Evaluating Shifter for HPC Applications

Donald Bahls
Research & Development
Cray, Inc
St Paul, MN, USA
e-mail: dmb@cray.com

Abstract— Shifter is a powerful tool that has the potential to expand the availability of HPC applications on Cray XC systems by allowing Docker-based containers to be run with little porting effort. In this paper, we explore the use of Shifter as a means of running HPC applications built for commodity Linux cluster environments on a Cray XC under the Shifter environment. We compare developer productivity, performance, and scaling of stock applications compiled for commodity Linux clusters with both Cray XC tuned Docker images as well as natively compiled versions outside of the Shifter environment. We also discuss pitfalls and issues associated with running non-SLES-based Docker images in the Cray XC environment.

Keywords: Docker, Shifter, Cray XC, Cray XE/XK

I. INTRODUCTION

In the past several years there has been considerable excitement surrounding Linux container packages such as Docker that facilitate the transfer of self-contained applications from one Linux environment to another without the overhead of virtual machines. These packages make use of *cgroup* support in the Linux kernel that allows it to isolate sets of processes and place limits on various system resources, thereby resulting in a much smaller memory footprint than the same application run within a virtual machine [1]. With the addition of Shifter to the Cray XC and XE/XK product lines, a subset of the functionality provided by Docker is now available to users of Cray systems.

In this paper, we look at the implications of using containers on the Cray platforms by comparing the timing of several applications compiled natively for the environment to applications contained within Docker-based images using two techniques that are capable of taking advantage of the Cray HSN (High Speed Network). These techniques focus on MPI-based applications, which constitute a large percentage of traditional HPC workloads.

In the first technique, a portion of the Cray software stack is added to the image to provide native support for the HSN using standard libraries. This technique is referred to in this paper as a *Cray MPI UDI* (User Defined Image). The second technique makes use of the open source MPICH distribution to construct an image. At Runtime, shared libraries from the TCP-based MPICH version are replaced with the corresponding Cray MPI version of the MPICH-ABI libraries allowing the application to run natively on the Cray HSN. This technique is referred to in this paper as an *MPICH-ABI UDI*. We discuss the developer productivity aspects of creating a Docker-based image as it compares to the native environment. The runtime performance of a select number of applications is also considered. Several synthetic benchmarks are run to compare native IO and HSN performance to same metrics for UDI based techniques.

II. BACKGROUND AND SETTINGS

Shifter is a set of tools that provide the means to make use of a subset of the functionality of Docker images in an HPC environment. The Shifter environment provides an end-user with a mechanism to pull a Docker image to the system for use on compute resources. After the image is retrieved from an external source (e.g. Docker Hub, hub.docker.com), the contents of the image are extracted and converted into a file system image by a gateway daemon running on a Service node. This file system image and its related resources are referred to as a UDI. At job run time, the user selects which UDI should be used. Prior to application launch, the UDI is mounted on the compute nodes assigned to the job and then performs a `chroot` to the alternate file system root contained within the image. This environment includes bind mounts of various file systems required for Linux to function correctly as well as site- and possibly user-defined mounts. As with traditional Linux container environments, the only running kernel is that of the underlying host operating system. To increase the security of Shifter, all processes are executed as the normal user running the batch job and all mounts within the UDI are mounted with the *nosuid* option at run-time [2]. The *nosuid* mount option is used to ignore set-user-identifier and set-group-identifier options on binaries within the UDI, thereby removing one avenue for privilege escalation. Shifter support was added to the Cray CLE-5.2UP04 release in early 2016 via several patches, and it included support for the Cray XE/XK and Cray XC platforms.

As with any new technological additions to the HPC environment, it is important to understand the performance and usability implications of the new technology. This paper does just that by evaluating: end-user productivity— what is required to get an application running on the system; application performance— how do applications using Shifter perform relative to an application compiled natively for the system; scaling— how well does the new technology

scale to higher numbers of processors relative to native versions of the same code.

