Launching Python Applications on Peta-scale Massively Parallel Systems

Yu Feng[‡§*], Nick Hand[‡]

https://youtu.be/CfrRDI71vTc

Abstract—We introduce a method to launch Python applications at near native speed on large high performance computing systems. The Python run-time and other dependencies are bundled and delivered to computing nodes via a broadcast operation. The interpreter is instructed to use the local version of the files on the computing node, removing the shared file system as a bottleneck during the application start-up. Our method can be added as a preamble to the traditional job script, improving the performance of user applications in a non-invasive way. Furthermore, we find it useful to implement a three-tier system for the supporting components of an application, reducing the overhead of runs during the development phase of an application. The method launches applications on Cray XC30 and Cray XT systems up to full machine capacity with an overhead of typically less than 2 minutes. We expect the method to be portable to similar applications in Julia or R. We also hope the three-tier system for the supporting components provides some insight for the container based solutions for launching applications in a development environment. We provide the full source code of an implementation of the method at *https://github.com/rainwoodman/python-mpi-bcast*. Now that large scale Python applications can launch extremely efficiently on state-of-the-art super-computing systems, it is time for the high performance computing community to seriously consider building complicated computational applications at large scale with Python.

Index Terms—Python, high performance computing, development environment, application

Introduction

The use of a scripting or interpreted programming language in high performance computing has the potential to go beyond post-processing and plotting results. Modern super-computers support dynamic linking and shared libraries, and thus, are capable of running the interpreters of a scripting programming language. Modern interpreters of scripting languages are equipped with the Just-In-Time (JIT) compilation technique that compiles the script in-time to achieve performances close to C or Fortran [LPS15], [BEKS14], [Aut06]. The Python programming language is of particular interest due to its large number of libraries and its wide user base. There are several Python bindings of the Message Passing Interface (MPI)[1] [DPKC11], [Mil02]. Bindings for higher level abstractions, e.g. [Spo12] also exist, allowing one to write complex parallel applications with MPI for simulations and data analysis.

However, it is still traditionally believed that Python does not coexist with large scale high performance computing. The major barrier is due to the slow and unpredictable amount of time required to launch Python applications on such systems. For example, it has been shown that the start-up time sometimes increases to hours for jobs with a relative small scale (a hundred ranks ..[#rank]). Some quantitative benchmarks can be see in [Fro13], [Lan12b].

The issue is an interplay between the current file system architecture on Peta-scale systems and the behavior of the Python interpreter. Peta-scale systems are typically equipped with a shared file system that is suitable for large band-width operations. The meta-data requests are handled by the so-called metadata servers, and usually, at most one master meta-data server serves all requests to a large branch of the file system; then, the data files are replicated to several data storage nodes [Sch03]. As an example, the Phase-I Cray XC 40 system Cori at NERSC is connected to 5 metadata servers (MDT) [NER15]. Because the file system is a shared resource with a limited throughput, it is relatively easy for an application to flood the file systems with requests and nearly bring an entire file system to a halt -- a phenomona most users to HPC systems are very familiar with.

Unfortunately, the Python interpreter is such an application, as has been repeatedly demonstrated in previous studies [Lan12a], [Lan12b], [Fro13], [ERSM11]. During start-up, a Python application will generate thousands of file system requests to locate and import files for scripts and compiled extension modules. We demonstrate the extent of the problematic behavior in Figure 1, where we measure the number of file system requests associated with several fairly commonly used Python packages on a typical system (Anaconda 2 and 3 in this case). The measurement is performed with `strace -ff -e file`. For either Python 2 or Python 3, the number of file system operations increases linearly with the number of entries in `sys.path` (controlled by the `PYTHONPATH` environment variable). Importing the scipy package with 10 additional paths requires 5,000+ operations on Python 2 and 2,000 operations on Python 3. Extrapolating to 1,000 instances or MPI ranks, the number of requests reaches 2 ~ 5 million. On a system that can handle 10,000 file system requests

* *Corresponding author: yfeng1@berkeley.edu*
‡ *Berkeley Center for Cosmological Physics, University of California, Berkeley CA 94720*
§ *Berkeley Institute for Data Science, University of California, Berkeley CA 94720*

1. Message Passing Interface (MPI) is the standard programming model on high performance computing. For readers that are unfamiliar with such topics, we recommend [Qui03] for an introduction to parallel programming and MPI.
2. A rank is defined as one of the concurrent processes of the application.

