Dedalus Project Documentation

Keaton Burns, Jeffrey S. Oishi, Geoff Vasil, Ben Brown, Daniel Le

Jul 10, 2018

Contents

CHAPTER 1

About Dedalus

Dedalus is a flexible framework for solving partial differential equations using spectral methods. The code is open-source and developed by a team of researchers working on problems in astrophysical and geophysical fluid dynamics.

The code is written primarily in Python and features an easy-to-use interface, including text-based equation entry. Our numerical algorithm produces highly sparse systems for a wide variety of equations on spectrally-discretized domains. These systems are efficiently solved using compiled libraries and multidimensional parallelization though MPI.

CHAPTER 2

Doc Contents

2.1 Installing Dedalus

2.1.1 Installation Script

Dedalus provides an all-in-one installation script that will build an isolated stack containing a Python installation and the other dependencies needed to run Dedalus. In most cases, the script can be modified to link with system installations of FFTW, MPI, and linear algebra libraries.

You can get the installation script from this link, or download it using:

```
wget https://bitbucket.org/dedalus-project/dedalus/raw/tip/docs/install.sh
```

and execute it using:

```
bash install.sh
```

The installation script has been tested on a number of Linux distributions and OS X. If you run into trouble using the script, please get in touch on the user list.

2.1.2 Manual Installation

Dependencies

Dedalus primarily relies on the basic components of a scientific Python stack using Python 3. Below are instructions for building the dependency stack on a variety of machines and operating systems:

Install notes for Mac OS X (10.9)

These instructions assume you're starting with a clean Mac OS X system, which will need `python3` and all scientific packages installed.

Mac OS X cookbook

```
#!bash

# Homebrew
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/
↪install)"
brew update
brew doctor
# ** Fix any errors raised by brew doctor before proceeding **

# Prep system
brew install gcc
brew install swig

# Python 3
brew install python3

# Scientific packages for Python 3
brew tap homebrew/science
brew install suite-sparse
pip3 install nose
pip3 install numpy
pip3 install scipy
brew install libpng
brew install freetype
pip3 install matplotlib

# MPI
brew install openmpi
pip3 install mpi4py

# FFTW
brew install fftw --with-mpi

# HDF5
brew install hdf5
pip3 install h5py

# Dedalus
# ** Change to the directory where you want to keep the Dedalus repository **
brew install hg
hg clone https://bitbucket.org/dedalus-project/dedalus
cd dedalus
pip3 install -r requirements.txt
python3 setup.py build_ext --inplace
```

Detailed install notes for Mac OS X (10.9)

Preparing a Mac system

First, install Xcode from the App Store and seperately install the Xcode Command Line Tools. To install the command line tools, open Xcode, go to `Preferences`, select the `Downloads` tab and `Components`. These command line tools install `make` and other requisite tools that are no longer automatically included in Mac OS X (as of 10.8).

Next, you should install the Homebrew package manager for OS X. Run the following from the Terminal:

```
#!bash
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/
↪install)"
brew update
brew doctor
```

Cleanup any problems identified by `brew doctor` before proceeding.

To complete the `scipy` install process, we'll need `gfortran` from `gcc` and `swig`, which you can install from Homebrew:

```
#!bash
brew install gcc
brew install swig
```

Install Python 3

Now, install python3 from Homebrew:

```
#!bash
brew install python3
```

Scientific packages for Python3

Next install the numpy and scipy scientific packages. To adequately warn you before proceeding, properly installing numpy and scipy on a Mac can be a frustrating experience.

Start by proactively installing UMFPACK from suite-sparse, located in homebrew-science on https://github.com/Homebrew/homebrew-science. Failing to do this may lead to a series of perplexing UMFPACK errors during the scipy install.

```
#!bash
brew tap homebrew/science
brew install suite-sparse
```

Now use pip, the (the standard Python package management system, installed with Python via Homebrew) to install `nose`, `numpy`, and `scipy` in order:

```
#!bash
pip3 install nose
pip3 install numpy
pip3 install scipy
```

The `scipy` install can fail in a number of surprising ways. Be especially wary of custom settings to `LDFLAGS`, `CPPFLAGS`, etc. within your shell; these may cause the `gfortran` compile step to fail spectacularly.

Also install `matplotlib`, the main Python plotting library, along with its dependencies, using Homebrew and pip:

```
#!bash
brew install libpng
brew install freetype
pip3 install matplotlib
```

Other libraries

Dedalus is parallelized using MPI, and we recommend using the Open MPI library on OS X. The Open MPI library and Python wrappers can be installed using Homebrew and pip:

```
#!bash
brew install openmpi
pip3 install mpi4py
```

Dedalus uses the FFTW library for transforms and parallelized transposes, and can be installed using Homebrew:

```
#!bash
brew install fftw --with-mpi
```

Dedalus uses HDF5 for data storage. The HDF5 library and Python wrappers can be installed using Homebrew and pip:

```
#!bash
brew install hdf5
pip3 install h5py
```

Installing the Dedalus package

Dedalus is managed using the Mercurial distributed version control system, and hosted online though Bitbucket. Mercurial (referred to as `hg`) can be installed using homebrew, and can then be used to download the latest copy of Dedalus (note: you should change to the directory where you want the put the Dedalus repository):

```
#!bash
brew install hg
hg clone https://bitbucket.org/dedalus-project/dedalus
cd dedalus
```

A few other Python packages needed by Dedalus are listed in the `requirements.txt` file in the Dedalus repository, and can be installed using pip:

```
#!bash
pip3 install -r requirements.txt
```

You then need to build Dedalus's Cython extensions from within the repository using the `setup.py` script. This step should be perfomed whenever updates are pulled from the main repository (but it is only strictly necessary when the Cython extensions are modified).

```
#!bash
python3 setup.py build_ext --inplace
```

Finally, you need to add the Dedalus repository to the Python search path so that the `dedalus` package can be imported. To do this, add the following to your `~/.bash_profile`, substituting in the path to the Dedalus repository you cloned using Mercurial:

```
# Add Dedalus repository to Python search path
export PYTHONPATH=<PATH/TO/DEDALUS/REPOSITORY>:$PYTHONPATH
```

Other resources

http://www.lowindata.com/2013/installing-scientific-python-on-mac-os-x/

http://stackoverflow.com/questions/12574604/scipy-install-on-mountain-lion-failing

https://github.com/jonathansick/dotfiles/wiki/Notes-for-Mac-OS-X

Install notes for TACC/Stampede

Install notes for building our python3 stack on TACC/Stampede, using the intel compiler suite. Many thanks to Yaakoub El Khamra at TACC for help in sorting out the python3 build and numpy linking against a fast MKL BLAS.

On Stampede, we can in principle either install with a `gcc/mpvapich2/fftw3` stack with `OpenBLAS`, or with an `intel/mvapich2/fftw3 stack` with `MKL`. Mpvaich2 is causing problems for us, and this appears to be a known issue with `mvapich2/1.9`, so for now we must use the `intel/mvapich2/fftw3` stack, which has `mvapich2/2.0b`. The intel stack should also, in principle, allow us to explore auto-offloading with the Xenon MIC hardware accelerators. Current `gcc` instructions can be found under NASA Pleiades.

Modules

Here is my current build environment (from running `module list`)

1. TACC-paths
2. Linux
3. cluster-paths
4. TACC
5. cluster
6. intel/14.0.1.106
7. mvapich2/2.0b

Note: To get here from a gcc default do the following:

module unload mkl module swap gcc intel/14.0.1.106

In the `intel` compiler stack, we need to use `mvapich2/2.0b`, which then implies `intel/14.0.1.106`. Right now, TACC has not built `fftw3` for this stack, so we'll be doing our own FFTW build.

See the Stampede user guide for more details. If you would like to always auto-load the same modules at startup, build your desired module configuration and then run:

```
module save
```

For ease in structuring the build, for now we'll define:

```
export BUILD_HOME=$HOME/build_intel
```

Python stack

Building Python3

Create ~\build_intel and then proceed with downloading and installing Python-3.3:

```
cd ~/build_intel
wget http://www.python.org/ftp/python/3.3.3/Python-3.3.3.tgz
tar -xzf Python-3.3.3.tgz
cd Python-3.3.3

# make sure you have the python patch, put it in Python-3.3.3
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/python_intel_patch.
↪tar
tar xvf python_intel_patch.tar

./configure --prefix=$BUILD_HOME \
                     CC=icc CFLAGS="-mkl -O3 -xHost -fPIC -ipo" \
                     CXX=icpc CPPFLAGS="-mkl -O3 -xHost -fPIC -ipo" \
                     F90=ifort F90FLAGS="-mkl -O3 -xHost -fPIC -ipo" \
                     --enable-shared LDFLAGS="-lpthread" \
                     --with-cxx-main=icpc --with-system-ffi

make
make install
```

To successfully build python3, the key is replacing the file ffi64.c, which is done automatically by downloading and unpacking this crude patch python_intel_patch.tar in your Python-3.3.3 directory. Unpack it in Python-3.3.3 (tar xvf python_intel_patch.tar line above) and it will overwrite ffi64.c. If you forget to do this, you'll see a warning/error that _ctypes couldn't be built. This is important.

Here we are building everything in ~/build_intel; you can do it whereever, but adjust things appropriately in the above instructions. The build proceeeds quickly (few minutes).

Installing FFTW3

We need to build our own FFTW3, under intel 14 and mvapich2/2.0b:

```
 wget http://www.fftw.org/fftw-3.3.3.tar.gz
 tar -xzf fftw-3.3.3.tar.gz
 cd fftw-3.3.3

./configure --prefix=$BUILD_HOME \
                      CC=mpicc \
                      CXX=mpicxx \
                      F77=mpif90 \
                      MPICC=mpicc MPICXX=mpicxx \
                      --enable-shared \
                      --enable-mpi --enable-openmp --enable-threads
 make
 make install
```

It's critical that you use mpicc as the C-compiler, etc. Otherwise the libmpich libraries are not being correctly linked into libfftw3_mpi.so and dedalus failes on fftw import.

Updating shell settings

At this point, `python3` is installed in `~/build_intel/bin/`. Add this to your path and confirm (currently there is no `python3` in the default path, so doing a `which python3` will fail if you haven't added `~/build_intel/bin`).

On Stampede, login shells (interactive connections via ssh) source only `~/.bash_profile`, `~/.bash_login` or `~/.profile`, in that order, and do not source `~/.bashrc`. Meanwhile non-login shells only launch `~/.bashrc` (see Stampede user guide).

In the bash shell, add the following to `.bashrc`:

```
export PATH=~/build_intel/bin:$PATH
export LD_LIBRARY_PATH=~/build_intel/lib:$LD_LIBRARY_PATH
```

and the following to `.profile`:

```
if [ -f ~/.bashrc ]; then . ~/.bashrc; fi
```

(from bash reference manual) to obtain the same behaviour in both shell types.

Installing pip

We'll use `pip` to install our python library depdencies. Instructions on doing this are available here and summarized below. First download and install setup tools:

```
cd ~/build
wget https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py
python3 ez_setup.py
```

Then install `pip`:

```
wget --no-check-certificate https://raw.github.com/pypa/pip/master/contrib/get-pip.py
python3 get-pip.py --cert /etc/ssl/certs/ca-bundle.crt
```

Now edit `~/.pip/pip.conf`:

```
[global]
cert = /etc/ssl/certs/ca-bundle.crt
```

You will now have `pip3` and `pip` installed in `~/build/bin`. You might try doing `pip -V` to confirm that `pip` is built against python 3.3. We will use `pip3` throughout this documentation to remain compatible with systems (e.g., Mac OS) where multiple versions of python coexist.

