Using the DLFM Package on the Cray XE6 System

Mike Davis, Cray Inc.
Version 1.4 - October 2013

1.0: Introduction

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 scale. 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.

2.0: Dynamic Linking and Python Importing Without DLFM

When a dynamically-linked application 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(1)` command. As ld.so processes each dependent library, it executes a series of system calls to find the library and make the library's contents available to the application. These include:

```
fd = open (/path/to/dependent/library, O_RDONLY);
read (fd, elfhdr, 832); // to get the library ELF header
fstat (fd, &stbuf); // to get library attributes, such as size
mmap (buf, length, prot, MAP_PRIVATE, fd, offset); // read text segment

mmap (buf, length, prot, MAP_PRIVATE, fd, offset); // read data segment
close (fd);
```

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, as described on the `ld.so(1)` man page.

When the application runs at very large width (that is, at high PE count or large MPI comm size), then 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 more or less the same time. This can cause serious file system contention and performance degradation.

When a Python application is executed, 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. For an extension module (as described in http://docs.python.org/2.7/extending/extending.html,Section 3, "Extending Python With C or C++"), these calls are as shown above for the case of a dynamic library; for a standard Python module, these calls include:

```
fd = open (/path/to/module, O_RDONLY);
fstat (fd, &stbuf); // to get module attributes, such as size
```

```
read (fd, buf, size); // to get the module contents
close (fd);
```

The number of `open()` calls can be many times larger than the number of import operations, because Python must search for each module file in multiple locations, as described in http://docs.python.org/tutorial/modules.html, Section 6.1.2, "The Module Search Path". In addition, the specified module might actually refer to a package of modules, stored in the file system as a directory with a special `__init__` module and other subordinate modules; Python uses a series of additional `stat()` calls to establish whether or not a module is actually a package.

When the application runs at very large width (that is, at high PE count or large MPI comm size), then the import sequence often directs every PE (or MPI rank) to execute the same set of system calls on the same file system object at more or less the same time. This can cause serious file system contention and performance degradation.

This document will refer to dependent libraries, Python extension modules and Python standard modules collectively as "objects."

3.0: DLFM Theory of Operation

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. This choice was made because of the timing issues involved. 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.

4.0: Dynamic Linking and Python Importing With DLFM

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 file system. In the case of dynamic libraries, the cache file is `dlcache.dat`, and the strategy is called DLCaching. In the case of Python modules, the cache file is `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(2)` 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. The sequence of operations becomes:

```
memcpy (elf_hdr, dl_ptr, 832); // to get the library ELF header
memcpy (&stbuf, dl_ptr, sizeof (stbuf)); // to get library attributes
mmap (buf, length, prot, MAP_PRIVATE | MAP_ANONYMOUS);
memcpy (buf, dl_ptr, length); // read text segment
mmap (buf, length, prot, MAP_PRIVATE | MAP_ANONYMOUS);
memcpy (buf, dl_ptr, length); // read data segment
```

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, the sequence of operations becomes the same as that shown above for dependent libraries. For a Python standard module, the sequence of operations becomes:

```
memcpy (&stbuf, fm_ptr, sizeof (stbuf)); // to get library attributes
memcpy (buf, fm_ptr, size); // to get the module contents
```

Note that in both cases, all file-related system calls (`open, stat, read, close`) have been eliminated. In both cases, the in-memory cache is freed once it has been entirely read.

Using DLFM in an application involves four steps. The first two steps are performed when the application is built, and the last two steps are performed when the application is executed.

Step 1: Specifying a Custom Dynamic Linker and Python Library

The procedure for performing the custom link is as follows. Suppose that the normal command for linking the application (called `myapp`) looks like this:

```
cc main.o solve.o report.o \
    -L/usr/lib64 -lpython2.6 \
    -L${HOME}/lib -lmytools -o myapp
```

To accomplish the custom link, one would instead perform the following command:

```
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
```

The second line of the new cc command specifies that the object files that make up the wrapper layer be statically linked into the executable. The third line is a modification of the original `cc` command's second line; it specifies that the custom Python library be dynamically linked to the executable. The fifth line specifies that a custom `ld.so` be used as the dynamic linker.

Note that, at this point, the custom `ld.so` does not yet exist. The next step describes the procedure for creating it.

Step 2: Customizing the dynamic linker

Customizing the dynamic linker involves patching some code in `ld.so` so that, instead of issuing the set of system calls to load a dependent library, the linker calls the corresponding wrapper functions that are now part of the application code. The tool that performs this patching is called `dlpatch`. `dlpatch` is part of the DLFM package. The command to customize the dynamic linker is:

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

The `dlpatch` command creates a copy of `ld.so` in the current working directory, locates the addresses of the wrappper functions in the executable `myapp`, and patches instructions into the `ld.so` that cause it to branch to the appropriate addresses in `myapp` instead of executing the system calls.

Once the `dlpatch` command has been executed, the `ldd(1)` command can be used to verify that the executable `myapp` will use the custom `ld.so` rather than the system default `ld.so`.