A. *Operating Environment for Performance Tests*

Two systems were used to execute application performance tests. These systems consisted of a Cray XC system with a mixture of Intel Ivybridge (IVB) processors as well as an older Cray XE/XK system with a mixture of AMD Interlagos and Abu Dhabi processors (Table I).

TABLE I. SYSTEM CONFIGURATION

Host	Node Configuration	Other Details
Cray XE6/XK7	100- 16 core/2.1 GHz AMD Interlagos 280- 32 core/2.1 GHz AMD Interlagos 96- 32 core/2.5 GHz AMD Abu Dhabi	PBS Professional 12.2.204; Direct Attached Lustre; CLE-5.2UP04 Gemini Network
Cray XC30	116- 20 core/3.0 GHz Intel IVB 116- 24 core/2.7 GHz Intel IVB 20- 24 core/2.7 GHz Intel IVB	Moab 8.1.1.2 & Torque 5.1.1.2; Sonexion 2000 / NEO-2.0.0 CLE-5.2UP04 Aries Network

Systems from the venerable Cray XE/XK line were included because many systems of that type are still in operation today, and because Shifter support was introduced for both the Cray XE/XK and Cray XC platforms.

B. *Application Selection*

We use a mixture of real-world applications and synthetic benchmarks are used to characterize the performance of Shifter image-based applications using several different techniques. The performance of these applications is compared to the performance of the same application built directly on the Cray system using the Cray Programming Environment. The HPC applications run include: PISM-0.7[1], Quantum Espresso 5.3.0[2], and POP2[3]. The applications were selected to provide a sampling of common HPC libraries including: BLAS, FFTW, GSL, LAPACK, NetCDF, PETSc, and ScaLAPACK as well as a mixture of C, C++ and Fortran code (Table II). These applications were also selected because each had an existing port for the Cray environment. This ensured there would not be excessive effort required to compile the application to capture baseline performance for the Cray environment. Each application test case is run at different core counts to allow for scaling comparisons. Two synthetic benchmarks are also considered to characterize IO performance and network bandwidth; those applications are IMB-3.2[4] and IOR-2.10.3[5]. Ubuntu is used as a base OS for the UDI images due to good availability of scientific libraries for that Linux distribution. The GNU compiler is used to compile applications within Docker images to avoid commercial license constraints that might limit distribution of the images to Docker Hub.

TABLE II. APPLICATION BUILD DEPENDENCIES

Application	Libraries/Build Dependencies
PISM (C++ based code)	BLAS, cmake, FFTW3, GSL, LAPACK, MPI, NetCDF, PETSc
POP2 (Fortran90 based code)	MPI, NetCDF
Quantum Espresso (Fortran90 / C based code)	BLAS, FFTW3, LAPACK, MPI, ScaLAPACK
IMB (C based code)	MPI
IOR (C based code)	MPI with MPI/IO support

Application compilation within the native Cray environment uses CCE, GNU and PGI compiler suites depending upon the application. Basic optimization flags (e.g. `-O3`), including processor specific tuning options, are used for applications.

III. TECHNIQUES USED TO BUILD IMAGES

Two methods are used to build applications within the Docker environment. These approaches were selected because preliminary work done within the Shifter environment suggested that each of these methods could deliver native HSN performance for MPI-based applications.

A. *MPICH-ABI UDI Method*

The *MPICH-ABI UDI* method takes advantage of MPICH-ABI support provided by the Cray MPI stack. During compilation the application is linked against a copy of MPICH supporting the MPICH ABI interface [3]. At runtime the `LD_LIBRARY_PATH` environment variable is set to point to the Cray MPI version of the MPICH-ABI library and dependencies, to the replace the MPI libraries in the image. This allows an application compiled with MPICH for a TCP network to make use of the Aries (or Gemini) network and get native HSN performance. For the experiments done herein, the shared libraries were copied at job launch to the local Lustre file system. This was done because the site-specific mount script did not mount the `/opt/cray` directory containing the necessary shared

[1] Parallel Ice Sheet Model-http://pism-docs.org/wiki/doku.php
[2] Quantum Espresso - http://www.quantum-espresso.org/
[3] Parallel Ocean Program - http://www.cesm.ucar.edu/models/cesm1.0/pop2/
[4] Intel MPI Benchmarks - https://software.intel.com/en-us/articles/intel-mpi-benchmarks
[5] IOR - https://github.com/LLNL/ior

libraries, and because the current Shifter batch prologue integration does not provide a mechanism to specify runtime mount options. The CLE-5.2UP04 release of the `shifter` binary does provide a mechanism to specify alternate mounts, but cannot be used more than once per node in this release.

