and troubleshooting
(Mon 30/1 11:00 Room B; Robert Esnouf)

Talk 5: ResComp centrally-managed applications
(Wed 1/2 10:00 Room B; Jon Diprose)

Talk 6: Doing your own thing (compiling and customizing)
(Thu 2/2 10:00 Room B; Jon Diprose)

The story so far…

The ResComp cluster has different types of node:

- 288 "A" cores @ 2.67GB/core; 640 "B" cores @ 8GB/core; 1720 "C" cores & 768 "D" cores @ 16GB/core; 144 "H" cores @ 42.66GB/core
- Login/submission nodes: rescomp[1-2] and the project servers

Files must be on the (chargeable) GPFS filesystems:

- /gpfs0: /users/*<group>*/*<user>*, /apps/well, /mgmt
- /gpfs1 & /gpfs2: /well/*<group>*

Refresher of basic Linux concepts

- User accounts, groups, commands, shells (bash)
- Environments and environment variables: $PATH and $LD_LIBRARY_PATH
- File streams (stdin, stdout and stderr) redirection & pipes ("<", ">", "|")
- Foreground and background jobs ("&"), "nohup" and "screen"
- Writing, executing and sourcing a shell script

The story so far…

Cluster computing is non-graphical and non-interactive!

- Sun Grid Engine (SGE) using a group-based share tree policy

Your job is to submit jobs, our job is to ensure that they start

- Don't wait for free slots, don't expect jobs to start immediately!

Submit scripts to the SGE scheduler using "qsub"

- Embed "qsub" arguments using "#$" lines
- Specify where stdout and stderr should go
- "#$ -pe shmem|mpi <*n*>" for multicore or MPI jobs
- "qalter", "qrls", "qmod", "qdel", "qstat", "qsum", "qacct", "qload" & "qconf"

Different queues: typically short (<24 hours) & long (<7 days)

- Jobs killed if they exceed time or memory limits
- Special queues for relion jobs and on project servers

Centrally-managed software suite

- /apps/well, /apps/strubi and /apps/htseq
- PATH environment variable, "module avail", "which"

Overview of this talk…

Source code

The compilation process

Installing an application

Common build frameworks

Cython and the like

Diagnosing failures

Source code

Source code is a set of 'human-readable' instructions

The source code is written in a programming language

- C, C++, Fortran, Java, Python, etc.

Choice of language can be based of a number of factors

- knowledge, experience, availability of help
- compatibility with a library that already does the hard work
- functionality
- performance vs safety
  - low-level vs high-level
- maintainability vs speed of development
  - strongly-typed vs weakly-typed
  - compiled vs interpreted

Source code

Programmers are lazy

- they want to re-use existing code

Code gets split up, into

- macros
- functions / classes
- multiple source code files
- header files
  - contain declarations of macros and functions / classes that can be used in or called from other source code files
- libraries
  - compiled code that can be incorporated into or used by other complied code

The Compilation Process

The source code must be 'compiled' into something the computer understands before it can be run

This is done by an application called a 'compiler'

There are several different compilers available on rescomp

- gcc
  - 4.4.7 – default version
  - 4.7.2 – after `scl enable devtoolset-1.1 bash`
  - 4.8.2 – after `scl enable devtoolset-2 bash`
  - 4.9.1 – after `scl enable devtoolset-3 bash`
    - Beware – this appears to have an old version of the openmp library
  - 4.9.3 – after `module load gcc/4.9.3`
  - 5.4.0 – after `module load gcc/5.4.0`
- intel
- llvm/clang

Simple Case

```
gcc -o myapp myapp.c
```

Compiles the binary `myapp` from myapp.c

Compilation Steps

http://www.thegeekstuff.com/2011/10/c-program-to-an-executable/

Pre-processor

- expands included files, replaces macros and variable names with their values, removes comments
- produces source code

Compilation

- compiles source code to assembler code

Assembly

- compiles assembler code to object (aka machine) code

Linker

- adds locations of any functions that have been referenced from external libraries plus standard startup and shutdown code
- produces the executable binary

