# MYMPI – MPI programming in Python

Timothy H. Kaiser, Ph.D. - San Diego Supercomputer Center, tkaiser@sdsc.edu
Leesa Brieger - San Diego Supercomputer Center
Sarah Healy - UCSD Department of Bioengineering

## ABSTRACT

We introduce an MPI Python module, MYMPI, for parallel programming in Python using the Message Passing Interface (MPI). This is a true Python module which runs with a standard Python interpreter.

In this paper we discuss the motivation for creating the MYMPI module, along with differences between MYMPI and pyMPI, another MPI Python interpreter. Additionally, we discuss three projects that have used the MYMPI module: Continuity, in computational biology, and Montage, for astronomical mosaicking. We will show an example, workit, that shows that the module can easily be used to create simple parallel pipelines.

## INTRODUCTION

Message Passing Interface (MPI) is the standard for the common library used in parallel high performance computing (HPC). There are several freely available implementations of MPI written for both Fortran and C. There are also some commercial versions of MPI available. Most HPC platforms have one or more versions of MPI installed.

Parallel applications written using MPI contain an initialization routine, MPI_INIT, a number of communication calls, such as MPI_SEND, and a finalization routine MPI_FINALIZE. Two other common calls are MPI_COMM_SIZE and MPI_COMM_RANK. These calls return the number of processors participating in a parallel application, say NPROC, and a number unique to each process, 0 to NPROC-1.

Python, is an interpreted, interactive, object-oriented, extensible programming language. The Python language is extended by writing modules. Modules can be written in Python or they can be written using a compiled language such as C or Fortran. C or Fortran modules are compiled as libraries or shared objects.

MYMPI is an implementation of a subset of the MPI standard written with bindings in Python. It can be used to write parallel applications using Python, or a mixture of Python, Fortran and C.

Python modules can be loaded at program execution time using the “import” statement. The MYMPI module is simply called “mpi.” We can import all of the functionality of the MYMPI module and have it behave as if it were originally part of Python. This can be done using the statement:

```
from mpi import *
```

A Python MPI version of “Hello world” could be written as:

```
#!/usr/bin/env python
from mpi import *
```

```
import sys

# initialize mpi
# the initialization routine requires the
# command line arguments held in sys.argv
sys.argv =  mpi_init(len(sys.argv),sys.argv)

# get the total number of processes in this parallel
# job and “myid”, the identifier for each process
myid=mpi_comm_rank(MPI_COMM_WORLD)
numprocs=mpi_comm_size(MPI_COMM_WORLD)

# each process will print its identifier and the total number
# of processes
print "Hello from ",myid, "The total # of processors is ",numprocs

mpi_finalize()
```

If this job is run on 2 processors the output might be:

```
Hello form 0 The total # of processors is 2
Hello form 1 The total # of processors is 2
```

Real programs would contain additional computation and communication routines. Often the various processes perform different tasks based on “myid”. For example, process 0 could read the input data then send it to the other processes using a broadcast command. Each process might do some computation and pass results back to process 0 for saving.

MYMPI is not the only Python MPI implementation. pyMPI is another Python MPI system.

The main difference between pyMPI and MYMPI is that pyMPI is actually a custom version of the Python interpreter along with a module. Our version, MYMPI, is a module only that can be used with a normal Python interpreter.

Having a module that can be used with a normal interpreter has some advantages. With MYMPI you can use the same version of Python for both parallel and serial applications. One reason this is advantageous is that as you add extensions you don’t need to add them to multiple versions of Python. The same modules will be available for both serial and parallel applications. MYMPI is easier to build and modify because you don’t need to rebuild the interpreter when making changes. Also, it is possible to have a production version of the MYMPI MPI module and a test version. The version that is loaded at runtime can be set by an environmental variable. MYMPI is contained in a single 33 Kbyte file and compiles on modern machines in about 3 seconds. pyMPI is about 1.7 Mbytes and compiles in minutes.

There is also a fundamental difference in the semantics of the two packages. With MYMPI the code executed by the interpreter is the parallel application. With pyMPI the Python interpreter is the parallel application. That is, when the pyMPI interpreter starts it will start as a MPI application. The particular implications of pyMPI starting as a MPI application are machine

dependent. However, on all machines the semantic difference requires a different programming style.