B. Cray MPI UDI Method

The *Cray MPI UDI* method uses a technique developed by the authors of Shifter in which a portion of the Cray libraries in `/opt` are saved from the Cray system into a tarball then imported into a Docker image. The application within the image is compiled and linked using the Cray MPI libraries and other dependent libraries. Directories containing MPI shared libraries and dependencies are added to the `/etc/ld.so.conf`, and the `ldconfig` command is run to update the `/etc/ld.so.cache`[4].

While this method results in native HSN network performance for the application, one key factor makes it a less appealing option. The addition of Cray libraries and header files to the Docker image results in Cray intellectual property being added to the image, making it illegal to upload the resultant image to the public Docker Hub. This issue can be avoided if a user has access to Docker on the Cray and can build and import the image directly to the system. Access to the Docker daemon has a number of security implications, so sites should treat this access at they would system *root* access. This consideration limits the general applicability of this method.

It should be noted that the *MPICH-ABI UDI* and *Cray MPI UDI* methods are both dependent on the Cray Programming Environment libraries to provide native HSN support. This means that there are some limitations on compiler and library compatibility. The Cray Programming Environment has support for particular versions of Intel and GCC (GNU Compiler Collection) compiler, and typically the versions are fairly new. This could potentially limit the applicability of these two techniques in the case where the OS used for a Docker image is based upon a much older GCC version.

IV. Productivity Impacts

Productivity improvements are one possible outcome of moving to a Docker image-based application. In practice, there is some additional effort involved in generating the *MPICH-ABI* based Docker images for the applications selected due to several factors. While there are MPICH-3.2 *dpkg* packages available for Ubuntu and other OS flavors, the shared library names do not match the MPICH ABI naming scheme, so the Cray MPICH ABI libraries cannot be swapped in at runtime. This issue can be worked around by building MPICH in the image, but that does add some overhead. Because an alternate MPI build is used, several dependent libraries need to be compiled within the Docker image (e.g. ScaLAPACK, PETSc). If MPI support were needed for other libraries (e.g NetCDF, FFTW, etc.) this would add to the set of locally compiled libraries. These factors add slightly to the complexity of constructing the Docker image, but are certainly not insurmountable. The use of MPICH does allow for a fairly straightforward build process that is easy to automate through the use of Docker build files (i.e. *Dockerfile(s)*). One potential drawback to using this method is that while MPICH works quite well for the Cray environment, it is an uncommon MPI stack on Infiniband-based (IB) systems. Newer versions of the Intel MPI stack support the MPICH ABI [5] and include IB support, so it is possible that Docker images based on MPICH could also take advantage of IB infrastructure (this option was not explored further).

The method used to construct *Cray MPI UDIs* whereby contents are imported from `/opt` into the image reduces the number of dependent libraries that need to be compiled, but has several drawbacks. The addition of `/opt` contents to an image substantially increases the size of the image. Several images were over 3 GB with the addition of this content, whereas all of the *MPICH-ABI* images were under 1 GB. While larger images do not appear to significantly affect startup performance, they[6] do take longer to transfer from one location to another. Additionally, the inclusion of Cray `/opt` to the image introduces the aforementioned legal encumbrance.

In one instance, the MPI Fortran 90 module file from the Cray MPI stack was not usable with the OS version of the compiler due to differences in the `gfortran` version and the available MPI libraries. This issue was worked around by using a newer Ubuntu version that had a GCC compiler version closer to the version supported by the Cray MPI library (an alternate Cray MPT version may have also had the correct GCC version support).

