Import without a filesystem:

scientific Python built-in with static linking and frozen modules

Pat Marion, Kitware Inc.
Aron Ahmadia
Bradley M. Froehle, University of California, Berkeley

Presentation outline

- Intro
- Problem statement
- Summary of approaches
- This talk’s approach: built-in modules
- Challenges, automation, adoption

About me

- Pat Marion
- R&D Engineer @ Kitware
- Lately, I do point cloud processing with python/c++
- Previously, supercomputing with python

Talk accompanying material

github.com/patmarion/**NumpyBuiltinExample**

Problem statement

Python *import* scales very poorly when parallel process independently request filesystem metadata from a shared, network filesystem

How bad is it?

“catastrophic”

-William Scullin “**Python for High Performance Computing**” PyCon 2011.

“For 32k processes on BlueGene/P...

35 minutes to load and initialize interpreter

5.5 hours to import 100 trivial C-extension modules”

-Asher Langton “**Improving Python+MPI import performance**” Jan. 2012 numpy-discussion.

Aron’s talk last year

Solving the Import Problem: scalable dynamic loading

http://pyvideo.org/video/1201/solving-the-import-problem-scalable-dynamic-load

How import works

Brett Cannon’s talk from PyCon 2013:

http://pyvideo.org/video/1707/how-import-works

“how the simple from sys import version turns out to be slightly complicated”

Import documentation

- PEP 302 --- New Import Hooks
  http://www.python.org/dev/peps/pep-0302/

- Python 3 Language Reference: The Import System
  http://docs.python.org/3/reference/import.html

import numpy

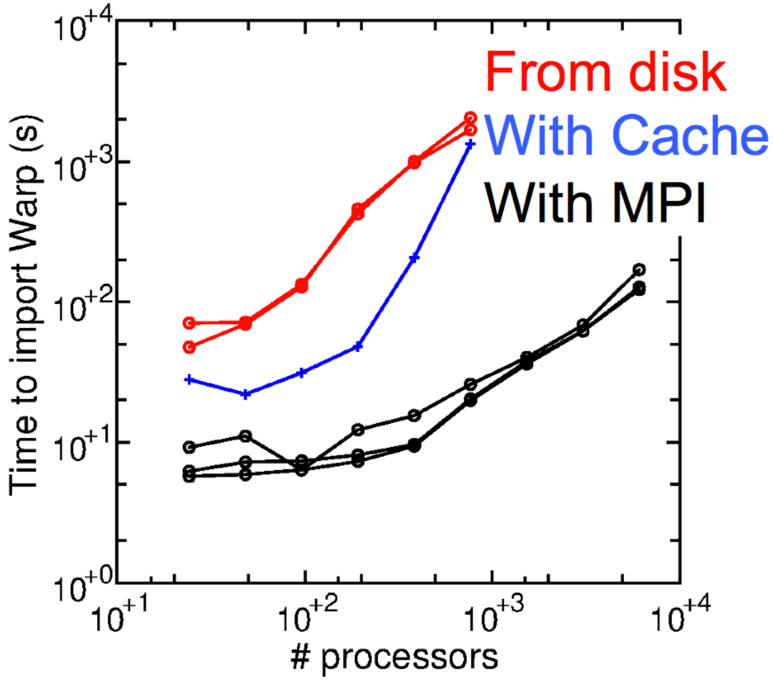
- Search each directory listed in *sys.path*
- Read python module (.py, .pyc)
- Read C extension module (.so)

```
>>> import numpy

>>> len([m for m in sys.modules.keys() if 'numpy' in m])

140
```

Timing results from Hopper



Results from Dave Grote's talk *Python in a Parallel Environment @ NERSC User Group meeting 2013*

Two ways to fix

- Change Python’s import mechanism
- Change filesystem access methods

Summary of approaches

- Intercept basic I/O ops and replace with MPI
- Path caching
- Built-in

Projects

- walla, collfs, DLcache, FMcache
- mpi_import.py, cached_import.py
- Slither

Today’s topic

- Built-in: import without a filesystem
- Combination of build techniques:
    - built-in modules
    - frozen modules

Experiment at Argonne National Labs

- Intrepid IBM Blue Gene/P @ ALCF
- Ran PHASTA simulation code with ParaView Python coprocessor @ 160k cores (full machine scale)
- Without import solution, @ 32k nodes our job was terminated before import completed (>1 hour)
- With built-in + frozen modules, import took <0.1 seconds at @ 160k cores.