As discussed above, the routine MPI_Init is used to initialize a MPI program. That is, every process in the parallel job must call MPI_Init. pyMPI departs from this standard operating procedure. pyMPI programs do not call MPI_Init because the interpreter is the MPI application and it calls MPI_Init on startup. In MYMPI, programs explicitly call MPI_Init. One of the reasons why MYMPI was created was to provide better control of how and when MPI_Init is called. That is, our driving application required that MPI_Init be called at a certain point in the application. Also, it is possible that the decision to run the application in parallel or serial mode can be made after the application is started. (On some machines this is not possible because parallel applications need to be started using special queuing commands.)

MYMPI follows the syntax and semantics of C and Fortran MPI as much as practical. pyMPI has a more object-oriented flavor. We chose to match C and Fortran more closely because we intended from the start that we would be writing programs that mixed Python with the other languages. Also, the training materials we have developed for teaching standard MPI can be used with MYMPI. All our training examples have been rewritten in Python and work in combination with the Fortran and C examples.

One advantage of pyMPI is that it contains a nearly complete implementation of the MPI standard, about 120 routines. MYMPI contains about 25 of the most important routines of the library. The routines available in the MYMPI module are shown in Table 1.

| | |
|---|---|
| mpi_alltoall - Sends data from all to all processes | mpi_alltoallv - Sends data from all to all processes |
| mpi_barrier - Blocks until all process have reached this routine | mpi_bcast<br>Broadcasts a message |
| mpi_comm_create<br>Creates a new communicator | mpi_comm_dup - Duplicates an existing communicator |
| mpi_comm_group - Accesses the group associated with given communicator | **mpi_comm_rank -** Determines the rank of the calling process |
| **mpi_comm_size -** Determines the size of the group | mpi_comm_split - Creates new communicators |
| mpi_error - Checks for errors (not part of regular MPI) | **mpi_finalize -** Terminates MPI execution |
| mpi_gather - Gathers together values from a group of processes | mpi_gatherv - Gathers into specified locations from all processes |
| mpi_get_count - Gets the number of "top level" elements | mpi_group_include - Produces a group by reordering an existing group |
| mpi_group_rank - Returns the rank of this process | **mpi_init -** Initialize MPI |
| mpi_iprobe - Nonblocking test for a message | mpi_probe - Blocking test for a message |
| **mpi_recv -** Basic receive | mpi_reduce - Reduces values on all processes to a single value |
| mpi_scatter - Sends data from one task to other tasks | mpi_scatterv - Scatters a buffer in parts to all tasks |
| **mpi_send -** Performs a basic send | mpi_status - Return status information (not part of standard MPI) |

**Table 1. The routines available in the MYMPI MPI module.**

MYMPI requires that a MPI library is available on the platform on which it is run. MYMPI has been built and used on a number of different operating systems and different versions of the

MPI library. It was developed under OSX using the LAM and MPICH versions of MPI. It runs on the IBM "SP style" machines using the native MPI. MPICH and LAM have also been used under Redhat. Both MPICH ethernet and MPICH Myrinet have been used under SuSie.

## Applications

### *Continuity*

The driving application for the development of MYMPI was Continuity.

Continuity is a computational tool for continuum problems in bioengineering and physiology, especially those related to cardiac mechanics and electrocardiology research. For example, Continuity can be used to model such phenomena as vetricular arrhthmia. The use of the MYMPI module has made possible investigations into biological phenomena that take place of timescales exceeding one beat, e.g. the effect of sympathetic nervous system activation.

Modeling vetricular arrhthmia involves solving both partial differential equations (PDEs) and ordinary differential equations (ODEs). Currently, the PDE's are solved one time at the beginning of the simulation using a finite element method and the serial version of SuperLU, a linear solver.

