Using Python on Beagle2

To use python/2.7.6-vanilla with corresponding modules:

```bash
ams@login1:~> module load python/2.7.6-vanilla
Switching to GNU compiler environment
Python version 2.7.6-vanilla loaded
ams@login1:~> python
Python 2.7.6 (default, Apr 23 2014, 15:05:05)
[GCC 4.8.1 20130531 (Cray Inc.)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import matplotlib
>>> import scipy
>>> import numpy
>>> import _sqlite3
>>> import vcf
>>> import pysam
>>> import cython
>>> import blist
>>> import sqlalchemy
>>> import bio
>>> import pymongo
>>> import pg
>>> import psycopg2
>>> import nose
>>> import PIL
>>> import networkx

With this version of python Plink will be in your path:

ams@login1:~> plink
```

@----------------------------------------------------------@
|        PLINK!       |     v1.07      |   10/Aug/2009     |
@----------------------------------------------------------@
|  (C) 2009 Shaun Purcell, GNU General Public License, v2 |
@----------------------------------------------------------@

Web-based version check ( --noweb to skip )
Connecting to web...  OK, v1.07 is current
Writing this text to log file [ plink.log ]
Analysis started: Mon Apr 28 12:28:43 2014

Options in effect:

Before frequency and genotyping pruning, there are 0 SNPs
0 founders and 0 non-founders found
0 SNPs failed missingness test ( GENO > 1 )
0 SNPs failed frequency test ( MAF < 0 )
After frequency and genotyping pruning, there are 0 SNPs

ERROR: Stopping as there are no SNPs left for analysis

run_grit.py and it is automatically added to your path when you load this version of python.

If you decide to use Python version 2.7.3 with corresponding modules: NumPy, SciPy, Matplotlib and MPI4Py,
just loading python 2.7.3 will not load these modules. To use them please set the following paths:
export PYTHONPATH=${PYTHONPATH}:/soft/python/2.7/2.7.3/modules/numpy/1.7.0/lib/python2.7/site-packages
export PYTHONPATH=${PYTHONPATH}:/soft/python/2.7/2.7.3/modules/scipy/0.12.0/lib/python2.7/site-packages
export PYTHONPATH=${PYTHONPATH}:/soft/python/2.7/2.7.3/modules/matplotlib/1.2.1/lib/python2.7/site-packages
export PYTHONPATH=${PYTHONPATH}:/soft/python/2.7/2.7.3/modules/mpi4py/1.3.1/lib/python2.7/site-packages

Python version 2.7.1 is the default. To use it type module load python. The load command will automatically change compiler from Cray to GNU (therefore care must be paid that this is compatible with the calculations being done).

To use Numpy and Scipy, type the following instruction before starting python module load scipy, they will automatically switch from python 2.6 to python 2.7.1 (and therefore also switch from Cray to GNU) and load these two modules.

Using IPython on Beagle2

ams@login1:~/julia> module swap python/3.3.0-vanilla python/2.7.3-vanilla
Python version 2.7.3-vanilla loaded
Python version 3.3.0-vanilla unloaded
ams@login1:~/julia> export PATH=$PATH:/soft/python/2.7.3-vanilla/virtualenvs/ober/bin/ipython
ams@login1:~/julia> ipython
WARNING: IPython History requires SQLite, your history will not be saved
Python 2.7.3 (default, Oct 19 2012, 15:25:23)
Type "copyright", "credits" or "license" for more information.
IPython 0.13.1 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help  -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

In [1]:

To use Matplotlib inside of IPython login on Beagle via: ssh -X <user_name>@login.beagle.ci.uchicago.edu

ams@login4:~> module load python/2.7.3-vanilla

ams@login4:~> ipython --pylab

Using python sqlite3 module

ams@login1:~> module load python/2.7.6-vanilla
Python version 2.7.6-vanilla loaded
ams@login1:~> python
Python 2.7.6 (default, Jun 13 2014, 14:44:50)
[GCC 4.8.1 20130531 (Cray Inc.)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import _sqlite3
>>> _sqlite3.version
'2.6.0'

If you need to use python 2.7.3 you would need to upgrade sqlite3 in python 2.7.3 (your installation) inside a virtualenv.