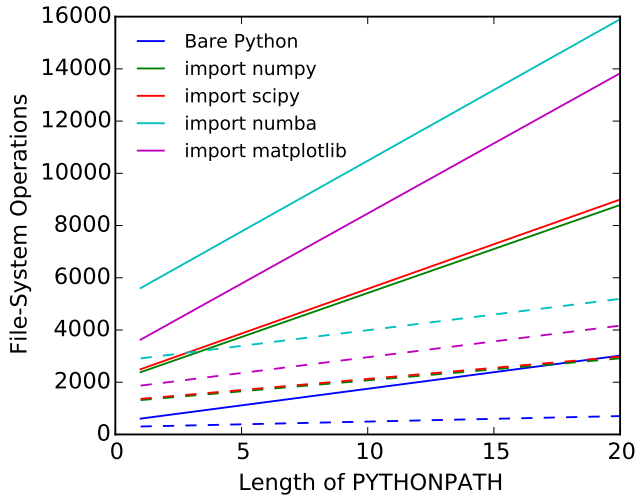

Fig. 1: *Number of file system requests during Python start-up. Solid lines: Python 2. Dashed lines: Python 3. We increase the number of entries in* `PYTHONPATH` *to simulate the number of packages installed in user directory or loaded via* `modules` *command commonly used on HPC systems.*

per second, consuming these requests takes 200 ~ 500 seconds of the full capacity of the entire system. Furthermore, the application becomes extremely sensitive to the load on the shared file system: when the file system is heavily loaded, the application will start extremely slowly.

It is worth pointing out that although the number of requests per rank can be significantly reduced, the total number of requests still increases linearly with the number of MPI ranks, and will become a burden at sufficiently large scale. For example, due to improvements in the importing system, the number of requests per rank is reduced by 50% in Python 3 as compared to Python 2 (seen in Figure 1). Therefore, a plain Python 3 application will handle twice as many ranks as Python 2 does.

In this paper, we present a solution (which we name `python-mpi-bcast`) that addresses the start-up speed without introducing a burden on the users. We have been using this method to launch data analysis applications in computational cosmology (e.g. [FH16]) at National Energy Research Scientific Computing Center (NERSC).

In Section 2, we collect and overview the previous solutions developed over the years. In Section 3, we describe our solution python-mpi-bcast. In Section 4, we discuss the management of the life-cycles of components. In Section 5, we demonstrate the cleanness of python-mpi-bcast with an example script. We conclude this paper and discuss possible extensions of this work in Section 6.

Previous Solutions

Given the importance and wide-adoption of the Python programming language, the application launch time issue has been investigated by several authors. We briefly review them in this section. These solutions either do not fully solve the problem or introduce a burden on the users to maintain the dependency packages.

The application delivery mechanism on a super-computer can deliver the full binary executable to the computing nodes. In fact, older systems can only deliver one statically linked executable file to the computing nodes during the job launch. The support of dynamic libraries on Cray systems was once very limited [ZDA+12] -- a significant amount of work has been invested to solve this limitation in the context of shared library objects (e.g. [AA14]).

One can take advantage of the standard delivery mechanism and launch the application at an optimal speed, by bundling the entire support system of the Python application as one statically compiled executable. [Fro13], [PM12] both fall into this category. We also note that the yt-project has adopted some similar approaches for their applications [TSO+11]. While being a plausible solution, the technical barrier of this approach is very high. Statically compiled Python is not widely used by the mainstream community, and special expertise is required to patch and incorporate every dependency package for individual applications. Although the steps are documented very well, the effort is beyond the knowledge of a typical Python developer.

Fortunately, in recent years the support for dynamic libraries on high performance computing systems has significantly improved, as super-computing vendors began to embrace a wider user base for general, data-intensive analysis. On these platforms, the main bottleneck has shifted from the lack of support for dynamic libaries to the vast number of meta-data requests to import the full python runtime library.

A particularly interesting approach is to eliminate the meta-data requests altogether via caching. Caching can happen at the user level or operation system level. On the user level, mpiimport [Lan12b] and Scalable Python cite:*scalablepython* attempt to cache the meta-data requests with an import hook. After the hooks are enabled, the user application are supposed to run as is. Unfortunately, these methods are not as fully opaque as they appear to be. With import hooks, because the meta-data requests are cached, they have to be calculated by the root rank first. Therefore, an implicit synchronization constraint is imposed in order to ensure the cache is evaluated before the requests from the non-root ranks. All of the import operations must be made either collectively or un-collectively at the same time. We find that the collective importing scheme breaks site.py in the Python standard library and the un-collective importing scheme breaks most MPI-enabled scripts. At the system level, users can file a ticket to mark a branch of the file system as immutable, allowing the computing nodes to cache the requests locally. This requires special requirements from the administrators, and in practice the relief has been limited.

