Python import scaling

Nick Johnson, EPCC

Version 1.0, December 18, 2014

1. Introduction

On supercomputers such as ARCHER, Python is regularly used in all parts of the workflow, from computation to post-processing & data visualization. When used for computational work, scaling performance can appear poor, often much worse than for statically compiled codes such as a C or Fortran. This is oft attributed to Python being an interpreted language but we find this not to be the case. The problem, more generally, is in the loading of dynamic modules in a dynamically linked (shared) code. Each instance of the Python interpreter will dynamically load the same set of modules from the disk system. At small widths (ie, small core counts), this is negligible; as the core-count increases to a level that might be expected for mid-range computational work (say $\geq$ 2400 cores) a bottleneck develops while processes wait for I/O.

In this white-paper, we examine one solution to the problem, DLFM, developed by Mike Davis of Cray, to whom I express my thanks.

2. Dynamic Linking

When a dynamically-linked application (such as Python) is executed, the set of dependent libraries is loaded in sequence by `ld.so` prior to the transfer of control to the application's main function. This set of dependent libraries ordinarily numbers a dozen or so, but can number many more, depending on the needs of the application; their names and paths are given by the `ldd` command. As `ld.so` processes each dependent library, it executes a series of system calls (mostly `open()`) to find the library and make the library's contents available to the application.

The number of `open()` calls can be many times larger than the number of dependent libraries, because `ld.so` must search for each dependent library in multiple locations. These locations include[1]:

[1] See the `ld.so` man-page for a more exhaustive list.

- anywhere specified in the `DT_RUNPATH` section of the binary;
- anywhere specified in the environmental variable `LD_LIBRARY_PATH`;
- the cache file `/etc/ld.so.cache`; and
- the default paths, `/lib` and `/usr/lib` (and 64-bit variants thereof).

When an application runs at large width (ie, at high core count or large MPI comm world size), the dynamic library load sequence often directs every PE (or MPI rank) to execute the same set of system calls on the same file system object at roughly the same time. This can cause file system contention (ie, a bottleneck) and performance degradation. When a Python application is executed, a similar condition occurs. The set of imported Python modules is loaded in the sequence specified by the order of import statements that are executed by the Python main function and any subordinate functions. As Python processes each imported module, it executes a series of system calls to find the module and make the module's contents available to the application. It is not uncommon to load many Python libraries to do computational work, from the generic Numpy (ref needed) to domain specific libraries.

3. DLFM

DLFM, or more properly, the DLFM package, is a set of libraries and tools that can be applied to a dynamically-linked application, or an application that uses Python, to provide improved performance during the loading of dynamic libraries and importing of Python modules when running the application at large width (ie, high core count). Included in the DLFM package are: a set of wrapper functions that interface with the dynamic linker (`ld.so`) to cache the application's dynamic library load operations; and a custom `libpython.so`, with a set of optimized components, that caches the application's Python import operations.
The successful use of DLFM depends on three central assumptions:

1. the application can be executed at either small or large width;

2. the number of objects to be loaded/imported, and their order, do not change from small-width to large-width runs; and

3. the number of objects to be loaded/imported, and their order, are identical across PEs. The reasons for these central assumptions will be given in the next section.

DLFM uses a rudimentary communication package, based on sockets, to facilitate the distribution of cached data to the PEs. The loading of dependent libraries by `ld.so` occurs very early in the application's execution; thus, there is limited I/O, communication, and other system-related support available to the application at that time. Specifically, there is no opportunity at this stage to employ higher-level operations such as MPI.

DLFM eases file-system contention and delivers improved performance during the loading of dynamic libraries and the importing of Python modules by caching the contents of such objects in files on the parallel (Lustre) file system. In the case of dynamic libraries, the cache file is named `dlcache.dat`, and the strategy is called *DLCaching*. In the case of Python modules, the cache file is named `fmcache.dat`, and the strategy is called *FMCaching*. DLFM can be used to perform DLCaching alone, or DLCaching and FMCaching together.

