Shared library performance on Hopper

Zhengji Zhao[1)], Mike Davis[2)], Katie Antypas[1)], Yushu Yao[1)], Rei Lee[1)], Tina Butler[1)]
[1)] National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
Berkeley, CA
[2)] Cray, Inc

E-mail: zzhao@lbl.gov, u3186@cray.com, {kanytpas, yyao, rclee, tlbutler}@lbl.gov

***Abstract*—NERSC's petascale machine, Hopper, a Cray XE6, supports dynamic shared libraries through the DVS projection of the shared root file system onto compute nodes, while it uses the static objects by default otherwise. The performance of the dynamic shared libraries is crucial to some of the NERSC workload, especially for those large scale applications that use Python as the front end interfaces. The work we will present in this paper was motivated by the report from NERSC users, stating that the performance of dynamic shared libraries is very poor at large scale, and hence it is not possible for them to run large Python applications on Hopper. In this paper, we will present our performance test results on the shared libraries on Hopper, using the standard Python benchmark code Pynamic and a NERSC user application code Warp which has Python as the front end interface, and will also present a few options which we have explored and developed to improve the shared library performance at scale on Hopper. Our effort has enabled Warp to start up in 7 minutes at 40K core concurrency.**

Keywords-component; shared library; Python; performance; startup time; DLcache and FMcache;DLFM

I. INTRODUCTION

While it is often the case that shared libraries are not supported on HPC computers for maximum scalability and efficiency, NERSC's peta-flop Cray XE6 machine, Hopper [1], supports shared libraries on its compute nodes through Cray's Data Virtualization Service (DVS) [2]. DVS projects the shared root file system to the compute nodes at user's choice. Therefore, applications that were built dynamically can access the shared libraries at run time. This capability has brought great benefit to NERSC users by accommodating shared library workloads on Hopper. A NERSC user survey conducted in 2011 identified nine major applications that require shared library support on HPC machines [3]. However, while most of the shared library users enjoy this extended feature on Hopper, some users have reported that the applications with shared libraries perform poorly at scale, especially for Python applications. For example, one of our Warp [4] users reported that *"For 64 cores and up, the startup time scales linearly with the number of cores. It is close to 400 seconds for 1,024 cores, which means that a run with 8,000 cores will approach 1 hour of startup time and a run with 64,000 cores might take as much as 8 hours to start."* Here the startup time is the time spent on just importing the Python modules before doing any computations. Although Warp users would like to run Warp at 40K cores on Hopper, the huge startup time has made it impossible for them to run the code at that large scale. In the past, Warp users had to build a static binary to run on our Cray XT4 (Franklin) machine where there was no shared library support, and they were able to run the code using up to 10K cores. However, building full-featured Python applications statically requires tremendous effort because there are a huge number of dependent libraries and packages, and often requires modifying Python source codes. This had made code maintenance a non-trivial task for users whenever there were upgrades and bug fixes in user codes, Python and other dependent software as well as the OS.

The poor performance of the Python application on Hopper is not a Cray XE6 specific issue; the same problem would exist on any machines, eg., generic Infiniband Linux machines, if running Python applications at this scale. Cray's implementation of the shared library support on Cray XE6 through DVS introduces an extra layer of client-server software between the application and the storage device, thus may demonstrate an extra layer of overhead at lower scale compared to the generic Linux clusters. However, it is the huge capacity of the Cray machines which has encouraged users to try their applications at a scale that they would never be able to try on a generic Linux machine and has made the problem more acute. Therefore, in order to allow NERSC users to run Python applications at scale on Hopper, both Cray and NERSC are engaged in an effort to reduce the huge startup time of Python applications. In this paper, we will present our work in improving the shared library performance on Hopper. Similar effort has been made at Sandia National Laboratories [5] to support their visualization workload at large concurrency on Cielo, a Cray XE6 at Sandia National Laboratories. By dedicating a file system to store users' shared libraries and by repurposing 50 compute nodes to the IO servers for this dedicated file system, they were able to bring down the startup time of Pynamic (with 495 shared objects in total), a standard Python benchmark code [6], to 30 minutes at 32K core concurrency. Instead of pursuing this user dedicated file system approach, we have tested other options available,

which do not require static partitioning of the compute node pool and converting some of them to the IO servers.