The process of linking the required MPI libraries and dependencies is a bit more involved. Rather than attempt to change the build flags for the application significantly, compiler wrapper scripts were generated with the appropriate compiler options. The "`cc -dynamic -craype-verbose`" command was run on the target system to generate the list of compiler flags. While this option shows excellent network performance, linking with the Cray MPI stack limits the portability[7] of the application to other system types by adding a network dependency.

All of the applications profiled have working ports for Cray hardware, which resulted in straightforward builds for the native Cray environment. A vast majority of the library dependencies used by the applications are available from the Cray Programming Environment, resulting in low overhead

[6] It should be noted that non-essential content such as compilers, static libraries, source code, and object files, etc. was not removed from Docker images, and thus the image sizes are certainly high water marks.

[7] The Cray MPI stack supports the MPICH-ABI as well, so it should be possible to compile with MPICH-ABI support enabled to provide portability to other runtime environments. This option was not explored for this paper.

for building on the Cray systems. If applications with less standard dependencies had been selected, the effort to build on the Cray systems would have undoubtedly been more substantial.

Both Docker-based methods were fairly straightforward to use, with the *MPICH-ABI UDI* and *Cray MPI UDI* taking only minor batch script changes from the Cray native version. These changes primarily involved setting `LD_LIBRARY_PATH` appropriately and setting environment variables necessary for Shifter at runtime. It was observed, however, that runtime issues were more challenging to debug within the UDI environment. While it was possible to capture core dumps of applications running within the Shifter environment, shared library differences make it more complicated to interrogate the core dumps using gdb.[8]

A. Quantum Espresso Application Builds

Experiments done with Quantum Espresso are run using four different binary builds on the Cray XE/XK system. The first build is an MPICH-ABI UDI build using Ubuntu-16.04. This version uses gcc-5.3.1 as the compiler with MPICH-3.2, ScaLAPACK-2.0.2 and espresso-5.3.0 built within the Docker environment. The remainder of the library and compile-time dependencies are provided by the distribution. The Cray MPI UDI build is also based upon Ubuntu-14.04 using Cray MPT-7.3.2 for the MPI stack in place of MPICH-3.2. Compiler wrapper scripts are added to the environment to point to the appropriate include files and libraries to use Cray MPT. The two native builds of Espresso use CCE-8.4 and MPT-7.3.2 with the remainder of the dependencies being satisfied by the Cray Programming Environment. One version is built dynamically (the default for a GPU based system). The other CCE-based build is compiled statically. In both cases the *craype-interlagos* module was loaded to enable AMD Interlago-specific optimizations.

B. POP2 Application Builds

Experiments done with POP2 are run using five different binary builds on the Cray XC system. The first build is an MPICH-ABI based build using Ubuntu-14.04. This version uses gcc-4.8.4 as the compiler with MPICH-3.2 and POP2 built within the environment. The remainder of the library and compile-time dependencies are provided by the distribution. A second MPICH-ABI based image adds the compilers flags "`-O3 -march=corei7-avx`", but is otherwise the same as the first MPICH-ABI build. The Cray MPI UDI build is also based upon Ubuntu-16.04 using Cray MPT-7.3.2 for the MPI stack in place of MPICH-3.2. Compiler wrapper scripts are added to the environment to point to the appropriate include files and libraries to use Cray MPT. This version also uses the compiler flags "`-O3 -march=corei7-avx`". One native build of POP2 uses PGI-15.3.0 and MPT-7.3.2 with the remainder of the dependencies satisfied by the Cray Programming Environment. The other native POP2 build uses gcc-5.3.0 and MPT-7.3.2 with dynamic libraries. The remainder of the application dependencies are satisfied by the Cray Programming Environment. In both cases the *craype-ivybridge* module was loaded to enable Intel Ivybridge-specific optimizations.