Installing nose

Nose is useful for unit testing, especially in checking our numpy build:

```
pip3 install nose
```

Numpy and BLAS libraries

Building numpy against MKL

Now, acquire `numpy` (1.8.0):

```
cd ~/build_intel
wget http://sourceforge.net/projects/numpy/files/NumPy/1.8.0/numpy-1.8.0.tar.gz
tar -xvf numpy-1.8.0.tar.gz
cd numpy-1.8.0
wget http://lcd-www.colorado.edu/bpbrown/dedalus_documentation/_downloads/numpy_intel_
↪patch.tar
tar xvf numpy_inte_patch.tar
```

This last step saves you from needing to hand edit two files in `numpy/distutils`; these are `intelccompiler.py` and `fcompiler/intel.py`. I've built a crude patch, `numpy_intel_patch.tar` which can be auto-deployed by within the `numpy-1.8.0` directory by the instructions above. This will unpack and overwrite:

```
numpy/distutils/intelccompiler.py
numpy/distutils/fcompiler/intel.py
```

We'll now need to make sure that `numpy` is building against the MKL libraries. Start by making a `site.cfg` file:

```
cp site.cfg.example site.cfg
emacs -nw site.cfg
```

Edit `site.cfg` in the `[mkl]` section; modify the library directory so that it correctly point to TACC's `$MKLROOT/lib/intel64/`. With the modules loaded above, this looks like:

```
[mkl]
library_dirs = /opt/apps/intel/13/composer_xe_2013_sp1.1.106/mkl/lib/intel64
include_dirs = /opt/apps/intel/13/composer_xe_2013_sp1.1.106/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

These are based on intels instructions for compiling numpy with ifort and they seem to work so far.

Then proceed with:

```
python3 setup.py config --compiler=intelem build_clib --compiler=intelem build_ext --
↪compiler=intelem install
```

This will config, build and install numpy.

Test numpy install

Test that things worked with this executable script `numpy_test_full`. You can do this full-auto by doing:

```
wget http://lcd-www.colorado.edu/bpbrown/dedalus_documentation/_downloads/numpy_test_
↪full
chmod +x numpy_test_full
./numpy_test_full
```

or do so manually by launching `python3` and then doing:

```
import numpy as np
np.__config__.show()
```

If you've installed `nose` (with `pip3 install nose`), we can further test our numpy build with:

```
np.test()
np.test('full')
```

We fail `np.test()` with two failures, while `np.test('full')` has 3 failures and 19 errors. But we do successfully link against the fast BLAS libraries (look for `FAST BLAS` output, and fast dot product time).

Note: We should check what impact these failed tests have on our results.

Python library stack

After `numpy` has been built (see links above) we will proceed with the rest of our python stack. Right now, all of these need to be installed in each existing virtualenv instance (e.g., `openblas`, `mkl`, etc.).

For now, skip the venv process.

Installing Scipy

Scipy is easier, because it just gets its config from numpy. Download an install in your appropriate `~/venv/INSTANCE` directory:

```
wget http://sourceforge.net/projects/scipy/files/scipy/0.13.2/scipy-0.13.2.tar.gz
tar -xvf scipy-0.13.2.tar.gz
cd scipy-0.13.2
```

Then run

```
python3 setup.py config --compiler=intelem --fcompiler=intelem build_clib \
                                        --compiler=intelem --fcompiler=intelem build_
↪ext \
                                        --compiler=intelem --fcompiler=intelem install
```

Installing mpi4py

This should just be pip installed:

```
pip3 install mpi4py==2.0.0
```

Note: If we use use

```
pip3 install mpi4py
```

then stampede tries to pull version 0.6.0 of mpi4py. Hence the explicit version pull above.

Installing cython

This should just be pip installed:

```
pip3 install -v https://pypi.python.org/packages/source/C/Cython/Cython-0.20.tar.gz
```

The Feb 11, 2014 update to cython (0.20.1) seems to have broken (at least with intel compilers).:

```
pip3 install cython
```

Installing matplotlib

This should just be pip installed:

```
pip3 install -v https://downloads.sourceforge.net/project/matplotlib/matplotlib/
↪matplotlib-1.3.1/matplotlib-1.3.1.tar.gz
```

Note: If we use use

```
pip3 install matplotlib
```

then stampede tries to pull version 1.1.1 of matplotlib. Hence the explicit version pull above.

Installing sympy

Do this with a regular pip install:

```
pip3 install sympy
```

Installing HDF5 with parallel support

The new analysis package brings HDF5 file writing capbaility. This needs to be compiled with support for parallel (mpi) I/O:

```
wget http://www.hdfgroup.org/ftp/HDF5/current/src/hdf5-1.8.12.tar
tar xvf hdf5-1.8.12.tar
cd hdf5-1.8.12
./configure --prefix=$BUILD_HOME \
                    CC=mpicc \
                    CXX=mpicxx \
                    F77=mpif90 \
                    MPICC=mpicc MPICXX=mpicxx \
                    --enable-shared --enable-parallel
make
make install
```

Installing h5py

Next, install h5py. We wish for full HDF5 parallel goodness, so we can do parallel file access during both simulations and post analysis as well. This will require building directly from source (see Parallel HDF5 in h5py for further details). Here we go:

```
git clone https://github.com/h5py/h5py.git
cd h5py
export CC=mpicc
export HDF5_DIR=$BUILD_HOME
python3 setup.py configure --mpi
python3 setup.py build
python3 setup.py install
```

After this install, `h5py` shows up as an `.egg` in `site-packages`, but it looks like we pass the `suggested demo2.py` test from Parallel HDF5 in h5py.

Installing h5py with collectives

We've been exploring the use of collectives for faster parallel file writing. To build that version of the h5py library:

```
git clone https://github.com/andrewcollette/h5py.git
cd h5py
git checkout mpi_collective
export CC=mpicc
export HDF5_DIR=$BUILD_HOME
python3 setup.py configure --mpi
python3 setup.py build
python3 setup.py install
```

To enable collective outputs within dedalus, edit `dedalus2/data/evaluator.py` and replace:

```
# Assemble nonconstant subspace
subshape, subslices, subdata = self.get_subspace(out)
dset = task_group.create_dataset(name=name, shape=subshape, dtype=dtype)
dset[subslices] = subdata
```

with

```
# Assemble nonconstant subspace
subshape, subslices, subdata = self.get_subspace(out)
dset = task_group.create_dataset(name=name, shape=subshape, dtype=dtype)
with dset.collective:
    dset[subslices] = subdata
```

Alternatively, you can see this same edit in some of the forks (Lecoanet, Brown).

Note: There are some serious problems with this right now; in particular, there seems to be an issue with empty arrays causing h5py to hang. Troubleshooting is ongoing.

Dedalus2

With the modules set as above, set:

```
export BUILD_HOME=$HOME/build_intel
export FFTW_PATH=$BUILD_HOME
export MPI_PATH=$MPICH_HOME
export HDF5_DIR=$BUILD_HOME
export CC=mpicc
```

Then change into your root dedalus directory and run:

```
python setup.py build_ext --inplace
```

Our new stack (`intel/14`, `mvapich2/2.0b`) builds to completion and runs test problems successfully. We have good scaling in limited early tests.

Running Dedalus on Stampede

Source the appropriate virtualenv:

```
source ~/venv/openblas/bin/activate
```

or:

```
source ~/venv/mkl/bin/activate
```

grab an interactive dev node with `idev`. Play.

Skipped libraries

Installing freetype2

Freetype is necessary for matplotlib

```
cd ~/build
wget http://sourceforge.net/projects/freetype/files/freetype2/2.5.2/freetype-2.5.2.
↪tar.gz
tar -xvf freetype-2.5.2.tar.gz
cd freetype-2.5.2
./configure --prefix=$HOME/build
make
make install
```

Note: Skipping for now

Installing libpng

May need this for matplotlib?:

```
cd ~/build
wget http://prdownloads.sourceforge.net/libpng/libpng-1.6.8.tar.gz
./configure --prefix=$HOME/build
make
make install
```

Note: Skipping for now

UMFPACK

We may wish to deploy UMFPACK for sparse matrix solves. Keaton is starting to look at this now. If we do, both numpy and scipy will require UMFPACK, so we should build it before proceeding with those builds.

UMFPACK requires AMD (another package by the same group, not processor) and SuiteSparse_config, too.

If we need UMFPACK, we can try installing it from `suite-sparse` as in the Mac install. Here are links to UMFPACK docs and Suite-sparse

Note: We'll check and update this later. (1/9/14)

All I want for christmas is suitesparse

Well, maybe :) Let's give it a try, and lets grab the whole library:

```
wget http://www.cise.ufl.edu/research/sparse/SuiteSparse/current/SuiteSparse.tar.gz
tar xvf SuiteSparse.tar.gz

<edit SuiteSparse_config/SuiteSparse_config.mk>
```

Note: Notes from the original successful build process:

Just got a direct call from Yaakoub. Very, very helpful. Here's the quick rundown.

He got _ctypes to work by editing the following file:

vim /work/00364/tg456434/yye00/src/Python-3.3.3/Modules/_ctypes/libffi/src/x86/ffi64.c

Do build with intel 14 use mvapich2/2.0b Will need to do our own build of fftw3

set mpicc as c compiler rather than icc, same for CXX, FC and others, when configuring python. should help with mpi4py.

in mpi4py, can edit mpi.cfg (non-pip install).

Keep Yaakoub updated with direct e-mail on progress.

Also, Yaakoub is spear-heading TACCs efforts in doing auto-offload to Xenon Phi.

Beware of disk quotas if you're trying many builds; I hit 5GB pretty fast and blew my matplotlib install due to quota limits :)

Installing virtualenv (skipped)

In order to test multiple numpys and scipys (and really, their underlying BLAS libraries), we will use `virtualenv`:

```
pip3 install virtualenv
```

Next, construct a virtualenv to hold all of your python modules. We suggest doing this in your home directory:

```
mkdir ~/venv
```

Python3

Note: With help from Yaakoub, we now build `_ctypes` successfully.

Also, the mpicc build is much, much slower than icc. Interesting. And we crashed out. Here's what we tried with mpicc:

```
./configure --prefix=$BUILD_HOME \
                CC=mpicc CFLAGS="-mkl -O3 -xHost -fPIC -ipo" \
                CXX=mpicxx CPPFLAGS="-mkl -O3 -xHost -fPIC -ipo" \
                F90=mpif90 F90FLAGS="-mkl -O3 -xHost -fPIC -ipo" \
                --enable-shared LDFLAGS="-lpthread" \
                --with-cxx-main=mpicxx --with-system-ffi
```

Install notes for NASA/Pleiades

Best performance is coming from our newly developed Pleaides/Intel/MKL stack; we've retained our gcc/openblas build for future use.

Install notes for NASA/Pleiades: Intel stack

An initial Pleiades environment is pretty bare-bones. There are no modules, and your shell is likely a csh varient. To switch shells, send an e-mail to support@nas.nasa.gov; I'll be using `bash`.

Then add the following to your `.profile`:

```
# Add your commands here to extend your PATH, etc.

module load comp-intel
module load git
module load openssl
module load emacs

export BUILD_HOME=$HOME/build

export PATH=$BUILD_HOME/bin:$BUILD_HOME:/$PATH  # Add private commands to PATH

export LD_LIBRARY_PATH=$BUILD_HOME/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/nasa/openssl/1.0.1h/lib/:$LD_LIBRARY_PATH

export CC=mpicc

#pathing for Dedalus
export LOCAL_MPI_VERSION=openmpi-1.10.1
export LOCAL_MPI_SHORT=v1.10

# falling back to 1.8 until we resolve tcp wireup errors
# (probably at runtime with MCA parameters)
```

(continues on next page)

(continued from previous page)