The paper is organized as follows. After this introduction, we will introduce two benchmark codes we used to test the performance of shared libraries on Hopper in Section II. And in Section III, we will present our tests with the five different file systems available on Hopper to find out the optimal file system to store users' shared libraries. In Sections IV and V, we discuss the methods we have tested and developed to reduce the startup time of Python applications. And we conclude the paper with the recommendations to NERSC shared library users with the best practice and the method to improve the shared library performance on Hopper. We note that through out the paper, we will use the term "shared library" to encompass both shared-object libraries (.so files) and Python modules (.py and .pyc) for convenience unless we explicitly separate out the two items. In addition, the term 'core' is taken to be synonymous with 'PE' and 'rank' as all work done in this study is on applications that run MPI-only, versus hybrid mode.

II. Two benchmark codes: Pynamic and Warp

We selected two benchmark codes, Pynamic and Warp, to test the shared library performance on Hopper. Pynamic is a Python benchmark that is designed to test a system's ability to handle the dynamic linking and loading requirements of Python-based scientific applications [6]. Pynamic integrates MPI into the Python interpreter, pyMPI, to run parallel Python applications. The pyMPI can be built with all shared objects being linked in at link time to the pyMPI binary (pynamic-pyMPI) or built as vanilla pyMPI which loads and imports the shared objects and modules at run time. The total startup time (before doing any computation) for Pynamic is defined as the sum of launch time (startup time), module import time and module visit time. The launch time is the time to start up the pyMPI on all the cores, and the module import time is defined as the time to execute the Python import commands, and the module visit time is the time spent on visiting all the modules once. Pynamic can be configured to use any desired number of shared libraries and modules to mimic real Python applications. In our tests, Pynamic was configured with 495 total shared objects, 280 Python import statements and 215 utility libraries (the same configuration used in Ref [5]).

Warp is a multidimensional intense beam simulation program being developed and used by the heavy ion fusion researchers from the three main national laboratories, LBNL, LLNL, and PPPL and other sites. The discrete-particle models in WARP combine the particle-in-cell (PIC) technique commonly used for plasma modeling with a description of the "lattice" of accelerator elements. The code is written primarily in Fortran90 and is parallelized using domain decomposition and MPI. It has a flexible and powerful Python user interface. The parallel Warp code is launched using the parallel Python interpreter, pyMPI. At job start, it loads 48 shared-object libraries and imports 260 Warp modules before any computation starts. Warp users desire to run the code at up to 40K cores; unfortunately, the huge startup time (launch time + module import time) has been preventing them from running Warp at large concurrency.

III. Tests with five File systems on Hopper

Since the performance of the shared libraries depends on the performance of the file system where the shared libraries are stored, we first tested the performance of the shared libraries on each file system in order to find the optimal file system for users to store the shared libraries on Hopper.

There are five file systems available on Hopper which serve different purposes; they are /home, /scratch, /project, /gscratch (global scratch) and /usr/common file systems. The /scratch file system consists of two identically configured Lustre file systems each with 35GB/s of bandwidth and 1.1 PB of capacity. The /scratch file system is local to Hopper and is subject to purge periodically. NERSC advises users to run their data-intensive applications on the /scratch file system. The /home is a GPFS file system that can be accessed from multiple platforms at NERSC. It has 1.5GB/s of bandwidth and 40TB total storage space that is shared by more than 5000 users. NERSC advises users to use /home to store source codes, binaries, libraries and other data files that users want to save permanently. The /project directory is a GPFS file system with 15GB/s bandwidth and 1.4PB of total disk space. NERSC advises users to use /project to store source codes, binaries, libraries and other data files to be shared among project members. The /gscratch is a GPFS file system with 15GB/s of bandwidth and 1.1PB of total space and shared by most of the platforms at NERSC. Users are advised to run their production runs on /gscratch, which allows sharing of data between different platforms. Lastly, the /usr/common is the place where the software supported by NERSC staff is installed on (users do not have write permission on this file system). It has 2.3 TB of disk space and shared by most of the NERSC systems. Eg., the Python/2.7.1 module is installed on this file system.