Finally, one can locally mount a full application image on the computing node via a container-based solution [JCGB15]. The loopback mount adds a layer of caching to reduce the number of requests to the global file system. The drawback of the container-based solution is due to the requirement that the entire application is built as one image. Each time the application code is modified, the entire image needs to be re-generated before the job is ready to run. On super computing systems, it takes a long (and fluctuating) amount of time to build a non-trivial software package. Some of our support libraries (e.g. pfft-python) usually takes 10 to 20 minutes to rebuild from scratch. This waiting time can become an additional burden during code development. Furthermore, the user may need special privileges on the computing nodes in order to mount the images, requiring changes in the system security policy that can be challenging to implement for administration reasons; though we note that shifter has solved this problem at NERSC.

Our Solution: python-mpi-bcast

In this section, we show that the shared file system bottleneck can be solved with a much simpler approach that maintains a high level of compatibility with the main stream usage of the Python programming language.

Compatibility is maintained if one uses the vanilla C implementation of Python without any modifications to the import mechanism. A large number of file system requests during application start-up will be made, but we will reroute all shared file system requests to local file systems on the computing nodes, away from the limited shared file-system.

This is possible because the package search path of a Python interpreter is fully customizable via a few environment variables, a feature widely used in the community to provide support for 'environments' [LMR15], [Con15]. With python-mpi-bcast, we make use of this built-in relocation mechanism to fully bypass the scalability bottleneck of the shared file system. We note that none of the previous solutions make extensive use of this feature.

Because all file operations for importing packages are local after the re-routing, the start-up time of a Python application becomes identical to that of a single rank, regardless of the number of ranks used.

The only additional cost of our approach results from the delivery of the packages to the local file systems. In order to efficiently deliver the packages, we bundle the packages into tar files. The MPI broadcast function is used for the delivery. The tar files are uncompressed automatically with the tool `bcast.c` that could be linked into a static executable.

We will describe the steps in the following subsections:

1) Create bundles for dependencies and the application.
2) Deliver the bundles via broadcasting. The destination shall be a local file system on the computing nodes. (e.g. /dev/shm or /tmp)
3) Reroute Python search path (including shared library search path) to the delivery destination, bypassing the shared file system.
4) Start the Python application the usual way.

Creating bundles

We define a bundle as a compressed tar file that contains the full file system branch of a package or several packages, starting from the relative Python home directory. Three examples are:

1) The bundle file of a conda environment consists of all files in the bin, lib, include, and share directories of the environment. We provide a script (tar-anaconda.sh) for generating such a bundle from a conda environment. The size of a bundle for a typical conda environment is close to 300 MB.

2) The bundle file of a PIP installed package consists of all files installed by the `pip` command. We provide a wrapper command `bundle-pip` for generating a single bundle from a list of PIP packages.

3) The bundle file of basic system libraries includes those shared library files that are loaded by the dynamic linker for the Python interpreter. We provide three sample job scripts to generate these bundles for three Cray systems: XC30, XC40, and XT. The system bundle addresses the shared library bottleneck investigated in [ZDA+12] (DLFM) but without requiring an additional wrapper of the system dynamic linker.

The bundles only need to be updated when the dependencies of an application are updated.

Variable	Action
`PYTHONHOME`	Set to broadcast destination
`PYTHONPATH`	Purge
`PYTHONUSERBASE`	Purge
`LD_LIBRARY_PATH`	Prepended by `/lib` of the broadcast destination

TABLE 1: *Environment Variable used in* `python-mpi-bcast`

Delivery via broadcasting

Before launching the user application, the bundles built in the previous step must be delivered to the computing nodes -- we provide a tool for this task. On Cray systems, we make use of the memory file system mounted at `/dev/shm`. On a system with local scratch, `/tmp` may be used as well, although this has not been tested.

We use the broadcast function of MPI for the delivery. The tool first elects one rank per node to receive and deploy the bundles to a local storage space. The bundle is then uncompressed by the elected rank per computing node.

The new files are marked globally writable. Therefore, even if some of the files are not properly purged from a node, they can be overwritten by a different user when the same node is allocated to a new job. We note that this may pose a security risk in shared systems.