C. PISM Application Builds

Experiments done with PISM are run using three different binary builds on the Cray XC system. The first build is an MPICH-ABI based build using Ubuntu-14.04. This version uses gcc-4.8.4 as the compiler with MPICH-3.2 and PISM-0.7.2 built within the environment. The remainder of the library and compile-time dependencies are provided by the distribution. The Cray MPI UDI build is also based upon Ubuntu-14.04 using Cray MPT-7.3.2 for the MPI stack in place of MPICH-3.2. Compiler wrapper scripts are added to the environment to point to the appropriate include files and libraries to use Cray MPT. The native build of PISM uses gcc-5.3.0 and MPT-7.3.2 with dynamic linking. A locally compiled copy of GSL was installed, however the remainder of the dependencies are satisfied by the Cray Programming Environment. The *craype-ivybridge* module was loaded to enable Intel Ivybridge-specific optimizations. Attempts were made to compile using CCE however the application build system includes GCC-specific compiler flags, which result in compilation failure. Attempts to statically link were also inhibited by compiler warnings that are forced to be failures by the build system.

V. Application Performance

In the end, application performance is a key factor in the assessment of Shifter. If the UDI methods were significantly slower than a native-based method, it might not be a viable option. We ran test cases at various sizes to provide a comparison of runtime at various core counts. This is used as a first comparison between the UDI methods and native applications. A second comparison uses core counts and application runtime to calculate aggregate core hours for each data point. This provides an absolute comparison, with the minima being the optimal job size from the set of job sizes run. This metric is used rather than speedup because none of the test cases showed strong scaling. A last metric considered for application performance is job startup overhead. Since the two UDI methods require that the image be mounted on all compute nodes assigned to a job, there is conceivably extra overhead for that activity. The startup overhead is calculated as the

[8] It is possible that debugging could be done directly within the image if gdb had been installed within the image. Application debugging using Totalview or DDT was not attempted.