Note the NERSC recommendations regarding how to use these available file systems were for the Hopper native environment (running static binaries), which does not require the shared library support on the compute nodes. When our Warp users reported the poor performance of the shared libraries on Hopper, they stored their shared libraries on their /home directory which is a common user practice.

In these file system tests, we used the Pynamic code with its default configuration (495 total shared objects). Fig. 1 shows the Pynamic startup time on the four file systems that users can store their own shared libraries and Python modules on. Fig. 2 shows the same timing for the /usr/common file system where the NERSC supported software is installed on. User applications may access this directory at runtime (users have read and execution permissions only to this file system). In each file system test, we compared the startup time of Pynamic for two use scenarios: enabling the shared root file system by setting

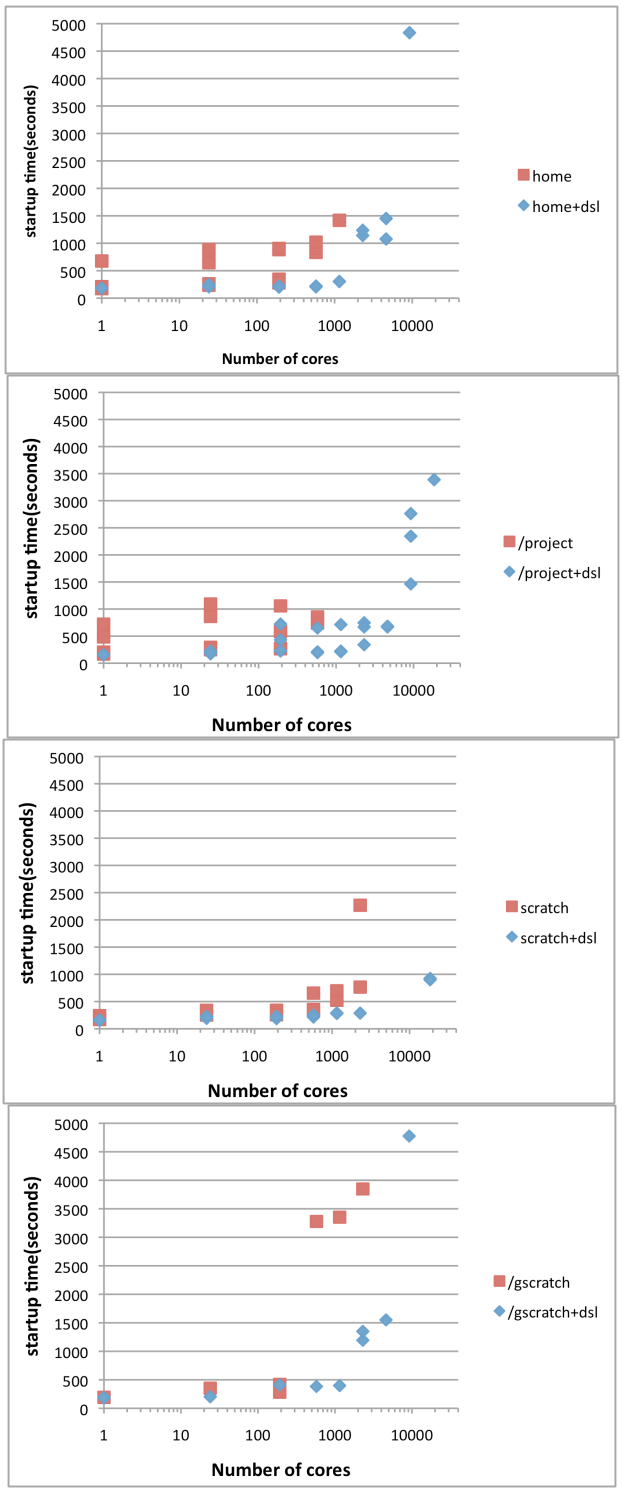

Figure 1. Pynamic startup time with (blue) and without (red) using shared root file system when the shared libraries and Python modules of Pynamic reside on the /home, /project, /scratch and /gscratch file systems respectively.