In other words you would need to have your python installation on /lustre/beagle2/myfolder

in my case that is in:

/lustre/beagle2/python

Before you start your installation make sure to load the proper environment:

module swap PrgEnv-cray PrgEnv-gnu

This will enable the C compiler to find the library and header files (remember that Cray compiler is default on Beagle2). After this step export paths to your installation of python, for me that was:

export PYTHONPATH=${PYTHONPATH}:/lustre/beagle2/python/lib/python2.7/site-packages
export PATH=/lustre/beagle2/python/bin:${PATH}
export LD_LIBRARY_PATH=/lustre/beagle2/python/lib:$LD_LIBRARY_PATH

Install virtualenv in your directory on lustre:
After this you will make sqlite dir inside of your virtualenv dir for me that is in /lustre/beagle2/ams/virtual/virtualenv-1.11.4/sqlite

and install the latest sqlite3 also make sure that you're installing the development package:

<verbatim>$ wget --no-check-certificate https://sqlite.org/2014/sqlite-autoconf-3080401.tar.gz</verbatim>
$ tar -xvf sqlite-autoconf-3080401.tar.gz
$ cd sqlite-autoconf-3080401
$ ./configure --prefix=/lustre/beagle2/ams/virtual/virtualenv-1.11.4
$ make
$ make install

Make sure that the sqlite lib dir is in ld_library_path, in my case that was:

export LD_LIBRARY_PATH=/lustre/beagle2/ams/virtual/virtualenv-1.11.4/sqlite/lib:$LD_LIBRARY_PATH

Once you have successfully installed the dependencies, you may proceed with the installation of pysqlite itself. Make a pysqlite dir inside of virtualenv dir and copy sqlite3.c inside it, for me that file was located in:

/lustre/beagle2/ams/virtual/virtualenv-1.11.4/sqlite/sqlite-autoconf-3080401/sqlite3.c

Next install your pysqlite inside of virtualenv dir. First get the latest version of pysqlite in usual way and compile pysqlite from source and together with the latest sqlite database with:

python setup.py build_static

Now go to your pysqlite dir, for me that was /lustre/beagle2/ams/virtual/virtualenv-1.11.4/pysqlite/pysqlite-2.6.3

and do:

python setup.py install

The above will actually install the pysqlite into path-to-virtualenv. During this step, the setup script moves the pysqlite2 package (including the newly compiled C extension) to the standard package directory of your Python installation so that Python will be able to import pysqlite2.dbapi2 and pysqlite2.test.

Now you have the latest version of sqlite (compiled into pysqlite) installed within your virtualenv, so you can test it:

ams@login1:/lustre/beagle2/ams/virtual/virtualenv-1.11.4/pysqlite/pysqlite-2.6.3> python
Python 2.7.3 (default, Apr 30 2013, 13:34:55)
[GCC 4.7.2 20130920 (Cray Inc.)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from pysqlite2 import dbapi2 as sqlite

Building and using relax library on Beagle2:

Please always do all installation on Lustre file system!

#loaded version of python and libraries I want to use:

module load python/2.7.3
export PYTHONPATH=/soft/python/2.7.3/python/2.7.2.7.3/python/bin/:$PATH
export LD_LIBRARY_PATH=/soft/python/2.7.3/python/lib:$LD_LIBRARY_PATH
export PYTHONPATH=/soft/python/2.7.2.7.3/modules/numpy/1.7.0/lib/python2.7/site-packages
export PYTHONPATH=/soft/python/2.7.2.7.3/modules/scipy/0.12.0/lib/python2.7/site-packages
export PYTHONPATH=/soft/python/2.7.2.7.3/modules/mpi4py/1.3/lib/python2.7/site-packages

#install virtualenv

wget https://pypi.python.org/packages/source/v/virtualenv/virtualenv-12.1.1.tar.gz
#md5=901ecbf302f5de9f9db31d84290b7217
tar -xvf virtualenv-12.1.1.tar.gz
cd virtualenv-12.1.1
python virtualenv.py myVE
virtualenv ENV
Building Python 2.7.6 on Beagle2

