Pynamic

See https://asc.llnl.gov/CORAL-benchmarks/
for examples of benchmark summary files and platform-specific problem sets

Summary Version

1.0

Purpose of Benchmark

Pynamic was developed to influence the design of the dynamic loading subsystem on large machines and to test its ability to handle the intensive use of dynamically-linked libraries, as often exhibited by large Python-based scientific applications. Running sequentially (i.e., a single MPI task), this benchmark measures how quickly the dynamic loader can load the configured number of dynamic libraries into a process, resolve all the symbols in these libraries, and execute all the routines defined in them. Running in parallel, it also captures how scalably the relevant subsystem can handle large numbers of file system operations that the dynamic loader requires. The scalable use of file systems during dynamic loading is critically important for large machines: when many parallel processes in a Python-based application simultaneously load dynamic Python modules, a file-system-access storm can be created, leading to unacceptably slow performance and further having an effect similar to a site-wide denial-of-service attack on the file systems where these modules reside.

Characteristics of Benchmark

Pynamic is a benchmark that generates a user-specified number of Python dynamic modules and utility libraries objects to emulate a wide range of dynamic linking and loading behavior exhibited by Python-based scientific programs. The benchmark can be run in parallel using either pyMPI (http://pympi.sourceforge.net) or mpi4py (http://mpi4py.scipy.org/) and can be tested in several modes. For example, in one mode, Pynamic directly links in all these modules at link time, creating the *pynamic-pyMPI* executable. In another mode, Pynamic dynamically loads these modules at runtime via Python's import construct into a vanilla *pyMPI* or a base *python* interpreter with the mpi4py module installed. Finally, Pynamic also generates a ~200MB *pynamic-bigexe* executable, which tests a system's ability to handle large, dynamically-linked executables.

While the benchmark supports configurable emulation of the DLL usage, we recommend large configurations in terms of the number and the size of Python dynamic modules and of utility libraries to be used for testing. Making full use of some of Python's popular features have led to applications that access extremely high numbers of DLLs. For example, one of LLNL's important multi-physics applications uses over nine hundred dynamic libraries. With the appropriate parameters, Pynamic can build dummy applications that closely model the footprint of important Python-based multi-physics codes.

Pynamic provides three performance metrics to capture each of three phases pertaining to the DLL usage: the startup time for library loading; the module-import time for symbol

resolution; and the visit time for execution. However, because contemporary dynamic loaders provide parameters that allow shifting of such overheads from one phase to another, the summation of all three metrics should form the overall dynamic loader performance metric.

Mechanics of Building Benchmark

Pynamic includes the source for pyMPI, which requires a Python installation for the compute nodes. Furthermore, to run the mpi4py driver, the Python installation must also include an mpi4py build. In addition, two of the key Pynamic files are themselves Python scripts, which requires a Python installation on the build node. The required configuration parameters are as follows:

```
./config_pynamic.py 900 1250 -e -u 350 1250 -n 150
```

This will create the specified set of DLLs and the pynamic-pyMPI and pynamic-bigexe executables with all of the DLLs linked in. It will also create a stand-alone pyMPI executable (i.e., without the DLLs linked in). Pynamic must be built as a dynamically-linked executable. Arguments can be passed into pyMPI's configure script by appending the -c option (run `` `./config_pynamic.py –h` `` for more details).

Mechanics of Running Benchmark

Pynamic shall be run in several modes:

```
srun ./pynamic-pyMPI pynamic_driver.py `date +%s`

srun ./pynamic-bigexe pynamic_driver.py `date +%s`

srun ./pyMPI pynamic_driver.py `date +%s`

# the following requires mpi4py to be installed in
# your python interpreter
srun python pynamic_driver_mpi4py.py `date +%s`
```

Note: The result of the `date +%s` command must be passed as a command-line argument in order to get the startup time of Pynamic.

Verification of Results

A successful serial run of Pynamic (i.e., no errors) is sufficient verification of functionality. The resulting *startup* + *module-import* + *visit* metric provides insight into the efficiency of the dynamic loading subsystem. A time comparison between *pynamic-pyMPI* and *pyMPI* provides insight into the benefits and the penalties of linking against the generated shared libraries. Additionally, measuring the time of a cold start (first invocation) captures the cost of initially loading Pynamic from the file system and the time of a warm start (subsequent invocations) provides insight into the time savings from running from a warm disk buffer cache. Finally, the *start-up* + *module-import* + *visit* metric resulting from a series of scaling runs of Pynamic illustrates the scalability of the

dynamic loading subsystem in handling large number of file system operations that the dynamic loader require at scale.