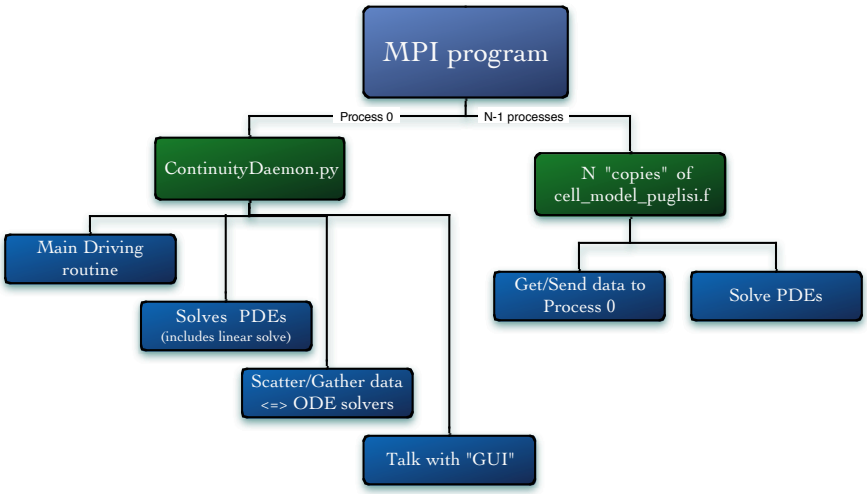


**Figure 1. High level Continuity block diagram.**

Figure 1 shows a high level block diagram of Continuity. It is an MPI application with process 0 written in Python and the other processes written in Fortran. Python is employed for user interfaces, communication, object-oriented component integration and wrapping of computationally efficient FORTRAN functions. Continuity uses the MYMPI module to allow communication between the root Python process and multiple Fortran MPI processes that perform compute-intensive ODE integration over the order of tens or hundreds of thousands of timesteps. This simulates the spread of electrical activation in a 3D finite element model of a heart. A simulation of a section of a rabbit heart uses 1024 elements, 1377 nodes, 8192 gauss points. For this problem the parallelization resulted in a maximum speedup over the serial code of 15x on 32 nodes. As with many simulations we expect the parallel efficiency to improve with

larger problems.

Larger simulations will be limited by root process memory. We plan to extend our use of SuperLU to the distributed version of the package. We will make use of MPI-specific communicator splitting in order to separated the linear solve processes and ODE integration processes.

## Montage Astronomical Mosaicking

The MYMPI module has served as part of the workflow management software developed for the Montage astronomical mosaicking project.

One purpose of the overall project is to create an online virtual observatory. As part of this effort, the three-band infrared Two Micron All Sky Survey (2MASS) is being mosaicked with the Montage software into 6-degree squares. This involves taking potentially many different astronomical images recorded under different conditions and combining them into square images. When doing the combinations corrections must be applied for such things as background. Figure 2 shows an original and corrected image.

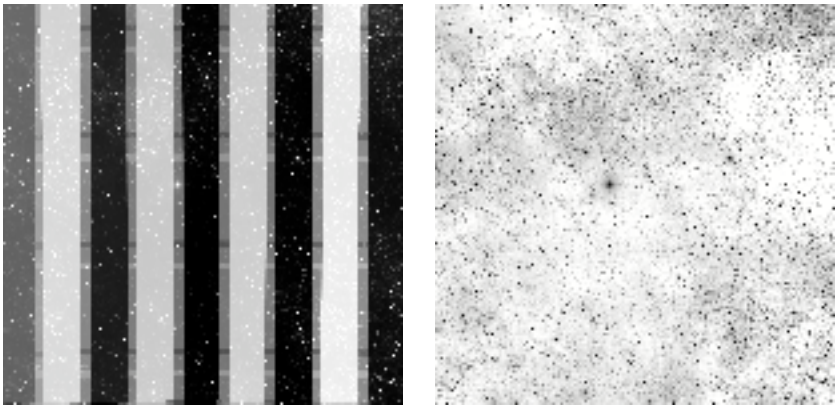

**Figure 2. 2MASS mosaic of a square degree of sky near Galactic center and the corrected image.**

The input to create a single image is 1300-4000 raw data files, each about 2 Mbytes. The output is a single mosaic, a 4 Gbyte file or subdivided tiles of that, eg. 144 tiles each of 27 Mbytes. 5202 such mosaics cover the entire sky for the three bands and will serve as pages in the HyperAtlas, now under construction as part of the National Virtual Observatory (NVO) project to federate disparate astronomical surveys.

The MYMPI module facilitates managing in a single parallel job submission, the production of many mosaics. The MYMPI module enabled us flexibility in managing job steps on many CPUs of a parallel machine. During the execution we alternated between two levels of granularity, that is, a collection of independent single processor tasks and whole machine MPI parallel applications. Some parts of the procedure are serial and some parts can be done in parallel. Say there were N mosaics to be formed and we had N processors. We start with a serial step but the serial step is done for all mosaics simultaneously, one on each CPU. The parallel

step is done for one mosaic at a time, using all the CPUs of the job to work on a single mosaic.