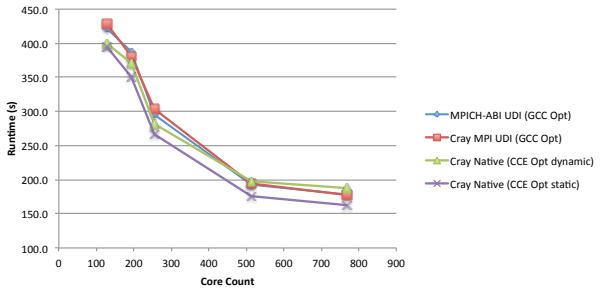

Figure A. Quantum Espresso Runtime – AUSURF112 Test Case

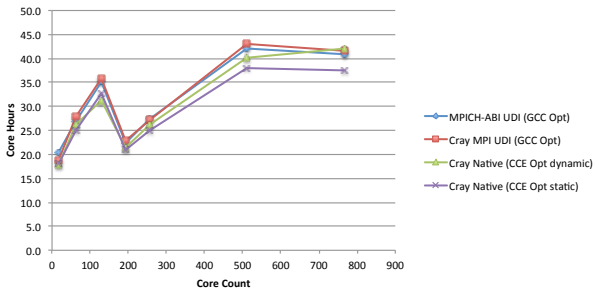

Figure B. Quantum Espresso Core Hours – AUSURF112 Test Case

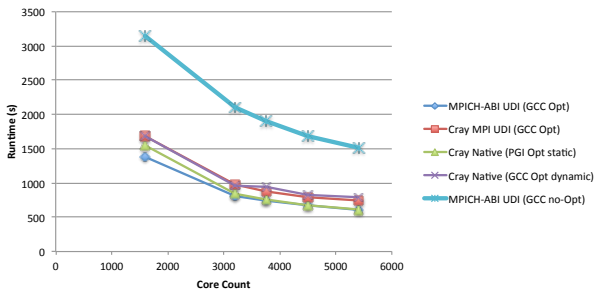

Figure C. POP2 Runtime – 30-Day Test Case

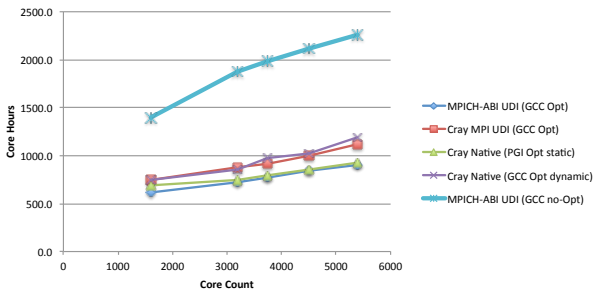

Figure D. POP2 Core Hours – 30-Day Test Case

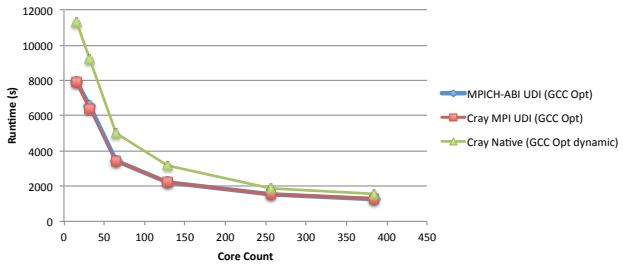

Figure E. PISM Runtime– 1000 Year Test Case

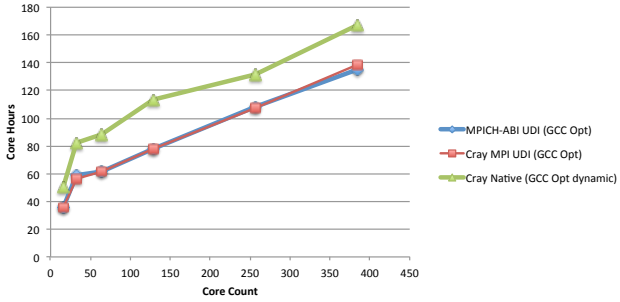

Figure F. PISM Core Hours– 1000 Year Test Case

difference between a time stamp captured within the batch script just prior to launching the application and the *start* unix time stamp saved in the PBS Professional and Torque job accounting records. For this reason all startup times are represented as integer second values.

A. Native Performance Characterization

In the case of Quantum Espresso (Figure A) and POP2 (Figure C), the native compiled applications show the best performance at nearly all test sizes. In the case of POP2, the native 1600 core test case was slightly slower than the MPICH-ABI counter part. In the case of PISM, the native build does not result in the best performance. Native applications show the least average startup time for all three applications (Figures G, H, I).

B. MPICH-ABI UDI Performance Characterization

In the case of PISM (Figure E) the MPICH-ABI UDI and Cray MPI UDI methods show optimal performance at all measured data points. In the case of POP2 (Figure C), the MPICH-ABI UDI application shows performance quite close to the native application when CPU optimizations are enabled with the 1600 core case running slightly faster than the native counter part. The Quantum Espresso case (Figure A) shows the MPICH-ABI UDI method is slightly slower than the statically linked native application. The runtimes for the MPICH-ABI UDI roughly mirror the dynamically linked native application. The MPICH-ABI UDI method showed the highest average startup time (Figures G, H, I). This is likely partially attributable to the workaround of copying libraries to Lustre in order to take advantage of the MPICH-ABI libraries within the UDI at runtime.

C. *Cray MPI UDI Performance Characterization*

In the case of PISM (Figure C) the Cray MPI UDI and MPICH-ABI UDI methods show optimal performance at all measured data points. In the case of POP2 (Figure B), the Cray MPI UDI application shows performance slightly slower than the native application. The Quantum Espresso case (Figure A) shows the Cray MPI UDI method is slightly slower than the statically linked native application. The runtimes for the Cray MPI UDI roughly mirror the dynamically linked native application. The Cray MPI method shows slightly lower average startup time than the MPICH-ABI UDI variant (Figures G, H, I).