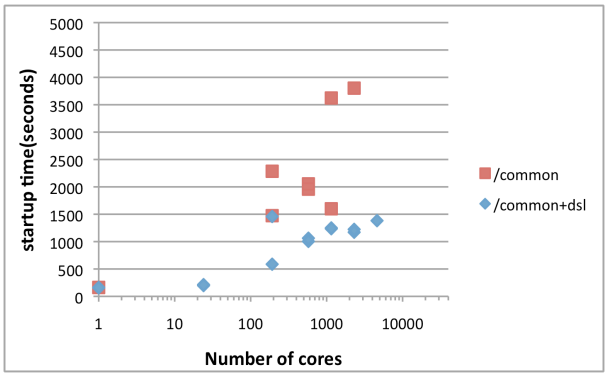

Figure 2. Pynamic startup time with (blue) and without (red) using shared root file system when the shared libraries and Python modules of Pynamic reside on the /usr/common file system.

CRAY_ROOTFS=DSL at runtime (blue in Fig. 1 and Fig. 2) and running Pynamic in the unsupported way (as we had to do on our Cray XT4 machine where the shared library feature was not supported by Cray) by storing all the shared libraries that Pynamic needs completely in the users' own directories. In the latter case, we need to provide a local copy of the /usr/lib64 and /usr/lib etc. in the users' own directories to make sure all the dependent shared libraries can be accessed at runtime. This is to see how much the shared library support Cray provides on Hopper through the scalable shared root file system has helped the shared library applications. Our tests show that on each file system, the Pynamic startup time is lowered significantly by using the shared root file system compared to not using the shared root file system on Hopper (the unsupported way of running shared applications on Hopper). And without the shared root, Pynamic could not run beyond 2048 cores on all the five file systems. Jobs beyond this concurrency hung or ran into errors. We believe this is some scalability issue with the current DVS implementation (which Cray is actively workin on) that all file systems, exept the Lustre /scratch file system, depend on on Hopper. Among the five file systems, the shortest startup time at large scale was observed when the shared libraries and Python modules were stored in the /scratch file system and the shared root file system was enabled.

With this knowledge, we ran the user code Warp by storing its shared libraries and Python modules in the /scratch file system. Fig. 3 shows the startup time of Warp. Compared to the original user prediction (they stored the shared libraries on the /home directory), the Warp code can start up around four times faster at 36K cores when storing shared libraries in /scratch versus /home.

One problem with using /scratch is that the startup time may be subject to a large fluctuation due to the heavy contention from other users.

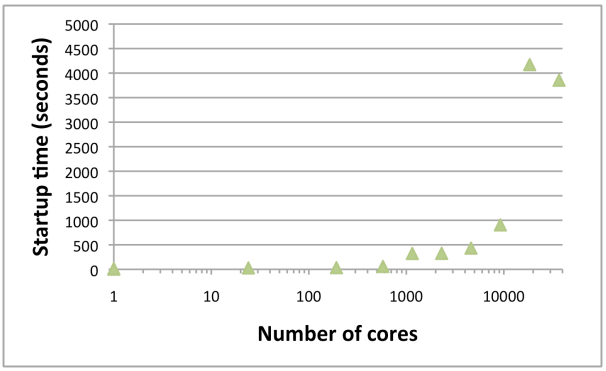

Figure 3. WARP startup time when the shared libraries and Python modules were stored in the /scratch file system and with the shared root enabled.