Dependencies and RUNPATH

To run a dynamic binary with a dependency on a shared library, the shared library must be installed somewhere the binary can find it

- shared library often available from the system's package manager
- often installed into a system-standard location
- can use **`LD_LIBRARY_PATH`** to point to non-standard location
- usually located in a directory called "lib" or "lib64"

To compile the application, the header files that describe the functions available from the shared library must also be installed

- *-devel packages
- usually located in a directory called "include"

Dependencies and RUNPATH

The compiler has standard sets of directories to search to find files to be included and libraries to be linked

- if required header files and libraries are not in these standard locations you will have to add extra directories to these lists

The **`–I`** argument (pre-processor phase)

- adds directories to the list for files to be included

The **`–l`** argument (linker phase)

- specifies the name of the library
- usually without the leading 'lib' and trailing '.so'

The **`–L`** argument (linker phase)

- adds directories to the list for libraries to be linked

Dependencies and RUNPATH

The location of the library can be compiled into the binary

- adds to the list of paths the application will search automatically to find the shared library
- **RPATH** – cannot be overridden by **LD_LIBRARY_PATH**
- **RUNPATH** – can be overridden by **LD_LIBRARY_PATH**

Set during the linker phase

- **-rpath** argument sets **RPATH**
- **--enable-new-dtags** modifier sets **RUNPATH**

Setting these correctly makes the application easier to run

- not necessary to set **LD_LIBRARY_PATH** at runtime

More Complex Case

```
HDF5_DIR=/apps/well/hdf5/1.8.12-gcc4.7.2
gcc -c -o function1.o function1.c
gcc -c -o function2.o \
    -I${HDF5_DIR}/include \
    function2.c
gcc -c -o main.o main.c
gcc -o myapp \
    -L${HDF5_DIR}/lib \
    -Wl,-rpath,${HDF5_DIR}/lib,--enable-new-dtags \
    -lhdf5 \
    function1.o function2.o main.o
```

Compiles \`myapp\` from function1.c, function2.c & main.c

Code uses functions from an HDF5 library ‘libhdf5.so’

- installed under /apps/well/hdf5/1.8.2-gcc4.7.2

More Complex Case

```
HDF5_DIR=/apps/well/hdf5/1.8.12-gcc4.7.2
gcc -c -o function1.o function1.c
gcc -c -o function2.o \
    -I${HDF5_DIR}/include \
    function2.c
gcc -c -o main.o main.c
gcc -o myapp \
    -L${HDF5_DIR}/lib \
    -Wl,-rpath,${HDF5_DIR}/lib,--enable-new-dtags \
    -lhdf5 \
    function1.o function2.o main.o
```

Where to find the HDF5 headers

More Complex Case

```
HDF5_DIR=/apps/well/hdf5/1.8.12-gcc4.7.2
gcc -c -o function1.o function1.c
gcc -c -o function2.o \
    -I${HDF5_DIR}/include \
    function2.c
gcc -c -o main.o main.c
gcc -o myapp \
    -L${HDF5_DIR}/lib \
    -Wl,-rpath,${HDF5_DIR}/lib,--enable-new-dtags \
    -lhdf5 \
    function1.o function2.o main.o
```

Add the directory containing the HDF5 dynamic library to the search path and link to it

More Complex Case

```
HDF5_DIR=/apps/well/hdf5/1.8.12-gcc4.7.2
gcc -c -o function1.o function1.c
gcc -c -o function2.o \
    -I${HDF5_DIR}/include \
    function2.c
gcc -c -o main.o main.c
gcc -o myapp \
    -L${HDF5_DIR}/lib \
    -Wl,-rpath,${HDF5_DIR}/lib,--enable-new-dtags \
    -lhdf5 \
    function1.o function2.o main.o
```

Add the directory containing the HDF5 library to the run-time search path, allowing it to be overridden by **`LD_LIBRARY_PATH`**