Before you start installation check your environment:

```
if [ ! $(module list -t 2>&1 | grep PrgEnv-gnu) ]; then
    module swap $(module list -t 2>&1 | grep PrgEnv) PrgEnv-gnu
fi
```

Python installation

```bash
pyvers=2.7.6
pydir=/soft/python/2.7/{pyvers}-vanilla
mkdir -p ${pydir}

— Vanilla settings: replace CC with gcc
myCC=gcc
myCXX=g++

—Build sqlite3
wget http://www.sqlite.org/2014/sqlite-autoconf-3080403.tar.gz
tar -xvf sqlite-autoconf-3080403.tar.gz
cd sqlite-autoconf-3080403
sqlitedir=/soft/python/2.7/sqlite3
mkdir -p ${sqlitedir}
./configure CC=${myCC} cc=${myCC} -prefix=${sqlitedir}
maker
make install
```

---

**Building Python 2.7.6 on Beagle2**

Before you start installation check your environment:

```bash
if [ ! $(module list -t 2>&1 | grep PrgEnv-gnu) ]; then
    module swap $(module list -t 2>&1 | grep PrgEnv) PrgEnv-gnu
fi
```

Python installation

```bash
pyvers=2.7.6
pydir=/soft/python/2.7/{pyvers}-vanilla
mkdir -p ${pydir}

— Vanilla settings: replace CC with gcc
myCC=gcc
myCXX=g++

—Build sqlite3
wget http://www.sqlite.org/2014/sqlite-autoconf-3080403.tar.gz
tar -xvf sqlite-autoconf-3080403.tar.gz
cd sqlite-autoconf-3080403
sqlitedir=/soft/python/2.7/sqlite3
mkdir -p ${sqlitedir}
./configure CC=${myCC} cc=${myCC} -prefix=${sqlitedir}
maker
make install
```

---
tar -xvf Python-${pyvers}.tgz

cd Python-${pyvers}
pyInst=/soft/python/2.7/2.7.6-vanilla/python

./configure
CC=ccX CXX=ccXX
CFLAGS=-I/soft/python/2.7/sqlite3/include -I/usr/include/ncurses -L/soft/python/2.7/sqlite3/lib -L/usr
       -Wl,-rpath=/soft/python/2.7/sqlite3/lib -Wl,-rpath=/usr/lib64/
F77=gfortran
--prefix=${pyInst}
--enable-shared

make -j8

—Install setuptools & pip

module load python/2.7.6-vanilla
wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py -O - | python
easy-install pip

—Install and set up virtualenv
# 04/25/2014 Pip for some reason doesn’t find it, we install directly (see below)

wget --no-check-certificate https://pypi.python.org/packages/source/v/virtualenv/virtualenv-1.11.4.tar.gz

tar -xvf virtualenv-1.11.4.tar.gz
pyInst=/soft/python/2.7/2.7.6-vanilla/python
python setup.py install

mkdir /soft/python/2.7/2.7.6-vanilla/virtualenvs

cd /soft/python/2.7/2.7.6-vanilla/virtualenvs
virtualenv ober

# We have an issue with the new version of pip because there is something broken with SSL
# so we downgrade pip to 1.2.1

pip install nose
pip install PIL
pip install networkx
pip install numpy

# Need some fixing for scipi to find BLAS and LAPACK
export BLAS="/opt/cray/libsci/12.1.01/GNU/48/mc12/lib/libsci_gnu.a"
export LAPACK="/opt/cray/libsci/12.1.01/GNU/48/mc12/lib/libsci_gnu.a"
export ATLAS="/opt/cray/libsci/12.1.01/GNU/48/mc12/lib/libsci_gnu.a"

pip install scapy
pip install matplotlib
pip install cython
pip install pyvcf
pip install blist
pip install sqlachemy
pip install biopython

#For now skipped — broken
pip install sip
pip install pyQT

—PLINK

wget http://pngu.mgh.harvard.edu/~purcell/plink/dist/plink-1.07-src.zip
unzip plink-1.07-src.zip

#edit Makefile (point lapack to the location of libsci)