When several bundles are broadcast in the same job, the later ones will overwrite the former ones. This overwriting mechanism provides a way to deliver updates as additional bundles.

We also register an exit handler to the job script that purges the local files to free up the local file system. This step is necessary on systems where the local storage space is not purged after a job is completed.

Rerouting file system requests

We list the environment variables that are relevant to the relocation in Table 1. After the relocation, all of the file system requests (meta-data and data) are rerouted to the packages in the local file system. As a result, the start-up time of the interpreter drops to that of a single rank.

We note that the variable PYTHONUSERBASE is less well-known, documented only in the `site` package, but not in the Python command-line help or man pages. If the variable is not set, Python will search for packages from the user's home directory `$HOME/.local/`. Unfortunately, the home file-system is typically the slowest one in a Peta-scale system. This directory is not part of the application, therefore we purge this variable by setting it to an invalid location on the local file system, the root of the broadcast destination. We also purge `PYTHONPATH` in the same manner, since all packages are located at the same place. The variable `PYTHONPATH` can be very long on systems where each Python package is provided as an individual module of the `modules` system. This negtively impacts the performance of launching Python applications, as we see in Figure 1, which clearly shows that the length of `PYTHONPATH` has a huge impact on the number of file system operations that occur during start-up.

Launching the Python application

We launch the Python application via the standard python-mpi wrapper provided by `mpi4py`. We emphasize that no modifica-

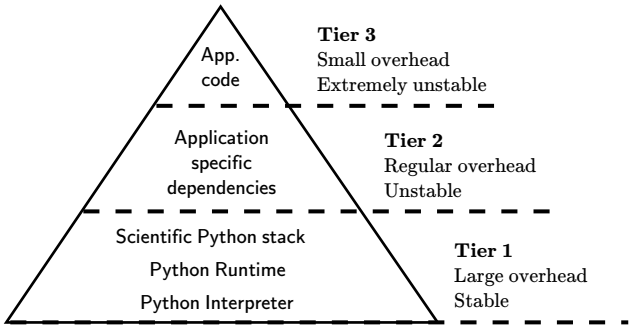

Fig. 2: *Three tiers of bundles. The most stable component (bottom of the pyramid, Tier 1) takes the most effort to build. The least stable component (top of the pyramid, Tier 3), takes the least effort to bundle. The split into three tiers allows the developers to save time in maintaining the bundles.*

tions to the python-mpi wrapper or to the user application are needed in our approach.

It is important to be aware that Python prepends the parent directory of the start-up script to the search path. If the start-up script of the application resides on a shared file system, the access to this directory will slow down the application launch. As an alternative, the application script (along with the full directory tree) can also be bundled and delivered via python-mpi-bcast before the launch. This is demonstrated in the example in Section 5, and we will discuss this case in more detail in the next section.

On a Cray system, the Python interpreter (usually `python-mpi`) must reside in a location that is accessible by the job manager node, because it will be delivered via the standard application launch process.

Three-tiers of bundles

Building bundles takes time and shifts the focus of the developer from application development to interfacing with the system. We therefore recommend to organize the components of an application into a three-tier system to minimize the redundant efforts required to to create bundles. The three-tier system is illustrated in Figure 2, and we describe the rationale and definitions in the following sections.

Tier 1 components

Tier 1 components consist of the Python interpreter, standard runtime libraries, and stable dependencies (dependencies that changes infrequently, for example, numpy, scipy, mpi4py, h5py). On a conda based Python distribution, the Tier 1 components map to the packages included in a conda environment. These components provide a basic Python computing environment, take the most time to install, yet barely change during the life-cycle of a project. Most super-computing facilities already maintain some form of these packages with the `modules` system, e.g. NCSA has a comprehensive set of Python packages [Ms14], and NERSC has the anaconda 2 and 3 based Python distribution.

It is straightforward to create bundles of these pre-installed components. We provide the `bundle` command with python-mpi-bcast for creating a bundle from a pre-installed 'modules' path. It is a good practice to create one bundle for each 'modules' path. The process can be time consuming, even though it does not invole compiling any source code packages. For example, creating a Tier 1 bundle from a full binary anaconda installation typically takes 5 minutes at NERSC facilities.

Tier 2 components

Tier 2 components consist of unstable dependencies of the application. These include packages used or developed specifically for the particular application, which are usually neither part of the conda distribution nor deployed at the computing system by the facility. Tier 2 components update frequently during the life-cycle of a project.