```
export LOCAL_MPI_VERSION=openmpi-1.8.6
export LOCAL_MPI_SHORT=v1.8

export LOCAL_PYTHON_VERSION=3.5.0
export LOCAL_NUMPY_VERSION=1.10.1
export LOCAL_SCIPY_VERSION=0.16.1
export LOCAL_HDF5_VERSION=1.8.15-patch1
export LOCAL_MERCURIAL_VERSION=3.6

export MPI_ROOT=$BUILD_HOME/$LOCAL_MPI_VERSION
export PYTHONPATH=$BUILD_HOME/dedalus:$PYTHONPATH
export MPI_PATH=$MPI_ROOT
export FFTW_PATH=$BUILD_HOME
export HDF5_DIR=$BUILD_HOME

# Openmpi forks:
export OMPI_MCA_mpi_warn_on_fork=0

# don't mess up Pleiades for everyone else
export OMPI_MCA_btl_openib_if_include=mlx4_0:1
```

Doing the entire build took about 2 hours. This was with several (4) open ssh connections to Pleaides to do poor-mans-parallel building (of python, hdf5, fftw, etc.), and one was on a dev node for the openmpi compile. The openmpi compile is time intensive and mus be done first. The fftw and hdf5 libraries take a while to build. Building scipy remains the most significant time cost.

Python stack

Interesting update. Pleiades now appears to have a python3 module. Fascinating. It comes with matplotlib (1.3.1), scipy (0.12), numpy (1.8.0) and cython (0.20.1) and a few others. Very interesting. For now we'll proceed with our usual build-it-from-scratch approach, but this should be kept in mind for the future. No clear mpi4py, and the `mpi4py` install was a hangup below for some time.

Building Openmpi

The suggested `mpi-sgi/mpt` MPI stack breaks with mpi4py; existing versions of openmpi on Pleiades are outdated and suffer from a previously identified bug (v1.6.5), so we'll roll our own. This needs to be built on a compute node so that the right memory space is identified.:

```
# do this on a main node (where you can access the outside internet):
cd $BUILD_HOME
wget http://www.open-mpi.org/software/ompi/$LOCAL_MPI_SHORT/downloads/$LOCAL_MPI_
↪VERSION.tar.gz
tar xvf $LOCAL_MPI_VERSION.tar.gz

# get ivy-bridge compute node
qsub -I -q devel -l select=1:ncpus=24:mpiprocs=24:model=has -l walltime=02:00:00

# once node exists
cd $BUILD_HOME
cd $LOCAL_MPI_VERSION
./configure \
    --prefix=$BUILD_HOME \
```

(continues on next page)

(continued from previous page)

```
    --enable-mpi-interface-warning \
    --without-slurm \
    --with-tm=/PBS \
    --without-loadleveler \
    --without-portals \
    --enable-mpirun-prefix-by-default \
    CC=icc CXX=icc FC=ifort

make -j
make install
```

These compilation options are based on `/nasa/openmpi/1.6.5/NAS_config.sh`, and are thanks to advice from Daniel Kokron at NAS. Compiling takes about 10-15 minutes with make -j.

Building Python3

Create `$BUILD_HOME` and then proceed with downloading and installing Python-3.4:

```
cd $BUILD_HOME
wget https://www.python.org/ftp/python/$LOCAL_PYTHON_VERSION/Python-$LOCAL_PYTHON_
↪VERSION.tgz --no-check-certificate
tar xzf Python-$LOCAL_PYTHON_VERSION.tgz
cd Python-$LOCAL_PYTHON_VERSION

./configure --prefix=$BUILD_HOME \
                    OPT="-mkl -O3 -axCORE-AVX2 -xSSE4.2 -fPIC -ipo -w -vec-report0 -
↪opt-report0" \
                    FOPT="-mkl -O3 -axCORE-AVX2 -xSSE4.2 -fPIC -ipo -w -vec-report0 -
↪opt-report0" \
                    CC=mpicc CXX=mpicxx F90=mpif90 \
                    LDFLAGS="-lpthread" \
                    --enable-shared --with-system-ffi \
                    --with-cxx-main=mpicxx --with-gcc=mpicc

make
make install
```

The previous intel patch is no longer required.

Installing pip

Python 3.4 now automatically includes pip. We suggest you do the following immediately to suppress version warning messages:

```
pip3 install --upgrade pip
```

On Pleiades, you'll need to edit `.pip/pip.conf`:

```
[global]
cert = /etc/ssl/certs/ca-bundle.trust.crt
```

You will now have `pip3` installed in `$BUILD_HOME/bin`. You might try doing `pip3 -V` to confirm that `pip3` is built against python 3.4. We will use `pip3` throughout this documentation to remain compatible with systems (e.g., Mac OS) where multiple versions of python coexist.

Installing mpi4py

This should be pip installed:

```
pip3 install mpi4py
```

Installing FFTW3

We need to build our own FFTW3, under intel 14 and mvapich2/2.0b, or under openmpi:

```
 wget http://www.fftw.org/fftw-3.3.4.tar.gz
 tar -xzf fftw-3.3.4.tar.gz
 cd fftw-3.3.4

./configure --prefix=$BUILD_HOME \
                      CC=mpicc        CFLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                      CXX=mpicxx CPPFLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                      F77=mpif90  F90FLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                      MPICC=mpicc MPICXX=mpicxx \
                      --enable-shared \
                      --enable-mpi --enable-openmp --enable-threads
 make
 make install
```

It's critical that you use `mpicc` as the C-compiler, etc. Otherwise the libmpich libraries are not being correctly linked into `libfftw3_mpi.so` and dedalus failes on fftw import.

Installing nose

Nose is useful for unit testing, especially in checking our numpy build:

```
pip3 install nose
```

Installing cython

This should just be pip installed:

```
pip3 install cython
```

Numpy and BLAS libraries

Numpy will be built against a specific BLAS library. On Pleiades we will build against the Intel MKL BLAS.

Building numpy against MKL

Now, acquire `numpy` (1.9.2):

```
cd $BUILD_HOME
wget http://sourceforge.net/projects/numpy/files/NumPy/$LOCAL_NUMPY_VERSION/numpy-
↪$LOCAL_NUMPY_VERSION.tar.gz
tar -xvf numpy-$LOCAL_NUMPY_VERSION.tar.gz
cd numpy-$LOCAL_NUMPY_VERSION
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_pleiades_intel_
↪patch.tar
tar xvf numpy_pleiades_intel_patch.tar
```

This last step saves you from needing to hand edit two files in `numpy/distutils`; these are `intelccompiler.py` and `fcompiler/intel.py`. I've built a crude patch, `numpy_pleiades_intel_patch.tar` which is auto-deployed within the `numpy-$LOCAL_NUMPY_VERSION` directory by the instructions above. This will unpack and overwrite:

```
numpy/distutils/intelccompiler.py
numpy/distutils/fcompiler/intel.py
```

This differs from prior versions in that "-xhost" is replaced with "-axCORE-AVX2 -xSSE4.2". NOTE: this is now updated for Haswell.

We'll now need to make sure that `numpy` is building against the MKL libraries. Start by making a `site.cfg` file:

```
cp site.cfg.example site.cfg
emacs -nw site.cfg
```

Note: If you're doing many different builds, it may be helpful to have the numpy site.cfg shared between builds. If so, you can edit ~/.numpy-site.cfg instead of site.cfg. This is per site.cfg.example.

Edit `site.cfg` in the `[mkl]` section; modify the library directory so that it correctly point to TACC's `$MKLROOT/lib/intel64/`. With the modules loaded above, this looks like:

```
[mkl]
library_dirs = /nasa/intel/Compiler/2015.3.187/composer_xe_2015.3.187/mkl/lib/intel64/
include_dirs = /nasa/intel/Compiler/2015.3.187/composer_xe_2015.3.187/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

These are based on intels instructions for compiling numpy with ifort and they seem to work so far.

Then proceed with:

```
python3 setup.py config --compiler=intelem build_clib --compiler=intelem build_ext --
↪compiler=intelem install
```

This will config, build and install numpy.

Test numpy install

Test that things worked with this executable script `numpy_test_full`. You can do this full-auto by doing:

```
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_test_full
chmod +x numpy_test_full
./numpy_test_full
```

We succesfully link against fast BLAS and the test results look normal.

Python library stack

After `numpy` has been built we will proceed with the rest of our python stack.

Installing Scipy

Scipy is easier, because it just gets its config from numpy. Dong a pip install fails, so we'll keep doing it the old fashioned way:

```
wget http://sourceforge.net/projects/scipy/files/scipy/$LOCAL_SCIPY_VERSION/scipy-
↪$LOCAL_SCIPY_VERSION.tar.gz
tar -xvf scipy-$LOCAL_SCIPY_VERSION.tar.gz
cd scipy-$LOCAL_SCIPY_VERSION
python3 setup.py config --compiler=intelem --fcompiler=intelem build_clib \
                                        --compiler=intelem --fcompiler=intelem build_
↪ext \
                                        --compiler=intelem --fcompiler=intelem install
```

Note: We do not have umfpack; we should address this moving forward, but for now I will defer that to a later day.

Installing matplotlib

This should just be pip installed. However, we're hitting errors with qhull compilation in every part of the 1.4.x branch, so we fall back to 1.3.1:

```
pip3 install matplotlib==1.3.1
```

Installing HDF5 with parallel support

The new analysis package brings HDF5 file writing capbaility. This needs to be compiled with support for parallel (mpi) I/O:

```
wget http://www.hdfgroup.org/ftp/HDF5/releases/hdf5-$LOCAL_HDF5_VERSION/src/hdf5-
↪$LOCAL_HDF5_VERSION.tar.gz
tar xzvf hdf5-$LOCAL_HDF5_VERSION.tar.gz
cd hdf5-$LOCAL_HDF5_VERSION
./configure --prefix=$BUILD_HOME \
                    CC=mpicc         CFLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                    CXX=mpicxx CPPFLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                    F77=mpif90  F90FLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                    MPICC=mpicc MPICXX=mpicxx \
                    --enable-shared --enable-parallel
make
make install
```

H5PY via pip

This can now just be pip installed (>=2.6.0):

```
pip3 install h5py
```

For now we drop our former instructions on attempting to install parallel h5py with collectives. See the repo history for those notes.

Installing Mercurial

On NASA Pleiades, we need to install mercurial itself. I can't get mercurial to build properly on intel compilers, so for now use gcc:

```
cd $BUILD_HOME
wget http://mercurial.selenic.com/release/mercurial-$LOCAL_MERCURIAL_VERSION.tar.gz
tar xvf mercurial-$LOCAL_MERCURIAL_VERSION.tar.gz
cd mercurial-$LOCAL_MERCURIAL_VERSION
module load gcc
export CC=gcc
make install PREFIX=$BUILD_HOME
```

I suggest you add the following to your `~/.hgrc`:

```
[ui]
username = <your bitbucket username/e-mail address here>
editor = emacs

[web]
cacerts = /etc/ssl/certs/ca-bundle.crt

[extensions]
graphlog =
color =
convert =
mq =
```

Dedalus

Preliminaries

Then do the following:

```
cd $BUILD_HOME
hg clone https://bitbucket.org/dedalus-project/dedalus
cd dedalus
pip3 install -r requirements.txt
python3 setup.py build_ext --inplace
```

Running Dedalus on Pleiades

Our scratch disk system on Pleiades is `/nobackup/user-name`. On this and other systems, I suggest soft-linking your scratch directory to a local working directory in home; I uniformly call mine `workdir`:

```
ln -s /nobackup/bpbrown workdir
```

Long-term mass storage is on LOU.

Install notes for NASA/Pleiades: Intel stack with MPI-SGI

Here we build using the recommended MPI-SGI environment, with Intel compilers. An initial Pleiades environment is pretty bare-bones. There are no modules, and your shell is likely a csh varient. To switch shells, send an e-mail to support@nas.nasa.gov; I'll be using `bash`.