Installing an Application

Can be installed anywhere

- anywhere you have write permission, that is…
- including where you have just built it
- /users or /well
- need to set your environment so that things it needs to find and that need to find it can do so
  - `PATH` environment variable for applications
    - or use fully specified path
  - `LD_LIBRARY_PATH` environment variable for dynamic libraries

Can sometimes be security measures that prevent applications being run from certain directories

- rescomp cluster not set up like that

Common build frameworks

As application size and complexity grows, so does the compilation process

Frameworks created to simplify the compilation process

- perform the various compilation steps as necessary
- in the right order
- referencing the correct external libraries
- can (should) include any steps necessary for installation

Many frameworks – most common are

- make
- autotools
  - ./configure
- cmake

Make

Compilation steps described in makefiles

- usually named Makefile
- series of make ‘targets’
- each target knows what it depends on
- text file that is sort-of human-readable

By convention:

- the default target will cause the application to be compiled
- the install target will cause it to be installed
- the clean target will delete all the files generated during the compilation if you need to start again

```
make
make install
```

Make

By convention, customized settings can be passed into the makefile via certain variables

- can be environment variables or arguments to `make`
- useful for libraries installed in non-standard locations
- **`CFLAGS`** – for settings relevant to C compilation steps other than linking
- **`CXXFLAGS`** – the same for C++ compilation
- **`LDFLAGS`** – for settings relevant to the linking step
- **`PREFIX`** – where the application will be installed

Make

```
HDF5_DIR=/apps/well/hdf5/1.8.12-gcc4.7.2
export CFLAGS=-I${HDF5_DIR}/include
export LDFLAGS="-L${HDF5_DIR}/lib \
  -Wl,-rpath,${HDF5_DIR}/lib,--enable-new-dtags"
export PREFIX=/apps/well/myapp/1.0

make
make install
```

Autotools

Customizing make builds via variables not enough

Autotools provides a framework for customizing the makefiles

- created to provide reliable builds across different unix variants
- discovers what is on the system
- generates a makefile from a template
- customization done by a script called `configure`
- configure has a standard set of arguments
  - --help
  - --prefix
- but can itself be configured to have more
- still uses make so same variables as before still work

Autotools

```
HDF5_DIR=/apps/well/hdf5/1.8.12-gcc4.7.2
export CFLAGS=-I${HDF5_DIR}/include
export LDFLAGS="-L${HDF5_DIR}/lib \
  -Wl,-rpath,${HDF5_DIR}/lib,--enable-new-dtags"

./configure –prefix=/apps/well/myapp/1.0
make
make install
```

Autotools

```
./configure --prefix=/apps/well/myapp/1.0 \
            --with-hdf5=/apps/well/hdf5/1.8.12-gcc4.7.2
make
make install
```

If the developer has been nice…

CMake

Autotools isn't the best cross-platform framework

- doesn't work reliably for Windows

CMake is better

- does much the same as autotools
- generates makefiles (or the equivalents for platforms with no make)
- still uses make to do the build
- different syntax for passing arguments
  - `CFLAGS => -DCMAKE_C_FLAGS`
  - `CXXFLAGS => -DCMAKE_CXX_FLAGS`
  - `LDFLAGS => -DCMAKE_EXE_LINKER_FLAGS`
  - `PREFIX => -DCMAKE_INSTALL_PREFIX`
  - again, additional customizations possible

CMake

```
HDF5_DIR=/apps/well/hdf5/1.8.12-gcc4.7.2
/apps/well/cmake/2.8.12.2/bin/cmake . \
  -DCMAKE_C_FLAGS=-I${HDF5_DIR}/include \
  -DCMAKE_EXE_LINKER_FLAGS="-L${HDF5_DIR}/lib \
     -Wl,-rpath,${HDF5_DIR}/lib,--enable-new-dtags" \
  -DCMAKE_INSTALL_PREFIX=/apps/well/myapp/1.0
make
make install
```

CMake

```
/apps/well/cmake/2.8.12.2/bin/cmake . \
  -DCMAKE_HDF5DIR=/apps/well/hdf5/1.8.12-gcc4.7.2 \
  -DCMAKE_INSTALL_PREFIX=/apps/well/myapp/1.0
make
make install
```