#build croaks for some files (explanation is at http://gcc.gnu.org/gcc-4.7/porting_to.html

g++ -O3 -I-DUNIX -static -DWITH_R_PLUGINS -DWITH_ZLIB -c sets.cpp
mv plink /soft/python/2.7/2.7.6-vanilla/opt

—install GRIT

wget http://grit-bio.org/releases/GRIT-1.1.2.tar.gz

tar -xvf GRIT-1.1.2.tar.gz

cd GRIT-1.1.2

export CFLAGS=-I/soft/python/2.7/${pyvers}-vanilla/python/lib/python2.7/site-packages/numpy/core/include

python setup.py install
Python vanilla installation

There are two python vanilla versions installed on Beagle2:

```
ams@login1:~> module avail python
```

Vanilla in this context means that installation is used without any customizations or updates applied to it. Take a look at this two different builds of python, one for non vanilla and another one for vanilla version. You will notice that python/3.3.0 is built with the assumption that it will be used with MPI (network is specified), while python/3.3.0-vanilla was built with gcc which doesn't support mpich2. Multiprocessing will work just fine with vanilla installation. In general those two installations have the same functionality except that vanilla can not be used for any MPI runs.

```
## Recipe for python/3.3.0
wget http://www.python.org/ftp/python/3.3.0/Python-3.3.0.tar.bz2
tar -xvf Python-3.3.0.tar.bz2
cd Python-${version}
module switch PrgEnv-cray PrgEnv-gnu
##executable is called python3
# Multinode-MPI version
pyver=3.3
version=3.3.0
installdir=/soft/python/${pyver}/${version}/python/
mkdir -p $installdir
module swap gcc/4.6.1 gcc/4.7.2
module swap atp/1.3.0 atp/1.5.2
./configure
  CC="cc" \
  CXX="CC" \
  F77="ftn" \
  XTPE_LINK_TYPE=dynamic \
  XTPE_NETWORK_TARGET=gemini\n  XTPE_COMPILE_TARGET=linux \n  XTPE_MC12_ENABLED=ON \n  --prefix=$installdir \n  --enable-shared
#Can't build
lzma              _sqlite3
make -j8
make -j8 install

## Recipe for python/3.3.0-vanilla
pyver=3.3
version=3.3.0
installdir=/soft/python/${pyver}/${version}-vanilla/python/
mkdir -p $installdir
./configure
  CC="gcc" \
  CXX="g++" \
  F77="gfortran" \
  XTPE_LINK_TYPE=dynamic \n  --prefix=$installdir \n  --enable-shared
#Can't build
lzma              _sqlite3
make -j8
make -j8 install
```

Cython (0.19.2)

In short Cython translates Python-looking code into C-code that compiles to reasonably fast C-code. Cython is an extension-module writing language that looks a lot like Python except for optional type declarations for variables. These type declarations allow the Cython compiler to replace generic, highly dynamic Python code with specific and very fast compiled code that is then able to be loaded into the Python run-time dynamically.
By adding some compiler directives to Cython to avoid some checks at each iteration of the loop, Cython can generate even faster C-code. To the top of your Cython code, you can add these lines:

```cython
#cython: boundscheck=False
#cython: wraparound=False
```

In this example we will show "step by step" instructions on how to apply Cython and MPI4Py to find Mandelbrot Set. All scripts mentioned below can be found in attachment.

To use Cython on Beagle2:

```bash
ams@login1:~> module load python/2.7.3-vanilla
Python version 2.7.3-vanilla loaded
```

Extract python mandelbrot function (or the function which you want to compile for fast C) to a new file .pyx

```bash
vi mandelbrot.pyx
```

```
# "cimport" is used to import special compile-time information about the numpy module
# we have to tell cython where to find numpy libraries
# on the python side it needs to know where numpy is

cimport numpy as np
import numpy as np

# Help Cython by adding annotations (they tell what the objects are)

cdef unsigned int i   # unsigned int means no negative indices with for loop
cdef unsigned int iteration

cdef double zx, zy, cx, cy, zx_new, zy_new

cdef np.ndarray[int, ndim=1] output = empty(dtype='i', shape=(len(c)))
cdef unsigned int lengthc = len(c)
...```

In main Python script (mandelbrot_mpi_numpy_cython.py) reference your function which you are going to compile in C