Then add the following to your `.profile`:

```
# Add your commands here to extend your PATH, etc.

module load mpi-sgi/mpt
module load comp-intel
module load git
module load openssl
module load emacs

export BUILD_HOME=$HOME/build

export PATH=$BUILD_HOME/bin:$BUILD_HOME:/$PATH  # Add private commands to PATH

export LD_LIBRARY_PATH=$BUILD_HOME/lib:$LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/nasa/openssl/1.0.1h/lib/:$LD_LIBRARY_PATH

# proper wrappers for using Intel rather than GNU compilers,
# Thanks to Daniel Kokron at NASA.
export MPICC_CC=icc
export MPICXX_CXX=icpc

export CC=mpicc

#pathing for Dedalus
export LOCAL_PYTHON_VERSION=3.5.0
export LOCAL_NUMPY_VERSION=1.10.1
export LOCAL_SCIPY_VERSION=0.16.1
export LOCAL_HDF5_VERSION=1.8.15-patch1
export LOCAL_MERCURIAL_VERSION=3.6

export PYTHONPATH=$BUILD_HOME/dedalus:$PYTHONPATH
export MPI_PATH=$MPI_ROOT
export FFTW_PATH=$BUILD_HOME
export HDF5_DIR=$BUILD_HOME

# Pleaides workaround for QP errors 8/25/14 from NAS (only for MPI-SGI)
export MPI_USE_UD=true
```

Python stack

Here we use the recommended MPI-SGI compilers, rather than our own openmpi.

Building Python3

Create `$BUILD_HOME` and then proceed with downloading and installing Python-3.4:

```
cd $BUILD_HOME
wget https://www.python.org/ftp/python/$LOCAL_PYTHON_VERSION/Python-$LOCAL_PYTHON_
↪VERSION.tgz --no-check-certificate
tar xzf Python-$LOCAL_PYTHON_VERSION.tgz
cd Python-$LOCAL_PYTHON_VERSION
./configure --prefix=$BUILD_HOME \
                     OPT="-w -vec-report0 -opt-report0" \
                     FOPT="-w -vec-report0 -opt-report0" \
                     CFLAGS="-mkl -O3 -ipo -axCORE-AVX2 -xSSE4.2 -fPIC" \
                     CPPFLAGS="-mkl -O3 -ipo -axCORE-AVX2 -xSSE4.2 -fPIC" \
                     F90FLAGS="-mkl -O3 -ipo -axCORE-AVX2 -xSSE4.2 -fPIC" \
                     CC=mpicc  CXX=mpicxx F90=mpif90  \
                     --with-cxx-main=mpicxx --with-gcc=mpicc \
                     LDFLAGS="-lpthread" \
                    --enable-shared --with-system-ffi

make
make install
```

The previous intel patch is no longer required.

Installing pip

Python 3.4 now automatically includes pip.

On Pleiades, you'll need to edit `.pip/pip.conf`:

```
[global]
cert = /etc/ssl/certs/ca-bundle.trust.crt
```

You will now have `pip3` installed in `$BUILD_HOME/bin`. You might try doing `pip3 -V` to confirm that `pip3` is built against python 3.4. We will use `pip3` throughout this documentation to remain compatible with systems (e.g., Mac OS) where multiple versions of python coexist.

We suggest doing the following immediately to suppress version warning messages:

```
pip3 install --upgrade pip
```

Installing mpi4py

This should be pip installed:

```
pip3 install mpi4py
```

version >=2.0.0 seem to play well with mpi-sgi.

Installing FFTW3

We build our own FFTW3:

```
 wget http://www.fftw.org/fftw-3.3.4.tar.gz
 tar -xzf fftw-3.3.4.tar.gz
 cd fftw-3.3.4
```

(continues on next page)

(continued from previous page)

```
./configure --prefix=$BUILD_HOME \
                      CC=icc        CFLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                      CXX=icpc CPPFLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                      F77=ifort  F90FLAGS="-O3 -axCORE-AVX2 -xSSE4.2" \
                      MPICC=icc MPICXX=icpc \
                      LDFLAGS="-lmpi" \
                      --enable-shared \
                      --enable-mpi --enable-openmp --enable-threads
 make -j
 make install
```

It's critical that you use `mpicc` as the C-compiler, etc. Otherwise the libmpich libraries are not being correctly linked into `libfftw3_mpi.so` and dedalus failes on fftw import.

Installing nose

Nose is useful for unit testing, especially in checking our numpy build:

```
pip3 install nose
```

Installing cython

This should just be pip installed:

```
pip3 install cython
```

Numpy and BLAS libraries

Numpy will be built against a specific BLAS library. On Pleiades we will build against the OpenBLAS libraries.

All of the intel patches, etc. are unnecessary in the gcc stack.

Building numpy against MKL

Now, acquire `numpy` (1.10.1):

```
cd $BUILD_HOME
wget http://sourceforge.net/projects/numpy/files/NumPy/$LOCAL_NUMPY_VERSION/numpy-
↪$LOCAL_NUMPY_VERSION.tar.gz
tar -xvf numpy-$LOCAL_NUMPY_VERSION.tar.gz
cd numpy-$LOCAL_NUMPY_VERSION
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_pleiades_intel_
↪patch.tar
tar xvf numpy_pleiades_intel_patch.tar
```

This last step saves you from needing to hand edit two files in `numpy/distutils`; these are `intelccompiler.py` and `fcompiler/intel.py`. I've built a crude patch, `numpy_pleiades_intel_patch.tar` which is auto-deployed within the `numpy-$LOCAL_NUMPY_VERSION` directory by the instructions above. This will unpack and overwrite:

```
numpy/distutils/intelccompiler.py
numpy/distutils/fcompiler/intel.py
```

This differs from prior versions in that "-xhost" is replaced with "-axCORE-AVX2 -xSSE4.2". I think this could be handled more gracefully using a extra_compile_flag option in the site.cfg.

We'll now need to make sure that `numpy` is building against the MKL libraries. Start by making a `site.cfg` file:

```
cp site.cfg.example site.cfg
emacs -nw site.cfg
```

Edit `site.cfg` in the `[mkl]` section; modify the library directory so that it correctly point to TACC's `$MKLROOT/lib/intel64/`. With the modules loaded above, this looks like:

```
[mkl]
library_dirs = /nasa/intel/Compiler/2015.3.187/composer_xe_2015.3.187/mkl/lib/intel64/
include_dirs = /nasa/intel/Compiler/2015.3.187/composer_xe_2015.3.187/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

These are based on intels instructions for compiling numpy with ifort and they seem to work so far.

Then proceed with:

```
python3 setup.py config --compiler=intelem build_clib --compiler=intelem build_ext --
↪compiler=intelem install
```

This will config, build and install numpy.

Test numpy install

Test that things worked with this executable script `numpy_test_full`. You can do this full-auto by doing:

```
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_test_full
chmod +x numpy_test_full
./numpy_test_full
```

We succesfully link against fast BLAS and the test results look normal.

Python library stack

After `numpy` has been built we will proceed with the rest of our python stack.

Installing Scipy

Scipy is easier, because it just gets its config from numpy. Dong a pip install fails, so we'll keep doing it the old fashioned way:

```
wget http://sourceforge.net/projects/scipy/files/scipy/$LOCAL_SCIPY_VERSION/scipy-
↪$LOCAL_SCIPY_VERSION.tar.gz
tar -xvf scipy-$LOCAL_SCIPY_VERSION.tar.gz
cd scipy-$LOCAL_SCIPY_VERSION
python3 setup.py config --compiler=intelem --fcompiler=intelem build_clib \
```

(continues on next page)

(continued from previous page)

```
                                        --compiler=intelem --fcompiler=intelem build_
↪ext \
                                        --compiler=intelem --fcompiler=intelem install
```

Note: We do not have umfpack; we should address this moving forward, but for now I will defer that to a later day.

Installing matplotlib

This should just be pip installed. In versions of matplotlib>1.3.1, Qhull has a compile error if the C compiler is used rather than C++, so we force the C complier to be icpc

```
export CC=icpc
pip3 install matplotlib
```

Installing HDF5 with parallel support

The new analysis package brings HDF5 file writing capbaility. This needs to be compiled with support for parallel (mpi) I/O. Intel compilers are failing on this when done with mpi-sgi, and on NASA's recommendation we're falling back to gcc for this library:

```
export MPICC_CC=
export MPICXX_CXX=
wget http://www.hdfgroup.org/ftp/HDF5/releases/hdf5-$LOCAL_HDF5_VERSION/src/hdf5-
↪$LOCAL_HDF5_VERSION.tar.gz
tar xzvf hdf5-$LOCAL_HDF5_VERSION.tar.gz
cd hdf5-$LOCAL_HDF5_VERSION
./configure --prefix=$BUILD_HOME CC=mpicc CXX=mpicxx F77=mpif90 \
                    --enable-shared --enable-parallel
make
make install
```

H5PY via pip

This can now just be pip installed (>=2.6.0):

    pip3 install h5py

For now we drop our former instructions on attempting to install parallel h5py with collectives. See the repo history for those notes.

Installing Mercurial

On NASA Pleiades, we need to install mercurial itself. I can't get mercurial to build properly on intel compilers, so for now use gcc:

```
cd $BUILD_HOME
wget http://mercurial.selenic.com/release/mercurial-$LOCAL_MERCURIAL_VERSION.tar.gz
tar xvf mercurial-$LOCAL_MERCURIAL_VERSION.tar.gz
```

(continues on next page)

(continued from previous page)

```
cd mercurial-$LOCAL_MERCURIAL_VERSION
module load gcc
export CC=gcc
make install PREFIX=$BUILD_HOME
```

I suggest you add the following to your ~/.hgrc:

```
[ui]
username = <your bitbucket username/e-mail address here>
editor = emacs

[web]
cacerts = /etc/ssl/certs/ca-bundle.crt

[extensions]
graphlog =
color =
convert =
mq =
```

Dedalus

Preliminaries

Then do the following:

```
cd $BUILD_HOME
hg clone https://bitbucket.org/dedalus-project/dedalus
cd dedalus
pip3 install -r requirements.txt
python3 setup.py build_ext --inplace
```

Running Dedalus on Pleiades

Our scratch disk system on Pleiades is /nobackup/user-name. On this and other systems, I suggest soft-linking your scratch directory to a local working directory in home; I uniformly call mine workdir:

```
ln -s /nobackup/bpbrown workdir
```

Long-term mass storage is on LOU.

Install notes for NASA/Pleiades: gcc stack

Note: Warning. These instructions for a gcc stack are quite outdated and have not been tested in well over a year. A lot has shifted in the stack since then (e.g., h5py, matplotlib) and using these is at your own risk. We have been using the intel compilers exclusively on Pleiades, so please see those instructions. These gcc instructions are kept for posterity and future use.

Old instructions

An initial Pleiades environment is pretty bare-bones. There are no modules, and your shell is likely a csh varient. To switch shells, send an e-mail to support@nas.nasa.gov; I'll be using `bash`.

Then add the following to your `.profile`:

```
# Add your commands here to extend your PATH, etc.

module load gcc
module load git
module load openssl

export BUILD_HOME=$HOME/build

export PATH=$BUILD_HOME/bin:$BUILD_HOME:/$PATH  # Add private commands to PATH

export LD_LIBRARY_PATH=$BUILD_HOME/lib:$LD_LIBRARY_PATH

export CC=mpicc

#pathing for Dedalus2
export MPI_ROOT=$BUILD_HOME/openmpi-1.8
export PYTHONPATH=$BUILD_HOME/dedalus2:$PYTHONPATH
export MPI_PATH=$MPI_ROOT
export FFTW_PATH=$BUILD_HOME
```

Note: We are moving here to a python 3.4 build. Also, it looks like scipy-0.14 and numpy 1.9 are going to have happier sparse matrix performance.

Doing the entire build took about 1 hour. This was with several (4) open ssh connections to Pleaides to do poor-mans-parallel building (of openBLAS, hdf5, fftw, etc.), and one was on a dev node for the openmpi and openblas compile.

Python stack

Interesting update. Pleiades now appears to have a python3 module. Fascinating. It comes with matplotlib (1.3.1), scipy (0.12), numpy (1.8.0) and cython (0.20.1) and a few others. Very interesting. For now we'll proceed with our usual build-it-from-scratch approach, but this should be kept in mind for the future. No clear mpi4py, and the `mpi4py` install was a hangup below for some time.

Building Openmpi

The suggested `mpi-sgi/mpt` MPI stack breaks with mpi4py; existing versions of openmpi on Pleiades are outdated and suffer from a previously identified bug (v1.6.5), so we'll roll our own. This needs to be built on a compute node so that the right memory space is identified.:

```
# do this on a main node (where you can access the outside internet):
cd $BUILD_HOME
wget http://www.open-mpi.org/software/ompi/v1.8/downloads/openmpi-1.8.tar.gz
tar xvf openmpi-1.7.3.tar.gz
```

(continues on next page)

(continued from previous page)

```
# get ivy-bridge compute node
qsub -I -q devel -l select=1:ncpus=20:mpiprocs=20:model=ivy -l walltime=02:00:00

# once node exists
cd $BUILD_HOME
cd openmpi-1.7.3
./configure \
    --prefix=$BUILD_HOME \
    --enable-mpi-interface-warning \
    --without-slurm \
    --with-tm=/PBS \
    --without-loadleveler \
    --without-portals \
    --enable-mpirun-prefix-by-default \
    CC=gcc

make
make install
```

These compilation options are based on `/nasa/openmpi/1.6.5/NAS_config.sh`, and are thanks to advice from Daniel Kokron at NAS.

We're using openmpi 1.7.3 here because something substantial changes in 1.7.4 and from that point onwards instances of mpirun hang on Pleiades, when used on more than 1 node worth of cores. I've tested this extensively with a simple hello world program (http://www.dartmouth.edu/~rc/classes/intro_mpi/hello_world_ex.html) and for now suggest we move forward until this is resolved.

Building Python3

Create `$BUILD_HOME` and then proceed with downloading and installing Python-3.4:

```
cd $BUILD_HOME
wget https://www.python.org/ftp/python/3.4.0/Python-3.4.0.tgz --no-check-certificate
tar xzf Python-3.4.0.tgz
cd Python-3.4.0

./configure --prefix=$BUILD_HOME \
                    CC=mpicc \
                    CXX=mpicxx \
                    F90=mpif90 \
                    --enable-shared LDFLAGS="-lpthread" \
                    --with-cxx-main=mpicxx --with-system-ffi

make
make install
```

All of the intel patches, etc. are unnecessary in the gcc stack.

Note: We're getting a problem on `_curses_panel` and on `_sqlite3`; ignoring for now.

Installing pip

Python 3.4 now automatically includes pip.

On Pleiades, you'll need to edit `.pip/pip.conf`:

```
[global]
cert = /etc/ssl/certs/ca-bundle.crt
```

You will now have `pip3` installed in `$BUILD_HOME/bin`. You might try doing `pip3 -V` to confirm that `pip3` is built against python 3.4. We will use `pip3` throughout this documentation to remain compatible with systems (e.g., Mac OS) where multiple versions of python coexist.

Installing mpi4py

This should be pip installed:

```
pip3 install mpi4py
```

Note: Test that this works by doing a:

from mpi4py import MPI

This will segfault on sgi-mpi, but appears to work fine on openmpi-1.8, 1.7.3, etc.

Installing FFTW3

We need to build our own FFTW3, under intel 14 and mvapich2/2.0b:

```
 wget http://www.fftw.org/fftw-3.3.4.tar.gz
 tar -xzf fftw-3.3.4.tar.gz
 cd fftw-3.3.4

./configure --prefix=$BUILD_HOME \
                      CC=mpicc \
                      CXX=mpicxx \
                      F77=mpif90 \
                      MPICC=mpicc MPICXX=mpicxx \
                      --enable-shared \
                      --enable-mpi --enable-openmp --enable-threads
 make
 make install
```

It's critical that you use `mpicc` as the C-compiler, etc. Otherwise the libmpich libraries are not being correctly linked into `libfftw3_mpi.so` and dedalus failes on fftw import.

Installing nose

Nose is useful for unit testing, especially in checking our numpy build:

```
pip3 install nose
```

Installing cython

This should just be pip installed:

```
pip3 install cython
```

The Feb 11, 2014 update to cython (0.20.1) seems to work with gcc.

Numpy and BLAS libraries

Numpy will be built against a specific BLAS library. On Pleiades we will build against the OpenBLAS libraries.

All of the intel patches, etc. are unnecessary in the gcc stack.

Building OpenBLAS

From Stampede instructions:

```
# this needs to be done on a frontend
cd $BUILD_HOME
git clone git://github.com/xianyi/OpenBLAS

# suggest doing this build on a compute node, so we get the
# right number of openmp threads and architecture
cd $BUILD_HOME
cd OpenBLAS
make
make PREFIX=$BUILD_HOME install
```

Here's the build report before the `make install`:

```
OpenBLAS build complete. (BLAS CBLAS LAPACK LAPACKE)

OS               ... Linux
Architecture     ... x86_64
BINARY           ... 64bit
C compiler       ... GCC  (command line : mpicc)
Fortran compiler ... GFORTRAN  (command line : gfortran)
Library Name     ... libopenblas_sandybridgep-r0.2.9.rc2.a (Multi threaded; Max num-
↪threads is 40)
```

Building numpy against OpenBLAS

Now, acquire `numpy` (1.8.1):

```
wget http://sourceforge.net/projects/numpy/files/NumPy/1.8.1/numpy-1.8.1.tar.gz
tar xvf numpy-1.8.1.tar.gz
cd numpy-1.8.1
```

Create `site.cfg` with information for the OpenBLAS library directory

Next, make a site specific config file:

```
cp site.cfg.example site.cfg
emacs -nw site.cfg
```

Edit `site.cfg` to uncomment the `[openblas]` section; modify the library and include directories so that they correctly point to your `~/build/lib` and `~/build/include` (note, you may need to do fully expanded paths). With my account settings, this looks like:

```
[openblas]
libraries = openblas
library_dirs = /u/bpbrown/build/lib
include_dirs = /u/bpbrown/build/include
```

where `$BUILD_HOME=/u/bpbrown/build`. We may in time want to consider adding fftw as well. Now build:

```
python3 setup.py config build_clib build_ext install
```

This will config, build and install numpy.

Test numpy install

Test that things worked with this executable script `numpy_test_full`. You can do this full-auto by doing:

```
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_test_full
chmod +x numpy_test_full
./numpy_test_full
```

We succesfully link against fast BLAS and the test results look normal.

Python library stack

After `numpy` has been built we will proceed with the rest of our python stack.

Installing Scipy

Scipy is easier, because it just gets its config from numpy. Dong a pip install fails, so we'll keep doing it the old fashioned way:

```
wget http://sourceforge.net/projects/scipy/files/scipy/0.13.3/scipy-0.13.3.tar.gz
tar -xvf scipy-0.13.3.tar.gz
cd scipy-0.13.3
python3 setup.py config build_clib build_ext install
```

Note: We do not have umfpack; we should address this moving forward, but for now I will defer that to a later day.

Installing matplotlib

This should just be pip installed:

```
pip3 install matplotlib
```

Installing sympy

This should just be pip installed:

```
pip3 install sympy
```

Installing HDF5 with parallel support

The new analysis package brings HDF5 file writing capbaility. This needs to be compiled with support for parallel (mpi) I/O:

```
wget http://www.hdfgroup.org/ftp/HDF5/current/src/hdf5-1.8.12.tar
tar xvf hdf5-1.8.12.tar
cd hdf5-1.8.12
./configure --prefix=$BUILD_HOME \
                    CC=mpicc \
                    CXX=mpicxx \
                    F77=mpif90 \
                    MPICC=mpicc MPICXX=mpicxx \
                    --enable-shared --enable-parallel
make
make install
```

Next, install h5py. For reasons that are currently unclear to me, this cannot be done via pip install.

Installing h5py with collectives

We've been exploring the use of collectives for faster parallel file writing.

git is having some problems, especially with it's SSL version. I suggest adding the following to `~/.gitconfig`:

```
[http]
sslCAinfo = /etc/ssl/certs/ca-bundle.crt
```

This is still not working, owing (most likely) to git being built on an outdated SSL version. Here's a short-term hack:

```
export GIT_SSL_NO_VERIFY=true
```

To build that version of the h5py library:

```
git clone git://github.com/andrewcollette/h5py
cd h5py
git checkout mpi_collective
export CC=mpicc
export HDF5_DIR=$BUILD_HOME
python3 setup.py configure --mpi
python3 setup.py build
python3 setup.py install
```

Here's the original h5py repository:

```
git clone git://github.com/h5py/h5py
cd h5py
export CC=mpicc
```

(continues on next page)

(continued from previous page)

```
export HDF5_DIR=$BUILD_HOME
python3 setup.py configure --mpi
python3 setup.py build
python3 setup.py install
```

Note: This is ugly. We're getting a "-R" error at link, triggered by distutils not recognizing that mpicc is gcc or something like that. Looks like we're failing `if self._is_gcc(compiler)` For now, I've hand-edited unixccompiler.py in `lib/python3.3/distutils` and changed this line:

> **def _is_gcc(self, compiler_name):** return "gcc" in compiler_name or "g++" in compiler_name

to:

> **def _is_gcc(self, compiler_name):** return "gcc" in compiler_name or "g++" in compiler_name or "mpicc" in compiler_name

This is a hack, but it get's us running and alive!

Note: Ahh... I understand what's happening here. We built with `mpicc`, and the test `_is_gcc` looks for whether gcc appears anywhere in the compiler name. It doesn't in `mpicc`, so the `gcc` checks get missed. This is only ever used in the `runtime_library_dir_option()` call. So we'd need to either rename the mpicc wrapper something like `mpicc-gcc` or do a test on `compiler --version` or something. Oh boy. Serious upstream problem for mpicc wrapped builds that cythonize and go to link. Hmm...

Installing Mercurial

On NASA Pleiades, we need to install mercurial itself:

```
wget http://mercurial.selenic.com/release/mercurial-2.9.tar.gz
tar xvf mercurial-2.9.tar.gz
cd mercurial-2.9
make install PREFIX=$BUILD_HOME
```

I suggest you add the following to your `~/.hgrc`:

```
[ui]
username = <your bitbucket username/e-mail address here>
editor = emacs

[web]
cacerts = /etc/ssl/certs/ca-bundle.crt

[extensions]
graphlog =
color =
convert =
mq =
```

Dedalus2

Preliminaries

With the modules set as above, set:

```
export BUILD_HOME=$BUILD_HOME
export FFTW_PATH=$BUILD_HOME
export MPI_PATH=$BUILD_HOME/openmpi-1.8
```

Then change into your root dedalus directory and run:

```
python setup.py build_ext --inplace
```

further packages needed for Keaton's branch:

```
pip3 install tqdm
pip3 install pathlib
```

Running Dedalus on Pleiades

Our scratch disk system on Pleiades is `/nobackup/user-name`. On this and other systems, I suggest soft-linking your scratch directory to a local working directory in home; I uniformly call mine `workdir`:

```
ln -s /nobackup/bpbrown workdir
```

Long-term mass storage is on LOU.

Install notes for NASA/Discover

This installation is fairly straightforward because most of the work has already been done by the NASA/Discover staff, namely Jules Kouatchou.

First, add the following lines to your `~/.bashrc` file and source it:

```
module purge
module load other/comp/gcc-4.9.1
module load lib/mkl-15.0.0.090
module load other/Py3Dist/py-3.4.1_gcc-4.9.1_mkl-15.0.0.090
module load other/mpi/openmpi/1.8.2-gcc-4.9.1

export BUILD_HOME=$HOME/build
export PYTHONPATH=$HOME/dedalus2
```

This loads the gcc compiler, MKL linear algebra package, openmpi version 1.8.2, and crucially various python3 libraries. To see the list of python libraries,

```
listPyPackages
```

We actually have all the python libraries we need for Dedalus. However, we still need fftw. To install fftw,

```
mkdir build
```

(continues on next page)

(continued from previous page)

```
cd $BUILD_HOME
wget http://www.fftw.org/fftw-3.3.4.tar.gz
tar -xzf fftw-3.3.4.tar.gz
cd fftw-3.3.4

./configure --prefix=$BUILD_HOME \
                    CC=mpicc \
                    CXX=mpicxx \
                    F77=mpif90 \
                    MPICC=mpicc MPICXX=mpicxx \
                    --enable-shared \
                    --enable-mpi --enable-openmp --enable-threads

make
make install
```

All that remains is to pull Dedalus down from Bitbucket and install it.