IV. TESTS WITH EXECUTION PATHS

Cray has been actively working on improving the shared library performance on their HPC machines. In addition to being engaged in improving the DVS performance that the shared libraries rely on, they have also changed the default search paths to the shared libraries in the programming environment in order to improve the shared library performance. In the past, RPATH was included in the dynamic executables with a path to the libraries to use at execution time. Now, at execution time, symbolic links to these libraries will be found in the single directory /opt/cray/lib64 using ld.so.cache which has the effect of cutting short the number of the search paths for the dynamic libraries they support. This should improve startup performance of dynamically-linked applications by reducing the time spent on traversing multiple paths to search for the needed shared libraries.

Fig. 4 shows the Warp startup time using Cray's new approach of using /opt/cray/lib64 as the path to the shared libraries (blue) and using the RPATH approach (green). Note the startup times at 36K cores from two methods are overlapped. One can see that with this new approach, the startup time of Warp does not show an observable improvement.

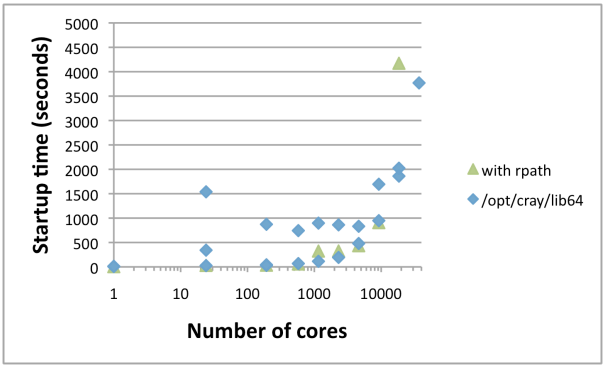

Figure 4. Comparison of Warp startup time when shared library search defaults changes.

Cray's default search path changes only affect the libraries Cray provided, so we tested the same idea for the Warp's own shared libraries. Warp uses 48 shared-object libraries distributed among the directories, so instead of allowing the code traverse a number of directories to search for these libraries, we stored as many shared libraries as possible in a single directory (hence the LD_LIBRARY_PATH is shorter) and tested its startup time. Fig. 5 shows the results. Compared to the use of the /opt/cray/lib64 (red), using a shorter LD_LIBRARY_PATH does not seem to bring an observable improvement in the Warp startup time. It should be noted that the optimizations discussed in this section, while possibly impacting the time spent loading the 48 shared-object libraries, do not address the cost of importing the 260 warp modules.

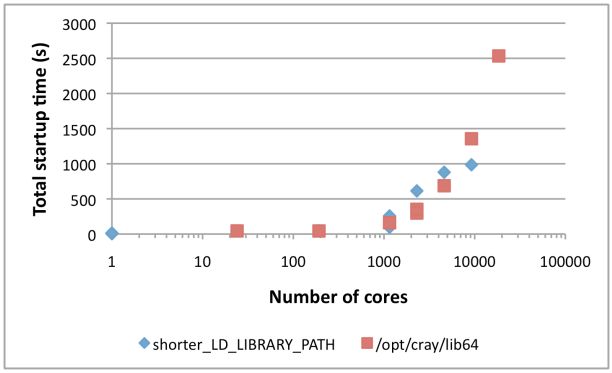

Figure 5. Warp startup time with a shorter LD_LIBRARY_PATH. Tests were done on top of the Cray default search paths change.

These tests suggest that without addressing the underlying mechanisms that Python applications rely on, it is difficult to address the slow startup time for Warp, especially on a production environment where there is heavy contention for the IO resouces from other users on the system. When a parallel Python application starts up (loading .so files and importing Python modules), all cores perform the same amount of IO operations, such as open(), stat(), read(), etc. During Warp code startup, each core performs 3388 opens (495 successful), 1632 stats (848 successful), and 500 reads which pound the metadata server of the /scratch file system heavily as shown in Fig. 6. Fig. 6 shows the high metadata server IO activities (only the open() operations shown in the Figure) while an 18K core Warp job was running. Without reducing the number of IO operations associated with Python applications, the startup time would not be easily addressed.