```python
from mandelbrot import mandelbrot
...
```

Create setup.py file in order to produce a compiled-module

```bash
vi setup.py
```

```python
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy

ext = Extension("mandelbrot", ["mandelbrot.pyx"],
    include_dirs = [numpy.get_include()])
setup(ext_modules=[ext],
    cmdclass = {'build_ext': build_ext})
```

Compile the extension module in this directory. After you do it you will get mandelbrot.so

```bash
python setup.py built_ext --inplace
```

Run your script on 10 compute nodes. PBS script is attached.

```bash
qsub mandel_mpi_numpy_cython.pbs
```

After running this script your data would be pickled in the same directory, which was specified with this line of your code in your mandel_mpi_numpy_cython.py script.

```python
outfile = open('/path/to/your/lustre/dir/mandelbrot_data.dat', 'wb')
```

and pickled data will be stored in this file mandelbrot_data.dat

In the very same directory you will run Python script which contains Matplotlib instructions. It requested that you run Matplotlib on login nodes.

```bash
python mandelplot.py
```

**Efficiency comparison:**

 mpi4py+numpy+cython
Matplotlib (1.2.1)

To use Matplotlib on Beagle2:

```
ams@login1:~> module load python/2.7.3
ams@login1:~> export PYTHONPATH=${PYTHONPATH}:/soft/python/2.7/2.7.3/modules/numpy/1.7.0/lib/python2.7/site-packages
ams@login1:~> export PYTHONPATH=${PYTHONPATH}:/soft/python/2.7/2.7.3/modules/scipy/0.12.0/lib/python2.7/site-packages
ams@login1:~> export PYTHONPATH=${PYTHONPATH}:/soft/python/2.7/2.7.3/modules/matplotlib/1.2.1/lib/python2.7/site-packages
```

It is not recommended to execute Matplotlib commands on compute nodes. Instead a good approach would be to pickle your result data and make a separate script which with Matplotlib commands and execute that script on login nodes. That approach is demonstrated in Python script mandelbrot_final.py and mandelplot.py (in attach) where we are calculating and plotting Mandelbrot set using MPI4PY and Matplotlib.

```
import pickle
outfile = open('/lustre/beagle2/ams/mandel/mandelbrot_data.dat', 'wb')
pickle.dump(C, outfile )
```

```
Python parallel computations on Beagle2
Graph above was made using Python and Matplotlib module (script is in attachment: matplotlib_efficiency.py). Experiment shows difference in performance on Beagle2 between: sequential, threaded, multiprocessed, MPI and MPI load balanced Python job when you run it on 2 or 20 nodes. (in our case script finds all prime numbers)

Example of using python with multiple nodes on Beagle2

Trivial example of program which finds prime numbers. This is MPI job with load balancing and it uses 20 nodes. We are using MPI4PY python binding for MPI.

PBS script:

```
#!/bin/bash
#PBS -N loadmpi
#PBS -q batch
#PBS -l walltime=00:30:00
#PBS -l mppwidth=640
#PBS -j oe

. /opt/modules/3.2.6.6/init/bash
module swap PrgEnv-cray PrgEnv-gnu
#module load python/2.7.3-vanilla
module load numpy
export PYTHONPATH=/soft/python/2.7.3/modules/mpi4py/1.3/lib/python2.7/site-packages/

cd $PBS_O_WORKDIR

aprun -n 20 -N 1 python loadmpi.py
```
Example of using python multiprocessing on Beagle2

Trivial example of program which finds prime numbers. This is multiprocessing job.

PBS script:

```bash
#!/bin/bash
#PBS -N multi10
#PBS -q batch
#PBS -l walltime=00:30:00
#PBS -l mppwidth=32
#PBS -j oe

. /opt/modules/3.2.6.6/init/bash
module swap PrgEnv-cray PrgEnv-gnu
export PATH=/soft/python/2.7/2.7.3/python/bin/:$PATH
export LD_LIBRARY_PATH=/soft/python/2.7/2.7.3/python/lib/:$LD_LIBRARY_PATH

cd $PBS_O_WORKDIR
aprun -n 1 -d 32 ./multiproc.py
```

Using Multiprocessing and weave module on Beagle2

In the following we describe a method for implementing parallel computations on a shared memory configuration (two "Abu Dhabi" processors on a 64 GB node of Beagle2, for a total of 32 threads). Most of the following content has been kindly provided by Yahui Peng from the Radiology Department @ UChicago.