```
cd $HOME
hg clone https://bitbucket.org/dedalus-project/dedalus2

export FFTW_PATH=$BUILD_HOME
export HDF5_DIR=$BUILD_HOME
export MPI_DIR=/usr/local/other/SLES11.1/openMpi/1.8.2/gcc-4.9.1
cd $HOME/dedalus2
python3 setup.py build_ext --inplace
```

Install notes for PSC/Bridges: Intel stack

Here we build using the recommended Intel compilers. Bridges comes with python 3.4 at present, but for now we'll maintain a boutique build to keep access to python >=3.5 and to tune numpy performance by hand (though the value proposition of this should be tested).

Then add the following to your `.bash_profile`:

```
# Add your commands here to extend your PATH, etc.

export BUILD_HOME=$HOME/build

export PATH=$BUILD_HOME/bin:$BUILD_HOME:/$PATH  # Add private commands to PATH
export LD_LIBRARY_PATH=$BUILD_HOME/lib:$LD_LIBRARY_PATH

export CC=mpiicc

export I_MPI_CC=icc

#pathing for Dedalus
export LOCAL_PYTHON_VERSION=3.5.1
export LOCAL_NUMPY_VERSION=1.11.0
export LOCAL_SCIPY_VERSION=0.17.0
export LOCAL_HDF5_VERSION=1.8.16
export LOCAL_MERCURIAL_VERSION=3.7.3

export PYTHONPATH=$BUILD_HOME/dedalus:$PYTHONPATH
export MPI_PATH=$MPI_ROOT
```

(continues on next page)

(continued from previous page)

```
export FFTW_PATH=$BUILD_HOME
export HDF5_DIR=$BUILD_HOME
```

Python stack

Here we use the recommended Intel mpi compilers, rather than our own openmpi.

Building Python3

Create `$BUILD_HOME` and then proceed with downloading and installing Python-3:

```
cd $BUILD_HOME
wget https://www.python.org/ftp/python/$LOCAL_PYTHON_VERSION/Python-$LOCAL_PYTHON_
↪VERSION.tgz --no-check-certificate
tar xzf Python-$LOCAL_PYTHON_VERSION.tgz
cd Python-$LOCAL_PYTHON_VERSION
./configure --prefix=$BUILD_HOME \
                     OPT="-w -vec-report0 -opt-report0" \
                     FOPT="-w -vec-report0 -opt-report0" \
                     CFLAGS="-mkl -O3 -ipo -xCORE-AVX2 -fPIC" \
                     CPPFLAGS="-mkl -O3 -ipo -xCORE-AVX2 -fPIC" \
                     F90FLAGS="-mkl -O3 -ipo -xCORE-AVX2 -fPIC" \
                     CC=mpiicc  CXX=mpiicpc F90=mpiifort  \
                     LDFLAGS="-lpthread"
make -j
make install
```

The previous intel patch is no longer required.

Installing pip

Python 3.4+ now automatically includes pip.

You will now have `pip3` installed in `$BUILD_HOME/bin`. You might try doing `pip3 -V` to confirm that `pip3` is built against python 3.4. We will use `pip3` throughout this documentation to remain compatible with systems (e.g., Mac OS) where multiple versions of python coexist.

We suggest doing the following immediately to suppress version warning messages:

```
pip3 install --upgrade pip
```

Installing mpi4py

This should be pip installed:

```
pip3 install mpi4py
```

This required setting the `I_MPI_CC=icc` envirnoment variable above; otherwise we keep hitting gcc.

Installing FFTW3

We build our own FFTW3:

```
 wget http://www.fftw.org/fftw-3.3.4.tar.gz
 tar -xzf fftw-3.3.4.tar.gz
 cd fftw-3.3.4

./configure --prefix=$BUILD_HOME \
                      CC=mpiicc        CFLAGS="-O3 -xCORE-AVX2" \
                      CXX=mpiicpc CPPFLAGS="-O3 -xCORE-AVX2" \
                      F77=mpiifort  F90FLAGS="-O3 -xCORE-AVX2" \
                      MPICC=mpiicc MPICXX=mpiicpc \
                      LDFLAGS="-lmpi" \
                      --enable-shared \
                      --enable-mpi --enable-openmp --enable-threads
 make -j
 make install
```

It's critical that you use `mpicc` as the C-compiler, etc. Otherwise the libmpich libraries are not being correctly linked into `libfftw3_mpi.so` and dedalus failes on fftw import.

Installing nose

Nose is useful for unit testing, especially in checking our numpy build:

```
pip3 install nose
```

Installing cython

This should just be pip installed:

```
pip3 install cython
```

Numpy and BLAS libraries

Numpy will be built against a specific BLAS library.

Building numpy against MKL

Now, acquire `numpy`. The login nodes for Bridges are 14-core Haswell chips, just like the compute nodes, so let's try doing it with the normal numpy settings (no patching to adjust the compiler commands in distutils for cross-compiling). Ah shoots. Nope. The numpy distutils only employs xSSE4.2 and none of the AVX2 arch flags, nor a basic xhost. Well. On we go. Change `-xSSE4.2` to `-xCORE-AVX2` in `numpy/distutils/intelccompiler.py` and `numpy/distutils/fcompiler/intel.py`. We should really put in a PR and an ability to pass flags via `site.cfg` or other approach.

Here's an automated way to do this, using `numpy_intel.patch`.:

```
cd $BUILD_HOME
wget http://sourceforge.net/projects/numpy/files/NumPy/$LOCAL_NUMPY_VERSION/numpy-
↪$LOCAL_NUMPY_VERSION.tar.gz
tar -xvf numpy-$LOCAL_NUMPY_VERSION.tar.gz
cd numpy-$LOCAL_NUMPY_VERSION
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_intel.patch
patch -p1 < numpy_intel.patch
```

We'll now need to make sure that `numpy` is building against the MKL libraries. Start by making a `site.cfg` file:

```
cat >> site.cfg << EOF
[mkl]
library_dirs = /opt/packages/intel/compilers_and_libraries/linux/mkl/lib/intel64
include_dirs = /opt/packages/intel/compilers_and_libraries/linux/mkl/include
mkl_libs = mkl_rt
lapack_libs =
EOF
```

Then proceed with:

```
python3 setup.py config --compiler=intelem build_clib --compiler=intelem build_ext --
↪compiler=intelem install
```

This will config, build and install numpy.

Test numpy install

Test that things worked with this executable script `numpy_test_full`. You can do this full-auto by doing:

```
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_test_full
chmod +x numpy_test_full
./numpy_test_full
```

Numpy has changed the location of _dotblas, so our old test doesn't work. From the dot product speed, it looks like we have succesfully linked against fast BLAS and the test results look relatively normal, but this needs to be looked in to.

Python library stack

After `numpy` has been built we will proceed with the rest of our python stack.

Installing Scipy

Scipy is easier, because it just gets its config from numpy. Scipy now is no longer hosted at sourceforge for anything past v0.16, so lets try git:

```
git clone git://github.com/scipy/scipy.git scipy
cd scipy
# fall back to stable version
git checkout tags/v$LOCAL_SCIPY_VERSION
python3 setup.py config --compiler=intelem --fcompiler=intelem build_clib \
```

(continues on next page)

(continued from previous page)

```
                                        --compiler=intelem --fcompiler=intelem build_
↪ext \
                                        --compiler=intelem --fcompiler=intelem install
```

Note: We do not have umfpack; we should address this moving forward, but for now I will defer that to a later day. Again. Still.

Installing matplotlib

This should just be pip installed. In versions of matplotlib>1.3.1, Qhull has a compile error if the C compiler is used rather than C++, so we force the C complier to be icpc

```
export CC=icpc
pip3 install matplotlib
```

Installing HDF5 with parallel support

The new analysis package brings HDF5 file writing capbaility. This needs to be compiled with support for parallel (mpi) I/O. Intel compilers are failing on this when done with mpi-sgi, and on NASA's recommendation we're falling back to gcc for this library:

```
wget http://www.hdfgroup.org/ftp/HDF5/releases/hdf5-$LOCAL_HDF5_VERSION/src/hdf5-
↪$LOCAL_HDF5_VERSION.tar.gz
tar xzvf hdf5-$LOCAL_HDF5_VERSION.tar.gz
cd hdf5-$LOCAL_HDF5_VERSION
./configure --prefix=$BUILD_HOME CC=mpiicc CXX=mpiicpc F77=mpiifort \
                    --enable-shared --enable-parallel
make
make install
```

H5PY via pip

This can now just be pip installed (>=2.6.0):

```
pip3 install h5py
```

For now we drop our former instructions on attempting to install parallel h5py with collectives. See the NASA/Pleiades repo history for those notes.

Installing Mercurial

Here we install mercurial itself. Following NASA/Pleiades approaches, we will use gcc. I haven't checked whether the default bridges install has mercurial:

```
cd $BUILD_HOME
wget http://mercurial.selenic.com/release/mercurial-$LOCAL_MERCURIAL_VERSION.tar.gz
tar xvf mercurial-$LOCAL_MERCURIAL_VERSION.tar.gz
```

(continues on next page)

(continued from previous page)

```
cd mercurial-$LOCAL_MERCURIAL_VERSION
module load gcc
export CC=gcc
make install PREFIX=$BUILD_HOME
```

I suggest you add the following to your ~/.hgrc:: cat >> ~/.hgrc << EOF [ui] username = <your bitbucket username/e-mail address here> editor = emacs

[extensions] graphlog = color = convert = mq = EOF

Dedalus

Preliminaries

Then do the following:

```
cd $BUILD_HOME
hg clone https://bitbucket.org/dedalus-project/dedalus
cd dedalus
# this has some issues with mpi4py versioning --v
pip3 install -r requirements.txt
python3 setup.py build_ext --inplace
```

Running Dedalus on Bridges

Our scratch disk system on Bridges is `/pylon1/group-name/user-name`. On this and other systems, I suggest soft-linking your scratch directory to a local working directory in home; I uniformly call mine `workdir`:

```
ln -s /pylon1/group-name/user-name workdir
```

Long-term spinning storage is on `/pylon2` and is provided by allocation request.

Install notes for Trestles

Note: These are a very old set of installation instructions. They likely no longer work.

Make sure to do

$ module purge

first.

Modules

This is a minimalist list for now:

- gnu/4.8.2 (this is now the default gnu module)
- openmpi_ib
- fftw/3.3.3 (make sure to do this one last, as it's compiler/MPI dependent)

Building Python3

I usually build everything in *~/build*, but you can do it whereever. Download Python-3.3. Once loading the above modules, in the Python-3.3 source directory, do

$ ./configure –prefix=$HOME/build

followed by the usual *make -j4; make install* (the *-j4* tells make to use 4 cores). You may get something like this:

```
Python build finished, but the necessary bits to build these modules were not found:
_dbm               _gdbm              _lzma
_sqlite3
To find the necessary bits, look in setup.py in detect_modules() for the module's␣
↪name.
```

I think this should be totally fine.

At this point, make sure the python3 you installed is in your path!

Installing virtualenv

In order to test multiple numpys and scipys (and really, their underlying BLAS libraries), I am using virtualenv. In order install virtulenv, once Python-3.3 is installed, you first need to install pip manuall. Follow the steps here http://www.pip-installer.org/en/latest/installing.html for "Install or Upgrade Setuptools" and then "Install or Upgrade pip". Briefly, you need to download and run ez_setup.py and then get-pip.py. This should run without incident. Once *pip* is installed, do

$ pip install virtualenv

Building OpenBLAS

To build openBLAS, first do *$ git clone https://github.com/xianyi/OpenBLAS.git* to get OpenBLAS. Then, with the modules loaded, do *make -j4*; and *make PREFIX=path/to/build/dir install*

Building numpy

First construct a virtualenv to hold all of your python modules. I like to do this right in my home directory. For example,

$ mkdir ~/venv (assuming you don't already have *~/venv*) *$ cd ~/venv $ virtualenv openblas*

will create an *openblas* directory, with a *bin* subdirectory. You "activate" the virtual env by doing *$ source path/to/virtualenv/bin/activate*. This will change all of your environment variables so that the active python will see whatever modules are in that directory. **Note that this messes with modules!** To be safe, I'd recommend *$ module purge; module load gnu openmpi_ib* afterwards.

- *$ cp site.cfg.example site.cfg*

edit *site.cfg* to uncomment the [openblas] section and fill in the include and library dirs to whereever you installed openblas.

- *$ python setup.py config*

to make sure that the numpy build has FOUND your openblas install. If it did, you should see something like this:

```
(openblas)trestles-login1:/home/../numpy-1.8.0 [10:15]$ python setup.py config
Running from numpy source directory.
F2PY Version 2
blas_opt_info:
blas_mkl_info:
  libraries mkl,vml,guide not found in ['/home/joishi/venv/openblas/lib', '/usr/local/
↪lib64', '/usr/local/lib', '/usr/lib64', '/usr/lib', '/usr/lib/']
  NOT AVAILABLE

openblas_info:
  FOUND:
    language = f77
    library_dirs = ['/home/joishi/build/lib']
    libraries = ['openblas', 'openblas']

  FOUND:
    language = f77
    library_dirs = ['/home/joishi/build/lib']
    libraries = ['openblas', 'openblas']

non-existing path in 'numpy/lib': 'benchmarks'
lapack_opt_info:
  FOUND:
    language = f77
    library_dirs = ['/home/joishi/build/lib']
    libraries = ['openblas', 'openblas']

/home/joishi/build/lib/python3.3/distutils/dist.py:257: UserWarning: Unknown␣
↪distribution option: 'define_macros'
  warnings.warn(msg)
running config
```

- *$ python setup.py build*

if successful,

- *$ python setup.py install*

Installing Scipy

Scipy is easier, because it just gets its config from numpy.

- *$ python setup.py config*

This notes a lack of UMFPACK. . . will that make a speed difference? Nevertheless, it works ok.

Do

- *$ python setup.py build*

if successful,

- *$ python setup.py install*

Installing mpi4py

This should just be pip installed, *$ pip install mpi4py*

Installing cython

This should just be pip installed, *$ pip install cython*

Installing matplotlib

This should just be pip installed, *$ pip install matplotlib*

UMFPACK

Requires AMD (another package by the same group, not processor) and SuiteSparse_config, too.

Dedalus2

With the modules set as above (for NOW), set *$ export FFTW_PATH=/opt/fftw/3.3.3/gnu/openmpi/ib* and *$ export MPI_PATH=/opt/openmpi/gnu/ib*. Then do *$ python setup.py build_ext –inplace*.

Install notes for CU/Janus

As with NASA/Pleiades, an initial Janus environment is pretty bare-bones. There are no modules, and your shell is likely a bash varient. Here we'll do a full build of our stack, using only the prebuilt openmpi compilers. Later we'll try a module heavy stack to see if we can avoid this.

Add the following to your `.my.bash_profile`:

```
# Add your commands here to extend your PATH, etc.

module load intel

export BUILD_HOME=$HOME/build

export PATH=$BUILD_HOME/bin:$BUILD_HOME:/$PATH  # Add private commands to PATH

export LD_LIBRARY_PATH=$BUILD_HOME/lib:$LD_LIBRARY_PATH

export CC=mpicc

#pathing for Dedalus
export LOCAL_MPI_VERSION=openmpi-1.8.5
export LOCAL_MPI_SHORT=v1.8
export LOCAL_PYTHON_VERSION=3.4.3
export LOCAL_NUMPY_VERSION=1.9.2
export LOCAL_SCIPY_VERSION=0.15.1
export LOCAL_HDF5_VERSION=1.8.15


export MPI_ROOT=$BUILD_HOME/$LOCAL_MPI_VERSION
export PYTHONPATH=$BUILD_HOME/dedalus:$PYTHONPATH
export MPI_PATH=$MPI_ROOT
export FFTW_PATH=$BUILD_HOME
export HDF5_DIR=$BUILD_HOME
```

Do your builds on the janus compile nodes (see MOTD). As a positive note, Janus compile nodes have access to the internet (e.g., wget), so you can download and compile on-node. For now we're using stock Pleiades compile flags and patch files. With intel 15.0.1 the cython install is now working well, as does h5py.

Building Openmpi

Tim Dunn has pointed out that we may (may) be able to get some speed improvements by building our own openmpi. Why not give it a try! Compiling on the janus-compile nodes seems to do a fine job, and unlike Pleiades we can grab software from the internet on the compile nodes too. This streamlines the process.:

```
cd $BUILD_HOME
wget http://www.open-mpi.org/software/ompi/$LOCAL_MPI_SHORT/downloads/$LOCAL_MPI_
↪VERSION.tar.gz
tar xvf $LOCAL_MPI_VERSION.tar.gz
cd $LOCAL_MPI_VERSION
./configure \
    --prefix=$BUILD_HOME \
    --with-slurm \
    --with-threads=posix \
    --enable-mpi-thread-multiple \
    CC=icc CXX=icpc FC=ifort

make -j
make install
```

Config flags thanks to Tim Dunn; the CFLAGS etc are from Pleiades and should be general.

Building Python3

Create `$BUILD_HOME` and then proceed with downloading and installing Python-3.4:

```
cd $BUILD_HOME
wget https://www.python.org/ftp/python/$LOCAL_PYTHON_VERSION/Python-$LOCAL_PYTHON_
↪VERSION.tgz
tar xzf Python-$LOCAL_PYTHON_VERSION.tgz
cd Python-$LOCAL_PYTHON_VERSION

./configure --prefix=$BUILD_HOME \
                    CC=mpicc         CFLAGS="-mkl -O3 -axAVX -xSSE4.1 -fPIC -ipo" \
                    CXX=mpicxx CPPFLAGS="-mkl -O3 -axAVX -xSSE4.1 -fPIC -ipo" \
                    F90=mpif90  F90FLAGS="-mkl -O3 -axAVX -xSSE4.1 -fPIC -ipo" \
                    --enable-shared LDFLAGS="-lpthread" \
                    --with-cxx-main=mpicxx --with-system-ffi

make -j
make install
```

The former patch for Intel compilers to handle ctypes is no longer necessary.

Installing pip

Python 3.4 now automatically includes pip.

You will now have `pip3` installed in `$BUILD_HOME/bin`. You might try doing `pip3 -V` to confirm that `pip3` is built against python 3.4. We will use `pip3` throughout this documentation to remain compatible with systems (e.g., Mac OS) where multiple versions of python coexist.

Installing mpi4py

This should be pip installed:

```
pip3 install mpi4py
```

Installing FFTW3

We need to build our own FFTW3, under intel 14 and mvapich2/2.0b, or under openmpi:

```
 cd $BUILD_HOME
 wget http://www.fftw.org/fftw-3.3.4.tar.gz
 tar xvzf fftw-3.3.4.tar.gz
 cd fftw-3.3.4

./configure --prefix=$BUILD_HOME \
                      CC=mpicc        CFLAGS="-O3 -axAVX -xSSE4.1" \
                      CXX=mpicxx CPPFLAGS="-O3 -axAVX -xSSE4.1" \
                      F77=mpif90  F90FLAGS="-O3 -axAVX -xSSE4.1" \
                      MPICC=mpicc MPICXX=mpicxx \
                      --enable-shared \
                      --enable-mpi --enable-openmp --enable-threads

 make -j
 make install
```

It's critical that you use `mpicc` as the C-compiler, etc. Otherwise the libmpich libraries are not being correctly linked into `libfftw3_mpi.so` and dedalus failes on fftw import.

Installing nose

Nose is useful for unit testing, especially in checking our numpy build:

```
pip3 install nose
```

Installing cython

This should just be pip installed:

```
pip3 install cython
```

Cython is now working (intel 15.0/openmpi-1.8.5).

Numpy and BLAS libraries

Numpy will be built against a specific BLAS library. On Pleiades we will build against the OpenBLAS libraries.

All of the intel patches, etc. are unnecessary in the gcc stack.

Building numpy against MKL

Now, acquire `numpy` (1.9.0):

```
cd $BUILD_HOME
wget http://sourceforge.net/projects/numpy/files/NumPy/$LOCAL_NUMPY_VERSION/numpy-
↪$LOCAL_NUMPY_VERSION.tar.gz
tar -xvf numpy-$LOCAL_NUMPY_VERSION.tar.gz
cd numpy-$LOCAL_NUMPY_VERSION
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_janus_intel_
↪patch.tar
tar xvf numpy_janus_intel_patch.tar
```

This last step saves you from needing to hand edit two files in `numpy/distutils`; these are `intelccompiler.py` and `fcompiler/intel.py`. I've built a crude patch, `numpy_janus_intel_patch.tar` which is auto-deployed within the `numpy-$LOCAL_NUMPY_VERSION` directory by the instructions above. This will unpack and overwrite:

```
numpy/distutils/intelccompiler.py
numpy/distutils/fcompiler/intel.py
```

This differs from prior versions in that "-xhost" is replaced with "-axAVX -xSSE4.1".

We'll now need to make sure that `numpy` is building against the MKL libraries. Start by making a `site.cfg` file:

```
cp site.cfg.example site.cfg
emacs -nw site.cfg
```

Edit `site.cfg` in the `[mkl]` section; modify the library directory so that it correctly point to TACC's `$MKLROOT/lib/intel64/`. With the modules loaded above, this looks like:

```
[mkl]
library_dirs = /curc/tools/x_86_64/rh6/intel/15.0.1/composer_xe_2015.1.133/mkl/lib/
↪intel64
include_dirs = /curc/tools/x_86_64/rh6/intel/15.0.1/composer_xe_2015.1.133/mkl/include
mkl_libs = mkl_rt
lapack_libs =
```

These are based on intels instructions for compiling numpy with ifort and they seem to work so far.

Then proceed with:

```
python3 setup.py config --compiler=intelem build_clib --compiler=intelem build_ext --
↪compiler=intelem install
```

This will config, build and install numpy.

Test numpy install

Test that things worked with this executable script `numpy_test_full`. You can do this full-auto by doing:

```
wget http://dedalus-project.readthedocs.org/en/latest/_downloads/numpy_test_full
chmod +x numpy_test_full
./numpy_test_full
```

We succesfully link against fast BLAS and the test results look normal.

Python library stack

After `numpy` has been built we will proceed with the rest of our python stack.

Installing Scipy

Scipy is easier, because it just gets its config from numpy. Dong a pip install fails, so we'll keep doing it the old fashioned way:

```
wget http://sourceforge.net/projects/scipy/files/scipy/$LOCAL_SCIPY_VERSION/scipy-
↪$LOCAL_SCIPY_VERSION.tar.gz
tar -xvf scipy-$LOCAL_SCIPY_VERSION.tar.gz
cd scipy-$LOCAL_SCIPY_VERSION
python3 setup.py config --compiler=intelem --fcompiler=intelem build_clib \
                                          --compiler=intelem --fcompiler=intelem build_
↪ext \
                                          --compiler=intelem --fcompiler=intelem install
```

Note: We do not have umfpack; we should address this moving forward, but for now I will defer that to a later day.

Installing matplotlib

This should just be pip installed. In versions of matplotlib>1.3.1, Qhull has a compile error if the C compiler is used rather than C++, so we force the C complier to be icpc

```
export CC=icpc
pip3 install matplotlib
```

Installing HDF5 with parallel support

The new analysis package brings HDF5 file writing capbaility. This needs to be compiled with support for parallel (mpi) I/O:

```
wget http://www.hdfgroup.org/ftp/HDF5/releases/hdf5-$LOCAL_HDF5_VERSION/src/hdf5-
↪$LOCAL_HDF5_VERSION.tar.gz

tar xvzf hdf5-$LOCAL_HDF5_VERSION.tar.gz
cd hdf5-$LOCAL_HDF5_VERSION
./configure --prefix=$BUILD_HOME \
                    CC=mpicc         CFLAGS="-O3 -axAVX -xSSE4.1" \
                    CXX=mpicxx CPPFLAGS="-O3 -axAVX -xSSE4.1" \
                    F77=mpif90  F90FLAGS="-O3 -axAVX -xSSE4.1" \
                    MPICC=mpicc MPICXX=mpicxx \
                    --enable-shared --enable-parallel

make -j
make install
```

Installing h5py

This now can be pip installed:

```
pip3 install hypy
```

Installing Mercurial

On Janus, we need to install mercurial itself. I can't get mercurial to build properly on intel compilers, so for now use gcc. Ah, and we also need python2 for the mercurial build (only):

```
module unload openmpi intel
module load gcc/gcc-4.9.1
module load python/anaconda-2.0.0
wget http://mercurial.selenic.com/release/mercurial-3.4.tar.gz
tar xvf mercurial-3.4.tar.gz
cd mercurial-3.4
export CC=gcc
make install PREFIX=$BUILD_HOME
```

I suggest you add the following to your `~/.hgrc`:

```
[ui]
username = <your bitbucket username/e-mail address here>
editor = emacs

[extensions]
graphlog =
color =
convert =
mq =
```

Dedalus

Preliminaries

With the modules set as above, set:

```
export BUILD_HOME=$BUILD_HOME
export FFTW_PATH=$BUILD_HOME
export MPI_PATH=$BUILD_HOME/$LOCAL_MPI_VERSION
```

Pull the dedalus repository::

```
hg clone https://bitbucket.org/dedalus-project/dedalus
```

Then change into your root dedalus directory and run:

```
pip3 install -r requirements.txt
python3 setup.py build_ext --inplace
```

Running Dedalus on Janus

Our scratch disk system on Pleiades is `/lustre/janus_scratch/user-name`. On this and other systems, I suggest soft-linking your scratch directory to a local working directory in home; I uniformly call mine `workdir`:

```
ln -s /lustre/janus_scratch/bpbrown workdir
```

I also suggest you move your stack to the `projects` directory, `/projects/user-name`. There, I bring back a symbolic link:

ln -s /projects/bpbrown projects ln -s projects/build build

Install notes for BRC HPC SAVIO cluster

Installing on the SAVIO cluster is pretty straightforward, as many things can be loaded via modules. First, load the following modules.

```
module purge
module load intel
module load openmpi
module load fftw/3.3.4-intel
module load python/3.2.3
module load nose
module load numpy/1.8.1
module load scipy/0.14.0
module load mpi4py
module load pip
module load virtualenv/1.7.2
module load mercurial/2.0.2
module load hdf5/1.8.13-intel-p
```

We next need to make a virtual environment in which to build the rest of the Dedalus depencencies.

```
virtualenv python_build
source python_build/bin/activate
```

The rest of the depencies will be pip-installed. However, because we are using intel compilers, we need to specify the compiler and some how to link things properly.

```
export CC=icc
export LDFLAGS="-lirc -limf"
```

Now we can use pip to install most of the remaining dependencies.

```
pip-3.2 install cython
pip-3.2 install h5py
pip-3.2 install matplotlib==1.3.1
```

Dedalus itself can be pulled down from Bitbucket.

```
hg clone https://bitbucket.org/dedalus-project/dedalus
cd dedalus
pip-3.2 install -r requirements.txt
```

To build Dedalus, you must specify the locations of FFTW and MPI.

```
export FFTW_PATH=/global/software/sl-6.x86_64/modules/intel/2013_sp1.4.211/fftw/3.3.4-
↪intel
export MPI_PATH=/global/software/sl-6.x86_64/modules/intel/2013_sp1.2.144/openmpi/1.6.
↪5-intel
python3 setup.py build_ext --inplace
```

Using Dedalus

To use Dedalus, put the following in your `~/.bashrc` file:

```
module purge
module load intel
module load openmpi
module load fftw/3.3.4-intel
module load python/3.2.3
module load numpy/1.8.1
module load scipy/0.14.0
module load mpi4py
module load mercurial/2.0.2
module load hdf5/1.8.13-intel-p
source python_build/bin/activate
export PYTHONPATH=$PYTHONPATH:~/dedalus
```

Install notes for MIT Engaging Cluster

This installation uses the Python, BLAS, and MPI modules available on Engaging, while manually building HDF5 and FFTW.

Modules and paths

The following commands should be added to your `~/.bashrc` file to setup the correct modules and paths. Modify the `HDF5_DIR`, `FFTW_PATH`, and `DEDALUS_REPO` environment variables as desired to change the build locations of these packages.

```
# Basic modules
module load gcc
module load slurm

# Python from modules
module load engaging/python/2.7.10
module load engaging/python/3.6.0
export PATH=~/.local/bin:${PATH}

# BLAS from modules
module load engaging/OpenBLAS/0.2.14
export BLAS=/cm/shared/engaging/OpenBLAS/0.2.14/lib/libopenblas.so

# MPI from modules
module load engaging/openmpi/2.0.3
export MPI_PATH=/cm/shared/engaging/openmpi/2.0.3

# HDF5 built from source
```

(continues on next page)

(continued from previous page)

```
export HDF5_DIR=~/software/hdf5
export HDF5_VERSION=1.10.1
export HDF5_MPI="ON"
export PATH=${HDF5_DIR}/bin:${PATH}
export LD_LIBRARY_PATH=${HDF5_DIR}/lib:${LD_LIBRARY_PATH}

# FFTW built from source
export FFTW_PATH=~/software/fftw
export FFTW_VERSION=3.3.6-pl2
export PATH=${FFTW_PATH}/bin:${PATH}
export LD_LIBRARY_PATH=${FFTW_PATH}/lib:${LD_LIBRARY_PATH}

# Dedalus from mercurial
export DEDALUS_REPO=~/software/dedalus
export PYTHONPATH=${DEDALUS_REPO}:${PYTHONPATH}
```

Build procedure

Source your `~/.bashrc` to activate the above changes, or re-login to the cluster, before running the following build procedure.

```
# Python basics
/cm/shared/engaging/python/2.7.10/bin/pip install --ignore-installed --user pip
/cm/shared/engaging/python/3.6.0/bin/pip3 install --ignore-installed --user pip
pip2 install --user --upgrade setuptools
pip2 install --user mercurial
pip3 install --user --upgrade setuptools
pip3 install --user nose cython

# Python packages
pip3 install --user --no-use-wheel numpy
pip3 install --user --no-use-wheel scipy
pip3 install --user mpi4py

# HDF5 built from source
mkdir -p ${HDF5_DIR}
cd ${HDF5_DIR}
wget https://support.hdfgroup.org/ftp/HDF5/current/src/hdf5-${HDF5_VERSION}.tar
tar -xvf hdf5-${HDF5_VERSION}.tar
cd hdf5-${HDF5_VERSION}
./configure --prefix=${HDF5_DIR} \
    CC=mpicc \
    CXX=mpicxx \
    F77=mpif90 \
    MPICC=mpicc \
    MPICXX=mpicxx \
    --enable-shared \
    --enable-parallel
make
make install
pip3 install --user --no-binary=h5py h5py

# FFTW built from source
mkdir -p ${FFTW_PATH}
cd ${FFTW_PATH}
```

(continues on next page)

(continued from previous page)

```
wget http://www.fftw.org/fftw-${FFTW_VERSION}.tar.gz
tar -xvzf fftw-${FFTW_VERSION}.tar.gz
cd fftw-${FFTW_VERSION}
./configure --prefix=${FFTW_PATH} \
    CC=mpicc \
    CXX=mpicxx \
    F77=mpif90 \
    MPICC=mpicc \
    MPICXX=mpicxx \
    --enable-shared \
    --enable-mpi \
    --enable-openmp \
    --enable-threads
make
make install

# Dedalus from mercurial
hg clone https://bitbucket.org/dedalus-project/dedalus ${DEDALUS_REPO}
cd ${DEDALUS_REPO}
pip3 install --user -r requirements.txt
python3 setup.py build_ext --inplace
```

Notes

Last updated on 2017/09/18 by Keaton Burns.

Dedalus Package

Dedalus is distributed using the mercurial version control system, and hosted on Bitbucket. To install Dedalus itself, first install mercurial, and then clone the main repository using:

```
hg clone https://bitbucket.org/dedalus-project/dedalus
```

Move into the newly cloned repository, and use pip to install any remaining Python dependencies with the command:

```
pip3 install -r requirements.txt
```

To help Dedalus find the proper libraries, it may be necessary to set the `FFTW_PATH` and `MPI_PATH` environment variables (see system-specific documentation). Dedalus's C-extensions can then be built in-place using:

```
python3 setup.py build_ext --inplace
```

Finally, add the repository directory to your `PYTHONPATH` environment variable to ensure that the `dedalus` package within can be found by the Python interpreter.

2.1.3 Updating Dedalus

To update your installation of Dedalus, move into the repository directory (located at `src/dedalus` within the installation script's build, or where you manually cloned it) and issue the mercurial update commands:

```
hg pull
hg update
```

Then rerun the pip requirements installation and python build, in case the dependencies or C-extensions have changes:

```
pip3 install -r requirements.txt
python3 setup.py build_ext --inplace
```

Dedalus should be updated.

2.2 Getting started with Dedalus

2.2.1 Tutorial Notebooks

This tutorial on using Dedalus consists of three short IPython notebooks, which can be downloaded and ran interactively, or viewed on-line through the links below.

The notebooks cover the basics of setting up and interacting with the primary facets of the code, culminating in the setup and simulation of the 1D KdV-Burgers equation.

- Tutorial 1: Bases and Domains
- Tutorial 2: Fields and Operators
- Tutorial 3: Problems and Solvers
- Tutorial 4: Analysis and Post-processing

2.2.2 Example Notebooks

Below are several notebooks that walk through the setup and execution of more complicated multidimensional example problems.

- Example 1: Kelvin Helmholtz Instability
- Example 2: Taylor-Couette Flow

2.2.3 Example Scripts

A wider range of examples are available under the `examples` subdirectory of the main code repository, which you can browse here. These examples cover a wider range of use cases, including larger multidimensional problems designed for parallel execution. Basic post-processing and plotting scripts are also provided with many problems.

These simulation and processing scripts may be useful as a starting point for implementing different problems and equation sets.

2.2.4 Contributions & Suggestions

If you have a script that you'd like to make available as an example, or to request an additional example covering different functionality or use cases, please get in touch on the dev list!

CHAPTER 3

Other Links

Learn more about Dedalus by visiting the

- Project homepage: http://dedalus-project.org
- Code repository: http://bitbucket.org/dedalus-project/dedalus
- Documentation: http://dedalus-project.readthedocs.org
- User list: https://groups.google.com/forum/#!forum/dedalus-users
- Dev list: https://groups.google.com/forum/#!forum/dedalus-dev