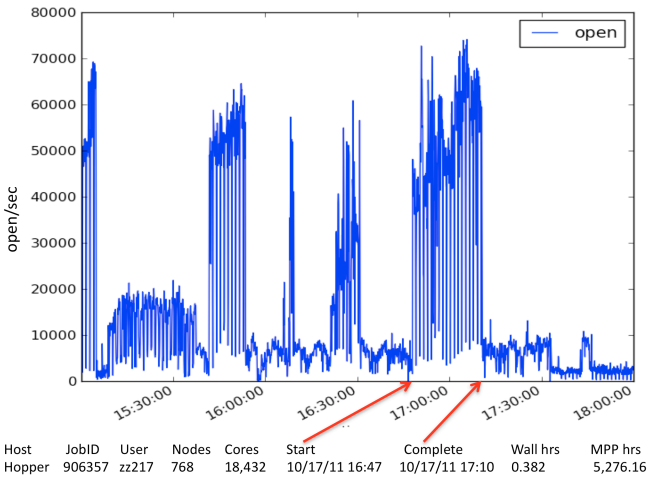

Figure 6. Lustre metadata server activities are high during large concurrency Warp job run. This figure was generated by Andrew Uselton's LMT tools [7].

V. DLCACHE AND FMCACHE (DLFM)

As part of the effort to optimize the shared-library operations being performed by the WARP application, two separate software packages were developed [8]. The first, DLcache, optimizes the importing of shared-object libraries (.so files). The second, FMcache, optimizes the importing of Python modules (.py and .pyc files). As mentioned in the Introduction, the term "shared library" is used to encompass both shared-object libraries and Python modules in the discussion. A Python-based application can be built to use either of these packages alone, or both together. An application that uses dynamic linking and dlopen, but does not use Python, can benefit from using DLcache alone. For this optimization effort, the WARP application was built to use the two packages together.

It was deemed best to use two separate packages to handle these two types of shared-library use. The Python library uses two separate mechanisms to perform imports. The import of a shared-object library uses a libdl function called dlopen, whereas the import of a Python byte-code module uses an internal libpython function called find_module. The find_module mechanism is specific to Python, and can benefit from FMcache, whereas the dlopen mechanism is used in many kinds of applications besides Python, and can benefit from DLcache. Both packages use a similar strategy to optimize the I/O operations associated with shared-library use. This strategy involves executing the application in a "trial mode" on a small number of cores (or PEs or MPI ranks) to produce a single file that contains a consolidation of all of the shared objects loaded by the application. The application can then be executed again "for real" on a large number of cores, and can be directed to read from the consolidated file instead of searching for and accessing the numerous shared-library files. Thus, a part of the optimization comes from reducing the number of metadata operations involved in searching for, opening, and closing many files on each of many cores.

Another part of the optimization comes from caching the consolidated file so that the number of cores actually doing I/O to external storage is greatly reduced. In the case of DLcache, the consolidated file is cached in the RAM-resident /tmp file system on each compute node hosting the application. In the case of FMcache, the consolidated file is read from external storage by the application's rank-0 process, and broadcast to the other cores via MPI. The two optimization packages do not use the same caching strategy because, in the case of FMcache, the MPI strategy is cleaner and more optimal than the /tmp strategy, and in the case of DLcache, the optimization is called into play during application startup, before MPI is initialized.

The optimization strategy depends on three central assumptions: (1) the application can be executed at either small or large width; (2) the number of shared libraries to be loaded, and their order, does not change from small- width to large-width runs; and (3) the number of shared libraries to be loaded, and their order, is identical across cores. In the case of WARP, these central assumptions are valid.

