Preprint
UCRL-JC-148522

Parallel, Distributed Scripting with Python

P. Miller

This article was submitted to
3rd Linux Clusters Institute International Conference on Linux Clusters: The HPC Revolution, St. Petersburg, Florida, October 23-25, 2002

May 24, 2002

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

Parallel, Distributed Scripting with Python*

Patrick Miller
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
patmiller@llnl.gov

May 24, 2002

Title: Parallel, Distributed Scripting with Python
Track: Applications & Tools

Presenter: Patrick Miller
E-mail: patmiller@llnl.gov
Phone: 925-423-0309

Abstract

Parallel computers used to be, for the most part, one-of-a-kind systems which were extremely difficult to program portably. With SMP architectures, the advent of the POSIX thread API and OpenMP gave developers ways to portably exploit on-the-box shared memory parallelism. Since these architectures didn't scale cost-effectively, distributed memory clusters were developed. The associated MPI message passing libraries gave these systems a portable paradigm too. Having programmers effectively use this paradigm is a somewhat different question. Distributed data has to be explicitly transported via the messaging system in order for it to be useful.

In high level languages, the MPI library gives access to data distribution routines in C, C++, and FORTRAN. But we need more than that. Many reasonable and common tasks are best done in (or as extensions to) scripting languages. Consider sysadm tools such as password crackers, file purgers, etc... These are simple to write in a scripting language such as Python (an open source, portable, and freely available interpreter). But these tasks beg to be done in parallel. Consider the a password checker that checks an encrypted password against a 25,000 word dictionary. This can take around 10 seconds in Python (6 seconds in C). It is trivial to parallelize **if** you can distribute the information and co-ordinate the work.

This paper will present pyMPI, a distributed implementation of Python extended with an MPI interface. The tool makes it easy to write parallel Python scripts for system administration, data exploration, file post-processing, and even for writing full blown scientific simulations. Parallel Python also allows developers to prototype the data distribution for parallel algorithms in a easy, interactive, and intuitive manner without having to compile code, build specialized MPI types, and build serialization mechanisms. pyMPI supports most of the MPI API. It allows access to sends, receives, barriers, asynchronous messaging, communicators, requests, and status. In short, it provides a fully functional parallel environment coupled with a powerful scripting engine. The combination simplifies the generation of large scale, distributed tools for clusters.

Scripting languages are often used as glue – to mix and match previously unrelated tools and modules to solve a problem. Parallel scripting languages can combine serial components to build parallel applications. In some cases, the parallelization is trivial and one simply needs a simple framework in which to execute – pyMPI provides that. For example, one could write a parallel Python script to undertake a parameter study of a serial application or component. It is a distributed loop, but it is easy to write in parallel Python:

```
import mpi
...
for x in mpi.scatter(parameters):
   os.system('sample_application -parameter=\%s'%x)
...
```

It is important that the parallel scripting framework have a complete and robust implementation of MPI. This is particularly true when tasks need to be co-ordinated. Consider pre-processing graphics files and combining into a master file. A simple shell script can only co-ordinate through the file system, but pyMPI can simply throw a barrier:

```
import mpi
...
for file in mpi.scatter(files):
   os.system('postprocess %s'%file)
mpi.barrier()
if mpi.rank == 0:
   os.system('buildmaster %s'%string.join(files))
...
```

One can also co-ordinate a parallel application written as a Python extension. In one application, we have written a sophisticated, parallel, multi-level time-step control in pure Python (enhanced with MPI). Similarly, one can use the capabilities of a parallel enhanced Python to bring distributed data back together so that serial tools can work on the combined data:

```
import mpi
...
# All processors do this...
rho = simulation.computeDensity()

globalPosition = mpi.gather(x)
globalRho = mpi.gather(rho)
if mpi.rank == 0:
  for position,density in map(None,globalPosition,globalRho):
    print position,density
...
```

Extra nodes can be utilized to perform clean-up, post-processing, and archival tasks that need to be tightly co-ordinated with a parallel task. In the example below, dump files are moved off to the archive after every step. Since the move is prompted by a MPI message from the application, the files will never be picked up in an intermediate state (e.g. the file exists, but has not been finalized).

```
import mpi
# Build a sub communicator for the ''real'' work
comm = mpi.dup(world[:-1])
if comm:
   worker = something(comm) # Use the sub-communicator
   for i in range(steps):
worker.do_work()
comm.barrier()   # Wait just on workers
mpi.send("archive",mpi.size-1,tag=0) # on world comm
   mpi.send("done",mpi.size-1,tag=1) # on world comm
else:
   # A single cleanup process
   while 1:
      msg, status = mpi.recv()
      if msg == "archive":
          <copy files to archive>
      elif msg == "done":
          break
```

In a similar manner, one can construct simulations of large grain components. Consider a physically divided computation such as a turbo-machinery simulation. The application follows air flows through compressor turbines through a combustor and then through exhaust vanes. Consider the major components of the application: a turbine simulator, a combustor simulator, and an interface control. Parallel Python can be used to set up the communicators for each, handle details of data manipulation, control outputs, handling load balancing, etc...

pyMPI couples the distributed parallelism of MPI with the scripting power of Python. MPI unleashes the power of large clusters, while Python brings access to hundreds of easy to use, freely available modules. Together, they can simplify both sysadm and scientific simulation tasks on high-end clustered machines.