In an application built with DLFM, the cache files are read into memory very early in application startup (on the occasion of the first `open()` call), and all subsequent library-load or module-import operations are serviced out of the in-memory cache buffers. In DLCaching, the system calls normally made by `ld.so` to find and load a dependent library are intercepted and redirected to the wrapper layer, which then accesses the cache to make the dependent library's contents available to the application.

In FMCaching, the functions within `libpython.so` responsible for finding and loading a Python module are modified to access the cache to make the module's contents available to the application. For a Python extension module (which generally contains dynamically-linked, compiled code), the sequence of operations becomes the same as

that for dependent libraries.

4. Results

GPAW is a density-functional theory (DFT) Python code. It can be run at large core counts and suffers from the import problems described above. Using DLFM can significantly reduce the import time and help speed-up a run. Figure 1 shows the results for a GPAW problem run with *vanilla* Python, with Scalable Python and with DLFM.
Figure 2 shows the results from using DLFM to decrease the loading time for the synthetic benchmark, Pynamic, using 10 libraries with a total size of roughly 9 MB.

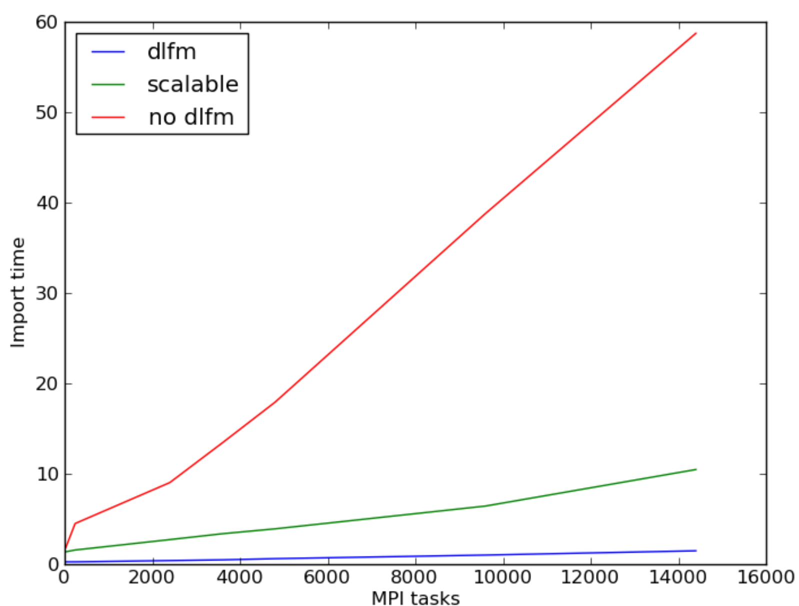

Figure 1: GPAW scaling properties with DLFM, Scalable Python and *vanilla* Python.

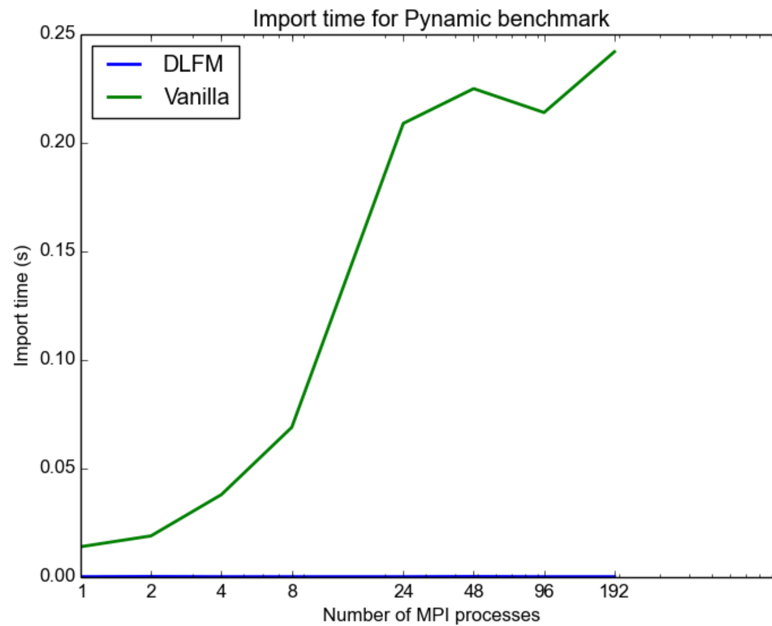

Figure 2: Pynamic benchmark scaling properties with DLFM, and *vanilla* Python.