If the developer has been nice…

Cython and the like

It is common for python packages to have a component written in C/C++ for performance reasons

- also true of R and perl

Generally use `make` to control the build

- so if it is necessary to customize the build one can just set the relevant environment variables

```
BDB_DIR=/apps/well/bdb/4.8.30
module load python/2.7
CFLAGS=-I${BDB_DIR}/include \
  LDFLAGS="-L${BDB_DIR}/lib \
    -Wl,-rpath,${BDB_DIR}/lib,--enable-new-dtags" \
  pip install wormtable
module unload python
```

Diagnosing Failures

First question is in which phase of the build the failure has happened

- might have to look back a few lines

./configure or cmake

- most likely to be a missing dependency
- possibly needs newer version of framework or compiler

Compilation phase – missing header file

- ask for relevant devel package to be installed
- pass in missing **`-I`** argument using **`CFLAGS/CXXFLAGS`**

Compilation phase – code error

- possibly wrong version of a dependency
- check for fixes to or newer version of source

Diagnosing Failures

Linker phase – missing symbol

- possibly wrong version of a dependency
- sometimes fixable by a `make clean` and trying again

Linker phase – missing library

- ask for relevant packages to be installed
- pass in missing **`–L`** argument using **`LDFLAGS`**
- set RUNPATH as well ('**`–Wl,-rpath,…`**')

Installation phase – not in the sudoers file

- you tried to do `sudo make install` and aren't allowed to
- don't use sudo
- set **`PREFIX`** to install somewhere you have write permission

Installation phase – permission denied

- you tried to install to somewhere you don't have write permission
- set **`PREFIX`** to install somewhere you have write permission

Diagnosing Failures

The command `readelf` allows you to see a dynamic binary's dependencies and RPATH / RUNPATH settings

- readelf –d /path/to/binary

The command `ldd` allows you to see how a dynamic binary's dependencies will get resolved

- ldd /path/to/binary
- libraries reported as 'not found' are of concern
  - have you forgotten a necessary `module load …`?
  - if it is a library, path could be specified by the binary it is loaded by
    - e.g. libpython path will come from python for cython libraries
  - rebuild setting RUNPATH appropriately
  - set **LD_LIBRARY_PATH** appropriately

Example

```
wget
ftp://ftp.broadinstitute.org/pub/crd/DiscovarExp/latest_source_code/disco
varexp-51875.tar.gz
tar xf discovarexp-51875.tar.gz
cd discovarexp-51875
./configure --prefix=/apps/well/discovarexp/51875
```

```
...
Failed...
configure: Configure failed with the error message: You must compile this
with g++ 4.7 or higher.
configure:
configure: Common error conditions in the build process are documented
configure: on our wiki page:
configure: http://www.broadinstitute.org/crd/wiki/index.php/Build_FAQ
configure: We also offer email support at crdhelp@broadinstitute.org
configure: error: in `/home/software/discovarexp-51875':
configure: error: Configure failed.
See `config.log' for more details.
```

Example

```
scl enable devtoolset-1.1 bash
./configure --prefix=/apps/well/discovarexp/51875
make
```

```
...
/bin/sh ../libtool --tag=CXX   --mode=link g++ -pthread -fopenmp -g -O2 -
std=c++11 -Wextra -Wall -Wsign-promo -Woverloaded-virtual -Wendif-labels
-Wno-unused -Wno-deprecated -Wno-long-long -Wno-parentheses -Wno-unused-
parameter -fno-nonansi-builtins -mieee-fp -fno-strict-aliasing -iquote .
-ggdb -DNDEBUG   -o DiscovarExp DiscovarExp.o libDiscovarExp.a -lz -
ljemalloc
libtool: link: g++ -pthread -fopenmp -g -O2 -std=c++11 -Wextra -Wall -
Wsign-promo -Woverloaded-virtual -Wendif-labels -Wno-unused -Wno-
deprecated -Wno-long-long -Wno-parentheses -Wno-unused-parameter -fno-
nonansi-builtins -mieee-fp -fno-strict-aliasing -iquote . -ggdb -DNDEBUG
-o DiscovarExp DiscovarExp.o  libDiscovarExp.a -lz -ljemalloc -pthread
/opt/centos/devtoolset-1.1/root/usr/libexec/gcc/x86_64-redhat-
linux/4.7.2/ld: cannot find -ljemalloc
collect2: error: ld returned 1 exit status
make[1]: *** [DiscovarExp] Error 1
make[1]: Leaving directory `/home/software/discovarexp-51875/src'
make: *** [all-recursive] Error 1
```