We applied the DLMF methods to Warp, and showed the startup time in Fig. 7 (blue). And for comparison, the original Warp startup time using /opt/cray/lib64 (blue dots shown in Fig. 4) was also included in the same figure (red). One can see that with the DLFM methods, Warp can start up in around 7 minutes at 40K concurrency! This is about a ten-fold speedup compared to the original startup time (shown in Fig. 4) at 36K concurrency (we don't have the startup time for 40K core runs for original method). If we compare the results of the DLFM methods to the original user prediction based on the startup time they observed (when the shared libraries were stored in the /home file system), the speedup is about 42 times! Now Warp users should be able to run the code at their desired concurrency (40K) on Hopper.

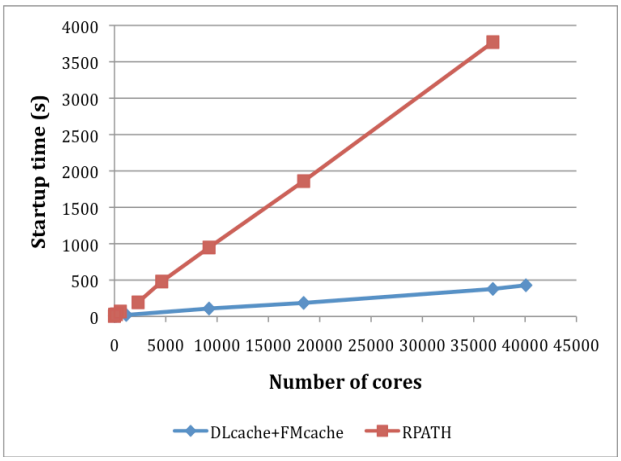

Figure 7. Warp startup time compariosn between the DLFM methods (blue) and the original method (after Cray exeution path change) (red).

In Fig. 9, the startup time breakdown between the load and the module import time of the DLFM methods is shown

with a stacked bar graph (the blue bars show the load time, and the purple bars show the module import time). And for comparison, we also showed the same data for the original method (the results shown in Fig. 4) in Fig. 9 as well. One can see that total speedup of DLFM methods are the effects from both the DLcache and FMcache methods. The load time is comprised entirely of DLcache operations. The import time is comprised of both DLcache and FMcache operations, since Warp imports both shared-object libraries and Python modules. The steeper increase in import time with increasing core count is likely due to the FMcache read-broadcast strategy, which comes into play on every import, whether the object being imported is large or small. There are definitely opportunities for further optimization here, though they may involve a cost in compute-node memory.

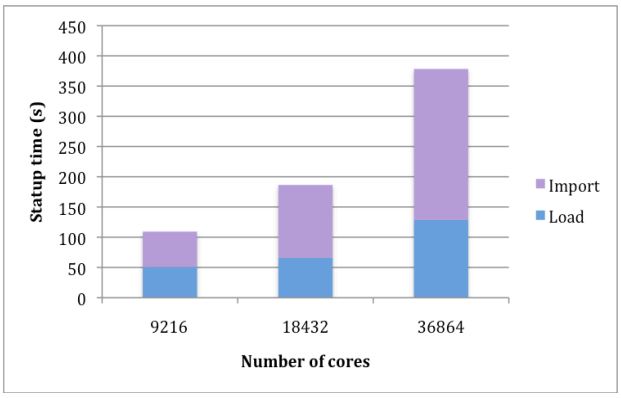

Figure 8. Warp startup time breakdown between the load and module import time when the DLFM methods were used. Where the blue bars show the load time and the purple bars show the import time.

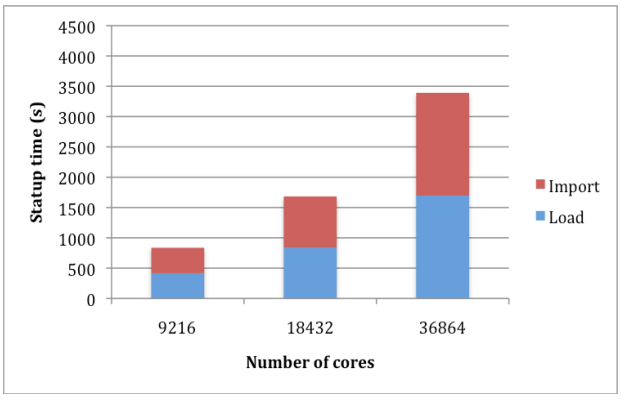

Figure 9. Warp startup time breakdown between the load and module import time when the /opt/cray/lib64 was used as the default search parth for the shared libraries. Where the blue bars show the load time and the purple bars show the import time