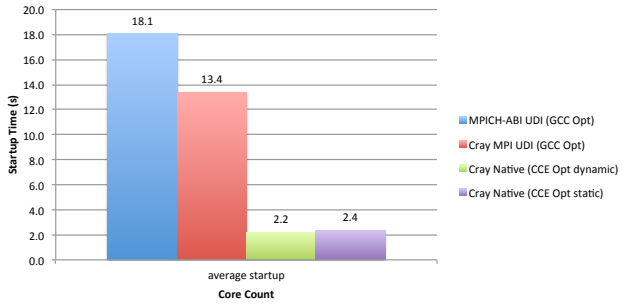

Figure G. Quantum Espresso Average Startup – AUSURF112 Test Case

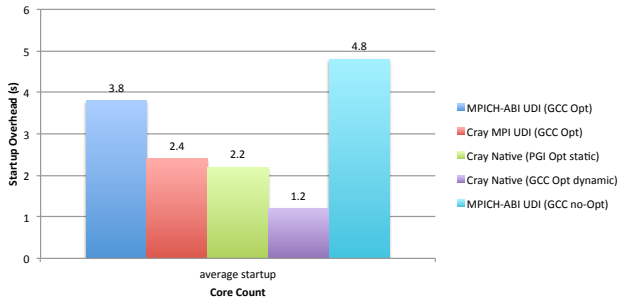

Figure H. POP2 Average Startup – 30-Day Test Case

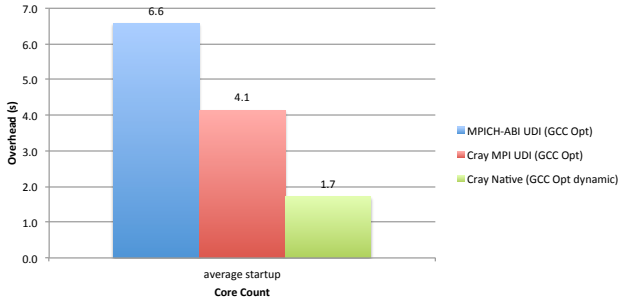

Figure I. PISM Average Startup– 1000 Year Test Case

A. IOR Performance Comparison

IO performance characterizations are made with IOR using 1 to 200 nodes with each node running with 4 IO tasks per node. Each task uses a 1MB transfer size and 4GB output file per task. IOR write and read performance was found to be comparable for the MPICH-ABI UDI environment and a Cray Native using CCE build (Figure J).

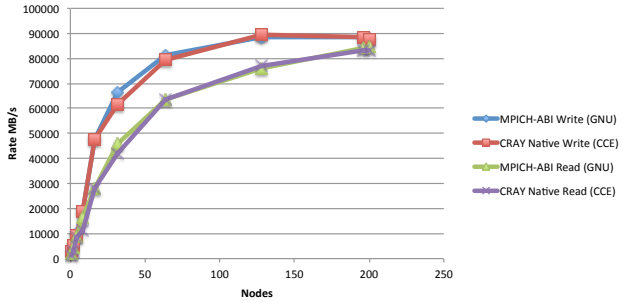

Figure J. IOR Write and Read Performance

B. IMB Performance Comparison

MPI performance experiments are run on 128 nodes using IMB to characterize MPI performance of an MPICH-ABI based UDI relative to a version of IMB compiled with a similar GNU version for the native Cray environment.

The MPICH-ABI UDI results are also compared to a native Cray build using CCE. The latter comparison uncovered some interesting performance discrepancies. With many MPI routines, the performance for the MPICH-ABI application is found to be within a few percent of the Native MPI rate/timing; the routines include: Sendrecv, Exchange, Allreduce, Reduce, Allgather, Allgatherv, Gatherv, Alltoall, Alltoallv, and Bcast. One such example is shown below (Figure K). Several other MPI operations were found to have much larger differences between the MPICH-ABI UDI case and the CCE-compiled native version, which increased significantly up to 3X at larger messages; these routines include: Gather, Reduce_scatter and Scatter (Chart 12,13). Review of a number of IMB results showed that this behavior occurred repeatedly. These differences in performance appear to be due to compiler differences.