Example

```
./configure --help | grep jemalloc
```

```
 --with-jemalloc=DIR     Force directory for location of jemalloc
library.
```

Example

```
./configure --prefix=/apps/well/discovarexp/51875 --with-
jemalloc=/apps/well/jemalloc/3.6.0-gcc4.7.2/lib
make
sudo make install
exit
cd ..
rm -rf discovarexp-51875
ldd /apps/well/discovarexp/51875/bin/
```

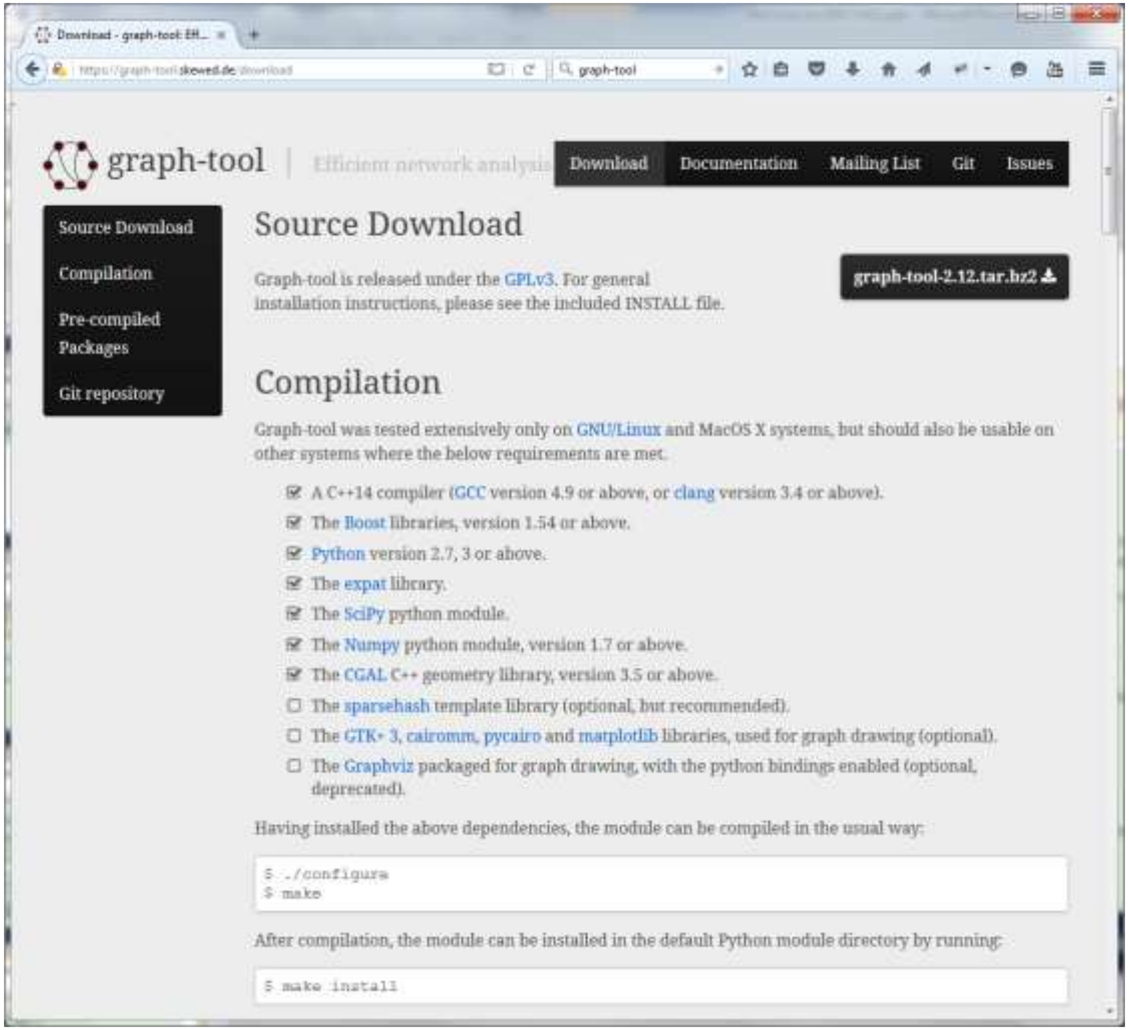
```
ldd /apps/well/discovarexp/51875/bin/DiscovarExp
        linux-vdso.so.1 =>  (0x00007fff0c57f000)
        libz.so.1 => /usr/lib64/libz.so.1 (0x0000003d69e00000)
        libjemalloc.so.1 => /apps/well/jemalloc/3.6.0-
gcc4.7.2/lib/libjemalloc.so.1 (0x00007f1836476000)
        libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x00000031ad200000)
        libm.so.6 => /lib64/libm.so.6 (0x0000003d69200000)
        libgomp.so.1 => /usr/lib64/libgomp.so.1 (0x0000003d6aa00000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x00000031ace00000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x0000003d69a00000)
        libc.so.6 => /lib64/libc.so.6 (0x0000003d68e00000)
        /lib64/ld-linux-x86-64.so.2 (0x0000003d68a00000)
        librt.so.1 => /lib64/librt.so.1 (0x0000003d6a200000)
```