We used the `multiprocessing` package because it side-steps the Global Interpreter Lock that renders the `threading` module unable to run multiple threads concurrently. The semaphore and barrier functionalities appear to be working properly on Beagle2.

Multiprocessing can also be used to run parallel jobs on multiple nodes (in principle), but this capability has not been tested yet (May 2011).

We will describe how we run computations that involved inlined C code using the `weave` module, which is part of `scipy`. It should be straightforward to remove those parts from the following instructions if they are not needed.

Recommended approach:

* In the python program add `import scipy` to use scipy and weave (it is best to call the function that uses the inlined code once before the spawning of multiple threads in order to create a single .so file. See a tutorial on its use @ https://github.com/scipy/scipy/blob/master/scipy/weave/doc/tutorial)

  - In the python program add `import multiprocessing`. See description @ http://docs.python.org/library/multiprocessing.html#module-multiprocessing for implementation details (not described here because problem specific)
  - use a PBS script based on the following example (based on the bash shell, other shells might need a few changes; note that terms between < and > need to be defined by the user – don’t confuse with output and error redirecting signs; note that here we assume that all directories were already created, add `mkdir -p <dir_name>` where needed; note that we use a simplified version of the PBS script that specifies only the total number of cores requested via the directive `#PBS -l mppwidth`. It produces the same outcome of more complex directives if used properly)
#!/bin/bash
#PBS -N <python job name>
#PBS -l mppwidth= <number of cores, 32 for single node>
#PBS -j oe # join output and error files from PBS
#PBS -l walltime=00:10:00 # total wall time requested, here the default 10 minutes
## other directives are possible here, see http://www.ci.uchicago.edu/wiki/bin/view/Beagle/ComputeOnBeagle#PBS_scripts

# Create the module command
. /opt/modules/3.2.6.7/init/bash
# load scipy and load suitable version of python to run it (as of June 2011 version 2.7) plus other modules you might need
# send result of the commands to PBS output file
module load scipy <other_modules_used> 2>&1
module list 2>&1

#scipy.weave attempts to write to $HOME because of that you have to change HOME variable
export HOME=/lustre/beagle2/$LOGNAME
export PYTHONPATH=<location of python modules used>:${PYTHONPATH}
# Location of the libraries for python 2.7
export LDFLAGS="-L/soft/python/2.7/2.7.1/lib*
# Set environmental variables to a tmp in lustre, to prevent the system from trying to write temp files to /home,
# which is not allowed on the compute nodes. These two are not always necessary.
export TMP="/lustre/beagle2/`whoami`/tmp"
export TMPDIR="/lustre/beagle2/`whoami`/tmp"
# This is necessary when using weave because otherwise python will attempt to work on /home,
# which cannot be done from compute nodes
export PYTHONCOMPILED="/lustre/beagle2/`whoami`/mypythonlibs"

# cd to the directory where the qsub command was initiated, here we assume it is where the input and executables are.
cd $PBS_O_WORKDIR
INPUTNAME=<input_file>
PROGRAM=<python_code>.py

# location of the directory where these calculations will be run
$MYLUSTREDIR = /lustre/beagle2/`whoami`/<run directory for this calculation>

# Copy run and input files to the run directory
cp $INPUTNAME $PROGRAM $MYLUSTREDIR

cd $MYLUSTREDIR
# Run for a single core, single MPI-like process (-n 1), 24 threads (-d 32), one MPI per node
# (-N 1, not really necessary for this specific example)
# of course specific programs might have a different call, but the general idea should be clear from the following
aprun -n 1 -d 32 -N 1 python ${PROGRAM} ${INPUTNAME} <other_flags>

# Copy results back to workdir
cp <output_file_names> $PBS_O_WORKDIR