The difference in update-frequency means that Tier 2 components should not be bundled with the Tier 1 components. Since Tier 2 components are usually much smaller and thus faster to bundle than Tier 1 components, bundling them separately reduces the overhead for running and testing the application live at the supercomputing facility.

We provide a pip wrapper script `bundle-pip` with python-mpi-bcast to build bundles for the Tier 2 components. A good practice is to create a single bundle for all of the Tier 2 components with one invocation to the tar-pip.sh wrapper.

Tier 3 components

Tier 3 components are the application itself and other non-package dependencies. These include the main script and files in the same directory as the main script. The Tier 3 components change most frequently among the three tiers during the life cycle of a project. As Tier 3 components mature and receive less frequent changes they should be migrated into Tier 2, following the usual software refactoring practices.

We implement two strategies for Tier 3 components. The *simple* strategy is to leave these files at the original location in the shared file system. In this case, Python will prepend the parent directory of the main script to the search path, not fully bypassing the shared file system. We find that the extra cost due to this additional search is usually small. However, when the system becomes highly congested (an ironic example is when another user attempts to start a large Python job without using our solution), the start-up time can observe a significant slow down.

A consistently reliable start-up time is obtained if Tier 3 components are also bundled and delivered to the local file system (*mirror* strategy). The location of the main script in the job script should be modified to reflect this change. Because the Tier 3 components are the most lightweight, typically consisting of only a few files, a good practice is to create the bundle automatically in the job script, without requiring the developer to manually create a bundle before every job submission. We provide a helper command *mirror* that implementes the strategy. The *mirror* strategy is demonstrated in the next section with examples.

Example Scripts

Generic Cray Systems

In this section, we show an example SLURM job script on a Cray XC 30 system. The script demonstrates the non-invasive nature of our method. After the bundles are built, a few extra lines are added to the job script to enable python-mpi-bcast and deliver the three tiers of components. The user application does not need to be specifically modified for python-mpi-bcast. We emphasize that the job script runs in the user's security context, without any special requirements from the facility.

```
# Script without NERSC integration
# Modify and adapt to use on a general
# HPC system

#! /bin/bash
#SBATCH -n 2048
#SBATCH -p debug

export PBCAST=/usr/common/contrib/bccp/python-mpi-bcast

source $PBCAST/activate.sh \
    /dev/shm/local "srun -n 1024"

# Tier 1 : anaconda
bcast -v $PBCAST/2.7-anaconda.tar.gz \
         $HOME/fitsio-0.9.8.tar.gz

# Tier 2 : commonly used packages
# e.g. installed in $PYTHONUSERBASE
bcast-userbase

# Tier 3 : User application
mirror /home/mytestapp/ \
    testapp bin

# Launch
time srun -n 1024 python-mpi
    /dev/shm/local/bin/main.py
```

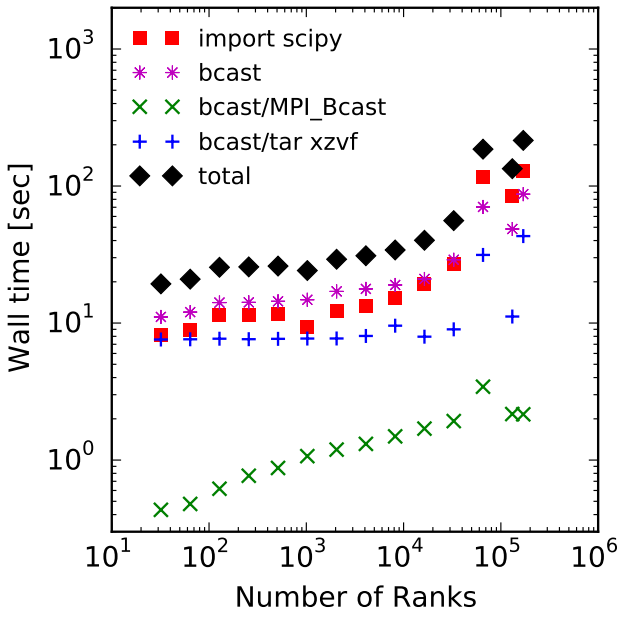

Fig. 3: *Time measurements of python-mpi-bcast on Edison, a Cray XC 30 system at NERSC. We perform tests launching a dummy Python 2 application (that imports scipy) with up to 127,440 MPI ranks. The total time in the bcast job step is shown in stars. The two major time consuming components of bcast, the call to* `MPI_Bcast` *(×) and the call to the* `tar` *command, are also shown (+). Note that large jobs incur a large overhead in the job step such that the sum of the latter differs from the job step times. The total time of the job step that launches the dummy application is shown in squares. The total time of both job steps is shown in diamonds.*

Integration with NERSC Facilities

On the NERSC systems where `python-mpi-bcast` was originally developed, we also provide a default installation of `python-mpi-bcast` that is integrated with the `modules` system and the Anaconda based Python installations. The full integration source code is hosted together in the main python-mpi-bcast repository and can be easily adapted to other systems.