An interesting aspect of this procedure is that MPI used was at both levels. The outer level used Python and the MYMPI mpi module. At this level Python and the module were being used as a scripting, or control language. We used Python's system interaction routines to launch a collection of independent serial applications, one per processor. After these each finished we then launched a C MPI program from within the Python MPI application. The C MPI program used all N of the processors. The Python MPI used the MPICH version of MPI communicating over ethernet and the C MPI program used a Myrinet enabled version of MPI. The system we are using does not support launching Myrinet MPI program from inside another Myrinet MPI program.

### *Workit.py Pipeline Processing*

Workit.py is a simple (140 line) pipeline processing or "workflow" program. It is shipped as part of the MYMPI source package. Workit.py can be used when you need to perform a series of sequential operations on data sets. It uses the MYMPI Python MPI module to spread work across multiple processors. MPI is used for synchronization and convenience of launching multiple processors.

We have used workit.py to post process a collection of output files to produce images and archive the data. The output files were from a 3d earthquake simulation, E3D, in connection with the GEON project. We started with a collection of data files, surface_*.z and wanted to do the following operations:

1. Extract floating point values from a the "raw" data file
2. Create an image from the floating point values file
3. Superimpose a mask on the image
4. Archive the data to long term storage and delete the local copy

Our normal script to process a single file, surface_11700.z, is

```
~/getsubimage < surface_11700.z > surface_11700.flt
~/rgbpng.exe -xmin 1110 -xmax 9984 -ymin 0 -ymax 2000 < surface_11700.flt
surface_11700.png
~/composite -dissolve 15 -tile output.png surface_11700.png
surface_11700_g.png
ls -lt surface_11700* ; hsi "put surface_11700*" ; rm -rf surface_11700*
```

To use workit.py, we created a "commands" file corresponding to the actions shown above by replacing the base part of the filename with DUMMY:

```
~/getsubimage < DUMMY.z > DUMMY.flt
~/rgbpng.exe -p I -xmin 1110 -xmax 9984 -ymin 0 -ymax 2000 < DUMMY.flt
DUMMY.png
~/composite -dissolve 15 -tile output.png DUMMY.png DUMMY_g.png
ls -lt DUMMY* ; hsi "put DUMMY*" ; rm -rf DUMMY*
```

We ran workit.py passing in the names of the raw data files on which to operate. It strips off the suffix of the file name. It then passes the "base" file name from one process to the next and replaces DUMMY. The first MPI process runs the first command on the input file, say surface_11700.z. Then the second MPI process would perform its operation on surface_11700.flt at the same time the first process is working on the second file, surface_11701.z, and so on.

Workit produces a global standard out file that contains information about each file processed. This includes the file number, the start time for that file on process 0 and the end time for that file on the last processor. Workit also saves the stdout and stderr for each command. Each processor produces its own stdout and stderr. This output also contains the timings and actual command for each file.

## Future Work

MYMPI has been used for a number of projects. It is finding additional usage in applications similar to those discussed above. We will continue to add features as requested. We will add better support for character data. We need to test using the new OPEN-MPI version of MPI that is replacing LAM MPI.

## Conclusion

We have developed a Python module MYMPI for creating parallel applications using MPI. It can be used on a variety of parallel platforms and using different MPI libraries. It has been used in several applications and areas of science including, biology, astronomy, and geoscience. We will continue to add features to the module as needed.

The development of MYMPI was done as part of the National Biomedical Computation Resource. NBCR is supported by the National Institutes of Health (NIH) through a National Center for Research Resources program grant P 41 RR08605. MYMPI is available from the NBCR web page at nbcr.sdsc.edu.

## References

GEON - Cyberinfrastructure for the Geosciences: www.geongrid.org
Homepage for SuperLU: www.cs.berkeley.edu/~demmel/SuperLU.html
LAM/MPI Parallel Computing: www.lam-mpi.org
Message Passing Interface Forum: www.mpi-forum.org
MPICH Home Page: www-unix.mcs.anl.gov/mpi/mpich
National Biomedical Computation Resource: nbcr.sdsc.edu
National Virtual Observatory: www.us-vo.org
Open Source High Performance Computing: www.open-mpi.org
pyMPI: http://sourceforge.net/projects/pympi
Python Programming Language: www.python.org