We note that due to the limited /tmp space (500MB) configured on Hopper, we were not able to test the DLFM methods with the default configuration of Pynamic (495 total shared objects), for which the total size of shared objects is 1.39GB.

VI. Conclusion

To provide a workaround for the huge startup time for Python applications on Hopper, we tested shared library performance on Hopper using two selected benchmark codes, a standard Python benchmark code, Pynamic and a NERSC user provided code called Warp. Through the Pynamic performance tests on each of the five file systems available on Hopper, we have identified the Lustre scratch file system as the optimal file system for users to store shared object files (.so files) and Python modules (the.py and .pyc files). This has brought a four-fold speedup compared to the original user prediction when storing Warp libraries and Python modules on the /home file system. We tested Cray's new shared library default path (/opt/cray/lib64) compared to the original RPATH approach deployed on Hopper, and we also tested a shorter LD_LIBRARY_PATH, and found no observable startup time reduction for this specific application code (Warp) which has a relatively small number of shared object files (48) to load compared to the number of python modules (260) to import. The DLcache and FMcache methods developed by one of the authors [Davis] are able to reduce both the shared object file loading time and the Python module import time effectively by reducing the number of the metadata IO operations and their contention, as well as reducing the number of cores doing IO operations. Our effort has brought a ten-fold speedup for Warp startup time at 36K core concurrency compared to the /opt/cray/lib64 directory as the shared library default search path, and has allowed Warp to start up in 7 minutes at 40K core concurrency.

Our tests suggest that when users run shared applications at scale on Hopper, they should store their shared libraries and/or Python modules on the /scratch file system, and adopt the DLcache and FMcache methods in their large scale runs where applicable to achieve a tractable startup time on Hopper.

VII. Acknowledgement

The authors would like to thank Sue Kelly at Sandia National Laboratories who helped us getting started with this work and bridged the collaboration between NERSC and Cray. The authors also thank Sean Zhu, a NERSC summer student in 2011, who was involved with the early work of this project. Authors also thank NERSC user David Grote (WARP developer), and Jean-Luc Vay who motivated us to investigate the performance issues with the shared libraries on Hopper. David Grote provided the WARP code and instructions to build and run the code, and was involved with the DLcache and FMcache tests. Authors would like to thank Nick Wright, Andrew Uselton, Thomas Davis, David Skinner, Harvey Wasserman and other NERSC staff for their valuable discussions and help. This work was supported by the ASCR Office in the DOE, Office of Science, under contract number DE-AC02-05CH11231. It

used the resources of National Energy Research Scientific Computing Center (NERSC).

REFERENCES

[1] http://www.nersc.gov/users/computational-systems/hopper/

[2] http://docs.cray.com/books/S-0005-4003/S-0005-4003.pdf

[3] Richard Gerber, The Case for Shared-Library Support in High Performance Scientific Computing, NERSC internal report, April, 2011.

[4] http://hif.lbl.gov/theory/WARP_summary.html/

[5] Suzanne M. Kelly, Ruth Klundt and James H. Laros III, Shared Libraries on a Capability Class Computer, CUG metting, May 23-26, 2011, Alaska, Fairbanks.

[6] http://computation.llnl.gov/casc/Pynamic/pynamic.htm

[7] Andrew Uselton, The Performance Monitoring Archive, https://portal-auth.nersc.gov/project/pma/

[8] Mike Davis, Using the DLcache and FMcache methods for large scale Python applications and shared library applications, http://www.nersc.gov/assets/userguide.txt/ http://www.nersc.gov/users/software/development-tools/python-tools/