What’s a built-in module?

```
>>> import sys

>>> sys.builtin_module_names

('__builtin__', '__main__', '_ast', '_codecs', '_sre',
'_symtable', '_warnings', '_weakref', 'errno',
'exceptions', 'gc', 'imp', 'marshal', 'posix', 'pwd',
'signal', 'sys', 'thread', 'xxsubtype', 'zipimport’)
```

module.__file__

```
>>> import math
>>> math.__file__
‘/usr/lib/python2.7/lib-dynload/math.so’
```

Math built-in

```
>>> import math
>>> math
<module ‘math’ (built-in)>

>>> math.__file__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute '__file__'
```

built-in: how to

- Place Setup.local in the Python Modules/ source code directory. Setup.local lists each module that should be built-in and the .c file for the module.
- Modules are compiled into libpython and linked into python interpreter.
- Python standard library modules built-in: OK
- Third party modules: hmm... what? I have to recompile libpython? No, but you must recompile the python interpreter and link your module library, or create your own python interpreter.

built-in: how it works

- “built-in” only works for C extension modules
- Register “math” with pointer to initmath() function.
- The initmath() function registers the module methods. The “import math
- When importing, Python’s import machinery looks at the table of built-in module names, finds “math” and calls initmath() directly
- ...instead of dlopen(“math.so”); dlsym(“initmath”);

Why is a patch required?

- Python’s support for built-in modules does not work for extension modules that are sub-modules (extensions inside a module package)
- Example of an extension module that is a sub-module:

```
>>> numpy.core.multiarray.__file__

‘/usr/lib/dist-packages/numpy/core/multiarray.so’
```

Python Freeze for .py files

- You can embed .py files using the freeze tool
- Part of Python source code distribution: Tools/freeze/freeze.py
- $ freeze.py myscript.py
- myscipt.py must be importable
- Collects list of all imported .py files
- Writes a C source file with the .py file content as a string (char array)

Python Freeze for .py files

```
>>> import os
>>> os
<module ‘os’ from ‘<frozen>’>

>>> os.__file__
‘<frozen>’
```

Python Freeze for .py files

```
>>> import imp
>>> imp.is_frozen(‘os’)
True
```

Python Freeze for .py files

- Just like built-in modules, you must recompile the python interpreter to add the frozen code source files (or create your own interpreter)
- Just like built-in modules, Python will search for module names in the table of frozen modules, it will get the module content as a char array instead of fopen(myfile.py); fread(file);

Remember: the main problem is the metadata

By using built-in and frozen modules, Python can locate modules by searching module tables in memory. No more searching directories in sys.path

Built-in + freeze: Pros

- It’s really fast – avoids filesystem access
- Built-in and freeze is officially supported by Python, no hacks
- It can be tested on any platform, as a single process or any scale
- Avoids dynamic loading, fully self-contained interpreter
- Although the build is complicated, once you have that figured out, then everything “just works”

Built-in + freeze: Cons

- Must compile a custom interpreter
- Requires a patch to Python (shouldn’t have to...)
- Must recompile if you change a .py file
- Build method to add third party modules is more complicated, sometimes very complicated.

Slither: static Python builds for HPC systems

- a command line tool for building static CPython binaries
- github.com/bfroehle/slither

NumPy Built-in Example

An example project demonstrating a simple hello world C program that contains NumPy and the Python standard library, built-in and frozen:

github.com/patmarion/NumpyBuiltinExample

numpy modules

```
>>> numpy.__file__

‘/usr/lib/dist-packages/numpy/__init__.pyc’


>>> numpy.__path__

[‘/usr/lib/dist-packages/numpy’]
```

Other issues: auxilary data

As mentioned in PEP 302 #open-issues:

“Modules often need supporting data files to do their job, particularly in the case of complex packages or full applications. Current practice is generally to locate such files via sys.path (or a package.__path__ attribute). This approach will not work, in general, for modules loaded via an import hook.”

numpy.test()

- Doesn’t work because nose searches for tests in the filesystem using module.__file__ attribute

Python CMake build system

github.com/davidsansome/python-cmake-buildsystem

- Portable and adaptable

Questions?