A Building an Application with DLFM

As an example, this appendix provides a walkthrough of the process of building an application to use DL caching and FM caching.

There are four steps in this work-flow, assuming you have an existing flow in which you can build and run your application:

1. Build the application with a modified Makefile (or script) that specifes linking to a (non-existant) custom dynamic loader;
2. Create a custom dynamic loader based on a copy of `ld.so`;
3. Create the cache-file using a pilot run at small width; and
4. Run the application at large (or full) width using the cache-files.

A1. Build

The process for building an application is reasonably straight forward. Beginning with a known-good building process, the Makefile (or build script) should be modified to link against a patched Python library and specify a custom linker.
If the existing link step looked like the following:

```
cc main.o solve.o report.o \
-L/usr/lib64 -lpython2.7 \
-L${HOME}/lib -lmytools -o myapp
```

Then it should be modified to be:

```
cc main.o solve.o report.o \
${DLFM_INSTALL_DIR}/lib/*.o \
-L${DLFM_INSTALL_DIR}/lib -lpython2.7 \
-L${HOME}/lib -lmytools -o myapp \
-Wl,--dynamic-linker=‘pwd‘/ld.so
```

A2. Patch

Customizing the linker involves taking a copy of the default system linker (`ld.so`) and patching it so that instead of searching for dependant libraries, it calls the functions present in the binary which came from the DLFM code.
Executing the following will create the necessary local copy of `ld.so` and perform the necessary patching:

```
${DLFM_INSTALL_DIR}/bin/dlpatch myapp
```

You should verify (using `ldd`) that the binary (`myapp` in this case) is now dependant on the patched (local) copy of `ld.so`.
As an examine, calling `ldd` on the executable from the Pynamic test suite (`pynamic-pyMPI`) after patching, the following result is seen:

```
libgomp.so.1 => /usr/lib64/libgomp.so.1 (0x00002aaaad699000)
```

```
/fs3/z01/z01/njohnso1/dlfmdec/dlfm/pynamic/pynamic-pyMPI-2.6a1/ld.so.pynamic-pyM
libmpl.so.0 => /opt/cray/lib64/libmpl.so.0 (0x00002aaaad8a8000)
libxpmem.so.0 => /opt/cray/xpmem/default/lib64/libxpmem.so.0 (0x00002aaaadaad000
```

A3. Pilot

With a binary and patched dynamic loader in place, you now need to run a pilot job to create the cache-files for later use. This is easily accomplished by adding a few extra lines to the pbs script file submitted to the job laucher.

```
export DLFM_OP=write-cache
${DLFM_INSTALL_DIR}/bin/dlfm.pre ${DLFM_OP}
export PYTHONHOME=${DLFM_INSTALL_DIR}
aprun -n 24 -N 24 myapp
```

This run may take very slightly longer than normal whilst it writes the cache-files.

A4. Main job

Once the files `dlcache.dat` and `fmcache.dat` have been created from the pilot job, any number of main jobs can be run at differing widths. Like the pilot job, some extra lines are needed in the job submission file.

```
export DLFM_OP=read-cache
${DLFM_INSTALL_DIR}/bin/dlfm.pre ${DLFM_OP} 24000
export PYTHONHOME=${DLFM_INSTALL_DIR}
aprun -n 24000 -N 24 myapp
```

A5. Usage on ARCHER

To make life a little simpler, DLFM has been installed as a package on ARCHER. Loading the module `dlfm` will ensure that `DLFM_INSTALL_DIR` and any associated Python paths are appropriately set for use.