The following script provides an example for using `python-mpi-bcast` in a pre-configured system. Note that the Python runtime environment (along with shared libraries from the Cray Linux Environment) are automatically delivered. The impact on the user application is limited to two lines in the job script: one line for enabling python-mpi-bcast and the other line to mirror the application to a local file system with the `mirror` command.

```
#! /bin/bash
#SBATCH -N 2048
#SBATCH -p debug

# select the Python environment
module load python/3.4-anaconda

# NERSC integration
PBCAST=/usr/common/contrib/bccp/python-mpi-bcast
source $PBCAST/nersc/activate.sh

# Directly deliver the user application
mirror /home/mytestapp/ \
    testapp bin

# launch the mirrored application
time srun -n 1024 python-mpi \
    /dev/shm/local/bin/main.py
```

Benchmark and Performance

In Figure 3 and 4, we show the measurement of wall clock time of python-mpi-bcast for a dummy Python 2 application on the Cray XC30 system Edison at NERSC and the Cray XT system BlueWaters at NCSA. The dummy application imports the scipy package on all ranks before exiting. We point out that in the benchmark it is important to import Python packages as done in a real application, because most of the metadata requests are to locate the Python scripts of packages rather than dynamic libraries associated with extension modules. Therefore, a benchmark based on performance of `simulating` dynamic libraries [LAdS+14] does not properly represent the true launch time of a realistic Python application. We do not perform another set of benchmarks for Python 3, but note that the stream-lined import system in Python 3 could perform better than Python 2. [van02]

The job includes two steps: the first involves the statically linked bcast program that delivers the bundles to the computing nodes (which does not involve Python), and the second launches the Python application.

The bcast step consists of two major components, a call to `MPI_Bcast` and a call to `libarchive`[Tim09] to inflate the tar ball. We observe that the scaling in the `MPI_Bcast` function is consistent with the expected $O[\log N]$ scaling of a broadcast algorithm. The call to inflate the tar ball remains roughly constant, but shows fluctuations for larger runs on the XC30 system. This is likely because the job has hit a few nodes that are in a non-optimal state, which is a common effect in jobs running near the capacity of the system.

As a further evidence, the fluctuation in the large jobs correlates with an increase in the time spent in the 'tar' stage of the bcast time step, as seen by comparing the tests with 49,152 ranks (2048 nodes), 98,304 ranks(4096 nodes), and 127,440 ranks (5310 nodes).

The time spent in the Python application (second job step) increases slowly as well, but the increase becomes more significant

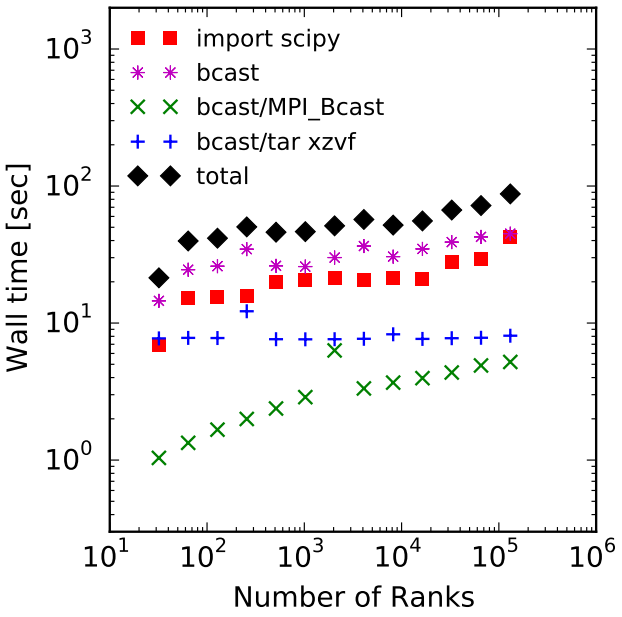

Fig. 4: *Time measurements of python-mpi-bcast on BlueWaters. a Cray XT system at NCSA. We perform tests launching a dummy Python 2 application (that imports scipy) with up to 127,440 MPI ranks. The total time in the bcast job step is shown in stars. The two major time consuming components of bcast, the call to* `MPI_Bcast` *(×) and the call to the 'tar' command, are also shown (+). Note that large jobs incur a large overhead in the job step such that the sum of the latter differs from the job step times. The total time of the job step that launches the dummy application is shown in squares. The total time of both job steps is shown in diamonds.*