Step 3: Creating the DLFM cache files

Before the application can make use of the DLFM cache files, it must first generate the files. This is done by executing the application in a "pilot" mode. Suppose that the job script to execute the application at a small width is as follows (the line numbers at the left are for annotation only; they do not appear in the actual script file):

```
1     #!/bin/bash
2     #PBS -S /bin/bash
3     #PBS -l mppwidth=24
4     #PBS -l mppnppn=24
5     #PBS -l walltime=1:00:00
6     #PBS -o run.out
7     #PBS -j oe
8     #
9     test "${PBS_O_WORKDIR}" != "" && cd ${PBS_O_WORKDIR}
10     aprun -n 24 -N 24 myapp
11     #
12     # All done
13     #
```

The changes necessary to create the DLFM cache files appear in the modified script below:

```
1     #!/bin/bash
2     #PBS -S /bin/bash
3     #PBS -l mppwidth=24
4     #PBS -l mppnppn=24
5     #PBS -l walltime=1:00:00
6     #PBS -v DLFM_INSTALL_DIR
7     #PBS -o run.out
8     #PBS -j oe
9     #
10     test "${PBS_O_WORKDIR}" != "" && cd ${PBS_O_WORKDIR}
11     export DLFM_OP=write-cache
12     ${DLFM_INSTALL_DIR}/bin/dlfm.pre ${DLFM_OP}
13     export PYTHONHOME=${DLFM_INSTALL_DIR}
14     aprun -n 24 -N 24 myapp
15     #
16     # All done
17     #
```

The modified script has four new lines. Line 6 exports the environment variable DLFM_INSTALL_DIR from the shell that submits the job to the shell that runs the job script. Line 11 sets the type of DLFM caching operation to perform (write-cache). Line 12 executes the `dlfm.pre` command to prepare the runtime environment for writing the cache files. This preparation includes creating the cache files with permissions specific for a write-cache run. Line 13 sets the PYTHONHOME environment variable to point to the path where the modules associated with the custom python reside. On line 14 of the modified script, as on line 10 of the original script, the application is executed at a width of 24 PEs, across the 24 cores of a single compute node. It is assumed that `myapp` has been built to do DLFM caching, as described in steps 1-2 above; as such, it will generate the cache files in the current working directory of the job script as it runs.

Step 4: Reading the DLFM cache files

Now that the application has been run at small width to create the DLFM cache files, a large-width run can be made to read these files. Suppose that the job script to execute the application at a large width is as follows (the line numbers at the left are for annotation only; they do not appear in the actual script file):

```
1     #!/bin/bash
2     #PBS -S /bin/bash
3     #PBS -l mppwidth=24000
4     #PBS -l mppnppn=24
5     #PBS -l walltime=1:00:00
6     #PBS -o run.out
7     #PBS -j oe
8     #
9     test "${PBS_O_WORKDIR}" != "" && cd ${PBS_O_WORKDIR}
10     aprun -n 24000 -N 24 myapp
11     #
12     # All done
13     #
```

The changes necessary to read the DLFM cache files appear in the modified script below:

```
1     #!/bin/bash
2     #PBS -S /bin/bash
3     #PBS -l mppwidth=24000
4     #PBS -l mppnppn=24
5     #PBS -l walltime=1:00:00
6     #PBS -v DLFM_INSTALL_DIR
7     #PBS -o run.out
8     #PBS -j oe
9     #
10     test "${PBS_O_WORKDIR}" != "" && cd ${PBS_O_WORKDIR}
11     export DLFM_OP=read-cache
12     ${DLFM_INSTALL_DIR}/bin/dlfm.pre ${DLFM_OP} 24000
13     export PYTHONHOME=${DLFM_INSTALL_DIR}
14     aprun -n 24000 -N 24 myapp
15     #
16     # All done
17     #
```

The modified script has four new lines. Line 6 exports the environment variable DLFM_INSTALL_DIR from the shell that submits the job to the shell that runs the job script. Line 11 sets the type of DLFM caching operation to perform (read-cache). Line 12 executes the `dlfm.pre` command to prepare the runtime environment to read the cache files. The command also takes a numeric argument that specifies the width used in the run. The preparation done by `dlfm.pre` includes modifying the cache files with permissions specific for a read-cache run, and striping the files to optimize the process of presenting the cache files to all of the PEs of the large-width run. Line 13 sets the PYTHONHOME environment variable to point to the path where the modules associated with the custom python reside. On line 13 of the modified script, as on line 10 of the original script, the application is executed at a width of 24000 PEs, across the 24 cores of 1000 compute nodes. It is assumed that `myapp` has been built to do DLFM caching, as described in steps 1-2 above; as such, it will read the cache files in the current working directory of the batch job as it runs.

5.0: Using the DLCaching Debugging Feature

The DLFM package is equipped to write debug trace files during DLCaching when specified by the user. The writing of trace files is controlled by the environment variable DLCACHE_DBG. When this variable is set to "on", "true", "yes", or "1", the dlfm.pre command will generate empty trace files, in a subdirectory called `dlcache.dbg` within the current working directory, for each node and core in the job's reservation. The application will then sense the presence of the trace files and will write traces to these files as it performs its dlcached operations. The trace file names will be of the form:

dlcache.dbg.<nodename>.<coreid>

where nodename is the name of the node (as given by `uname(2)`) that hosted the application PE, and coreid is an alphabetic character representing the processor core that hosted the PE (a-z for cores 0-25, A-Z for 26-51).