Example

```
/apps/well/discovarexp/51875/bin/DiscovarExp -?
```

```
Performing re-exec to adjust stack size.
i = 1, argv[i] = -?

Fatal error at Thu Jan 29 11:48:00 2015: You did not invoke this program
with white-space-free arguments of the form PARAMETER=VALUE.

To see the syntax for this command, type "DiscovarExp -h".

Invalid input detected.


Thu Jan 29 11:48:00 2015.  Abort.  Stopping.

Generating a backtrace...

Dump of stack:

0. CRD::exit(int), in Exit.cc:30
1. parsed_args::parsed_args(int, char**, unsigned char), in
ParsedArgs.cc:148
2. main, in DiscovarExp.cc:22
```

Example



Source Download

Graph-tool is released under the GPLv3. For general installation instructions, please see the included INSTALL file.

Compilation

Graph-tool was tested extensively only on GNU/Linux and MacOS X systems, but should also be usable on other systems where the below requirements are met.

- ☑ A C++14 compiler (GCC version 4.9 or above, or clang version 3.4 or above).
- ☑ The Boost libraries, version 1.54 or above.
- ☑ Python version 2.7, 3 or above.
- ☑ The expat library.
- ☑ The SciPy python module.
- ☑ The Numpy python module, version 1.7 or above.
- ☑ The CGAL C++ geometry library, version 3.5 or above.
- ☐ The sparsehash template library (optional, but recommended).
- ☐ The GTK+ 3, cairomm, pycairo and matplotlib libraries, used for graph drawing (optional).
- ☐ The Graphviz packaged for graph drawing, with the python bindings enabled (optional, deprecated).

Having installed the above dependencies, the module can be compiled in the usual way:

```
$ ./configure
$ make
```

After compilation, the module can be installed in the default Python module directory by running:

```
$ make install
```

That's all, folks!

Copies of the talks are available on the Centre's wiki:

http://wiki.well.ox.ac.uk/mediawiki/index.php/Introductory_Talks

[NB – accessible only from the Centre's network or vpn]

If any questions do come up later, please email:

rescomp@well.ox.ac.uk