as the size of the job approaches the capacity of the system. An additional cause of the increase can be attributed to the remaining few requests to the shared file system for unbundled shared libraries and Python configuration files that are not rerouted. For example, the configuration of mpi4py package is hard coded on the shared file system.

For jobs with less than 1024 nodes, the timing is close to 1 minute. In any case, the largest test on Edison that employs 127,440 MPI ranks (5310 nodes), spent 4 minutes in total for launching the application. We note that the slightly smaller job that employs 98,304 ranks (4096 nodes) spent less than 2 minutes in total.

Conclusions

We introduce `python-mpi-bcast`, a solution to start native Python applications on large, high-performance computing systems.

We summarize and review a set of previous solutions developed over the years and with varying usage in the community. Their limitations in terms of practical usability and efficiency are discussed.

Our solution `python-mpi-bcast` does not suffer from any of the drawbacks of previous solutions. Using our tool, the runtime environment of the Python application on Peta-scale systems is fully compatible with the the mainstream Python environment. The entire solution can be added as a preamble to a user job script to enhance the speed and reliability of launching Python applications on any scales, from a single rank to thousands of ranks.

Our solution makes use of the established infrastructure of the mainstream Python community to reroute support packages of an application from the shared file system to local file systems per node via bundles. The solution is compatible with Python 2 and 3 at the same time. Almost all accesses to the shared file system are eliminated, which avoids the main bottleneck typically encountered during the start-up stage of a Python application. We have performed tests up to 127,440 ranks on a Cray XC 30 system (limited by the available cores on the Edison system at NERSC) and on a Cray XT system BlueWaters at NCSA. There is no fundamental reason that the method does not scale to even larger jobs, given that the only non-local operation is a broadcast operation.

We introduce a three-tier bundling system that reflects the evolutionary nature of an application. Different components of an application are bundled separately, reducing the preparation overhead for launching an application during the development stage. The three-tier system is an improvement from the all-in-one approaches such as [Fro13] or [JCGB15]. We in fact advocate adopting a similar system in general-purpose, images-based application deployment infrastructure (e.g. in cloud computing). We note that a large burden from the users can be further removed if the computing facilities maintain the Tier 1 bundle(s) in parallel with their existing `modules` system. Further integration into the job system is also possible to provide a fully opaque user experience.

Finally, with few modifications, `python-mpi-bcast` can be easily generalized to support applications written in other interpreted languages such as Julia and R. In addition, we highly welcome reimplementing the stratagies documented in the paper as an extension of the Conda package distribution system, and provide the full source code of `python-mpi-bcast` at https://github.com/rainwoodman/python-mpi-bcast.

Given that large-scale Python applications can be launched extremely efficiently on state-of-the-art super-computing systems, it is the time for the high-performance computing community to begin serious development of complex computational applications at large scale with Python.

Acknowledgment

The original purpose of this work was to improve the data analysis flow of cosmological simulations. The work is developed on the Edison system and Cori Phase I system at National Energy Research Super-computing Center (NERSC), under allocations for the Baryon Oscillation Spectroscopic Survey (BOSS) program and the Berkeley Institute for Data Science (BIDS) program. We also performed benchmark on the Blue Waters system at National Center for Super-computing Applications (NCSA) as part of the NSF Peta-apps program (NSF OCI-0749212) for the BlueTides simulation. The authors thank Zhao Zhang of Berkeley Institute of Data Science, Fernando Perez of Berkeley Institute of Data Science, Martin White of Berkeley Center for Cosmology, Rollin Thomas of Lawrence Berkeley National Lab, Aron Ahmadia of Continuum Analysis Inc., for insightful discussions over the topic.

References

[AA14] William Scullin Aron Ahmadia, Jed Brown. Joint anl/ksl collaboration on collective file system module (with glibc dynamic

library interception), 2014. URL: https://github.com/ahmadia/collfs.

[Aut06] V8 Project Authors. V8 javascript engine, 2006. URL: https://chromium.googlesource.com/v8/v8.git.