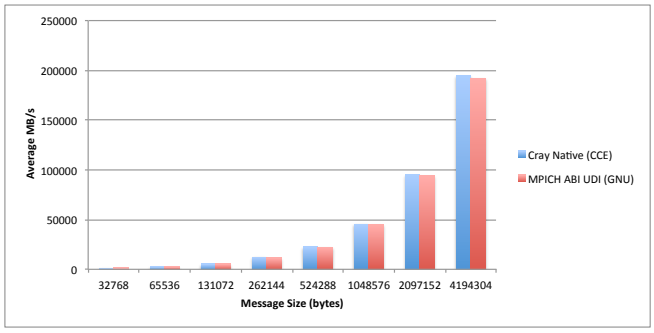

Figure K. MPI Alltoall Average Performance at 128 Nodes, 1 Rank/Node

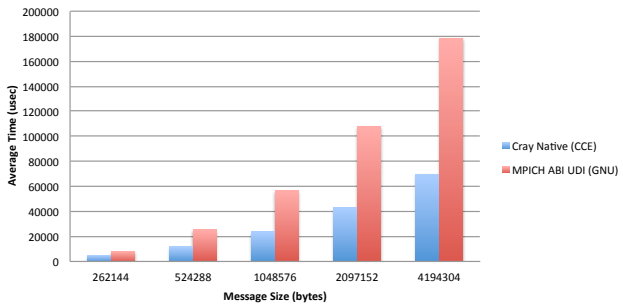

Figure L. MPI Gather Average Time at 128 Nodes, 1 Rank per Node

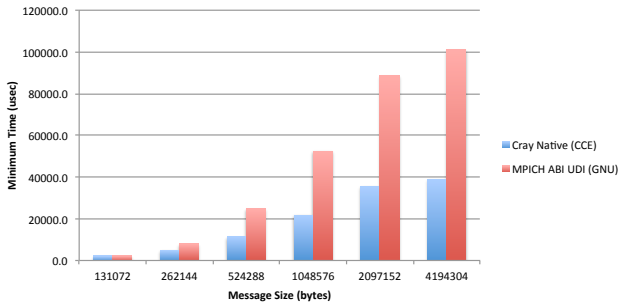

Figure M. MPI Gather Minimum Time at 128 Nodes, 1 Rank per Node

VI. Future Work

Application selection was primarily made on the basis on library dependencies. Several of the application test cases did not scale nearly as well as anticipated, which limited scaling to the available system sizes. For future work we plan to study applications with library dependencies that are less standard for the Cray XC environment as well as select test cases capable of scaling to larger core counts. In the cases where the UDI techniques outperformed native application versions, it is currently challenging to assess which factors contribute to the differences. While Cray performance tools could be used to profile the native application builds, corresponding performance tools are not yet available for the Shifter environment. Further investigation into performance tools for the UDI environment is also of interest.

VII. Summary

Shifter is still very much in its infancy as a product, but already can provide a means to run applications in new ways in the Cray environment. The applications profiled suggest that *MPICH ABI* and *Cray MPI* based UDIs can perform comparably to applications compiled natively with Cray MPI at least at for the scale of applications tested. Given the relatively minor performance differences between the two techniques, the *MPICH ABI UDI* technique is more applicable for the general user community because it avoids intellectual property concerns, is relatively straightforward to configure/deploy and allows the same image to target different networks without recompiling. The ability to deploy new MPI based software designed for different OS distributions while still retaining Cray HSN performance could drastically simplify the task of porting some applications to the Cray software environment.

Acknowledgment

The author would like to acknowledge the following people for their assistance and insights during this work—Steve Behling, Pierre Carrier, Brad Chamberlain, Jason Godfrey and Peter Johnsen from Cray.

References

[1] "Docker," https://www.docker.com/.

[2] D. Jacobsen, S. Canon, "Contain This, Unleashing Docker for HPC," presented at the Cray User Group., Chicago, IL., 2015

[3] S. Oyanagi, "Cray Support of the MPICH ABI Compatibility Initiative" pg 3-5, Cray Document number S-2544-704, February 2015, http://docs.cray.com/books/S-2544-704/S-2544-704.pdf -

[4] D. Jacobsen, J. Botts, S. Canon, "Never Port Your Code Again – Docker functionality with Shifter using SLURM", pg 16, presented at the SLURM User Group, Washington, DC, 2015 - http://slurm.schedmd.com/SLUG15/shifter.pdf

[5] MPICH ABI support - https://www.mpich.org/abi/