[BEKS14] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A Fresh Approach to Numerical Computing. *ArXiv e-prints*, November 2014. arXiv:1411.1607.

[Con15] Continuum Analytics, Inc. Conda, 2015. URL: http://conda.pydata.org/docs/.

[DPKC11] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124 – 1139, 2011. New Computational Methods and Software Tools. URL: http://www.sciencedirect.com/science/article/pii/S0309170811000777, doi:http://dx.doi.org/10.1016/j.advwatres.2011.04.013.

[ERSM11] Jussi Enkovaara, Nichols A. Romero, Sameer Shende, and Jens J. Mortensen. Gpaw - massively parallel electronic structure calculations with python-based software. *Procedia Computer Science*, 4:17 – 25, 2011. Proceedings of the International Conference on Computational Science, ICCS 2011. URL: http://www.sciencedirect.com/science/article/pii/S1877050911000615, doi:http://dx.doi.org/10.1016/j.procs.2011.04.003.

[FH16] Yu Feng and Nick Hand, 2016. URL: https://github.com/bccp/nbodykit.

[Fro13] Bradley M. Froehle, 2013. URL: https://github.com/bfroehle/slither.

[JCGB15] Doug Jacobsen, Shane Canon, Lisa Gerhardt, and Deborah Bard. Shifter: Bringing linux containers to hpc, 2015. URL: https://www.nersc.gov/research-and-development/user-defined-image.

[LAdS+14] Gregory L. Lee, Dong H. Ahn, Bronis R. de Supinski, John Gyllenhaal, and Patrick Miller. The python dynamic benchmark, 2014. URL: https://codesign.llnl.gov/pynamic.php.

[Lan12a] Asher Langton. Email communication to the mpi4py forum., 2012. URL: https://groups.google.com/forum/#!topic/mpi4py/h_GDdAUcviw.

[Lan12b] Asher Langton. An mpi-aware import module for python, 2012. URL: https://github.com/langton/MPI_Import.

[LMR15] Jannis Leidel, Carl Meyer, and Brian Rosner. Virtual python environment builder, 2015. URL: https://pypi.python.org/pypi/virtualenv.

[LPS15] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A llvm-based python jit compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, pages 7:1–7:6, New York, NY, USA, 2015. ACM. URL: http://doi.acm.org/10.1145/2833157.2833162, doi:10.1145/2833157.2833162.

[Mil02] Patrick Miller. pympi – an introduction to parallel python using mpi. 2002.

[Ms14] Colin MacLean and NCSA staff. bwpy: Python on bluewaters, 2014. URL: https://bluewaters.ncsa.illinois.edu/python.

[NER15] NERSC. Cori system specifications, 2015. URL: https://www.nersc.gov/users/computational-systems/cori/cori-phase-i/.

[PM12] Bradley M. Froehle Pat Marion, Aron Ahmadia. Import without a filesystem: scientific python built-in with static linking and frozen modules, 2012. URL: http://conference.scipy.org/scipy2013/presentation_detail.php?id=129.

[Qui03] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[Sch03] Philip Schwan. Lustre: Building a File System for 1,000-node Clusters. In *Proceedings of the 2003 Linux Symposium*, July 2003.

[Spo12] William F. Spotz. Pytrilinos: Recent advances in the python interface to trilinos. *Sci. Program.*, 20(3):311–325, July 2012. URL: http://dx.doi.org/10.1155/2012/965812, doi:10.1155/2012/965812.

[Tim09] Tim Kientzle and contributors. libarchive: Multi-format archive and compression library. http://libarchive.org, 2009.

[TSO+11] M. J. Turk, B. D. Smith, J. S. Oishi, S. Skory, S. W. Skillman, T. Abel, and M. L. Norman. yt: A Multi-code Analysis Toolkit for Astrophysical Simulation Data. *Astrophysical Journal Supplement*, 192:9–+, January 2011. arXiv:1011.3514, doi:10.1088/0067-0049/192/1/9.

[van02] van Rossum, Just and Moore, Paul . Pep 302 – new import hooks. 2002. URL: https://www.python.org/dev/peps/pep-0302/.

[ZDA+12] Zhengji Zhao, Mike Davis, Katie Antypas, Yushu Yao, Rei Lee, and Tina Butler. Shared library performance on hopper. *Cray User Group*, 2012. URL: https://cug.org/proceedings/attendee_program_cug2012/includes/files/pap124.pdf.