pyMPI – An introduction to parallel Python using MPI*

Patrick Miller

September 11, 2002

Abstract

The interpreted language, Python, provides a good framework for building scripts and control frameworks. While Python has a (co-routining) thread model, its basic design is not particularly appropriate for parallel programming. The pyMPI extension set is designed to provide parallel operations for Python on distributed, parallel machines using MPI.

1 Basic pyMPI

While pyMPI provides a more complete interface to MPI with communicators and advanced functions, it also provides a simplified interface suitable for basic programming. We will examine the unifying concepts that implement this basic interface in the section on communicators below. One of the simplest ways to use pyMPI is interactively, from the prompt. After starting up pyMPI in the normal way for your system (mpirun, prun, poe, etc...) you get what looks like the standard Python prompt (>>>). You are, however, running multiple cooperating processes as in Figure 1. Note that the processes are running asynchronously, so that the values printed from evaluating `mpi.rank` can appear in any order.

The `size` and `rank` attributes of the `mpi` module indicate the number of cooperating tasks and the unique identifier of the task respectively. When you

---

```
% mpirun -np 3 pyMPI
>>> 3
3
3
3
>>> import mpi
>>> mpi.rank
0
2
1
>>>
```

Figure 1: Simple interactions with pyMPI

```
>>> print 'running on',mpi.rank,'of',mpi.size
running on 2 of 3
running on 0 of 3
running on 1 of 3
>>> fp = open('foo%02d.data'%mpi.rank)
>>> fp
<open file 'foo02.data', mode 'r' at 0x81d39a0>
<open file 'foo01.data', mode 'r' at 0x81d5688>
<open file 'foo00.data', mode 'r' at 0x81d4e88>
>>>
```

Figure 2: SPMD in pyMPI

use the rank information, you can make the tasks perform different operations. This is known as SPMD (Single Program, Multiple Data) style parallelism. In Figure 1 we see how the tasks can open different data files.

2   Basic Broadcast and Barrier

To this point, the MPI tasks have not (overtly[1]) acted in a cooperative manner. Parallel programming requires coordination of resources. The simplest way to achieve this is through a barrier call that acts as a rendezvous. In Figure 2 we see how a barrier is used to make sure all tasks have finished work before declaring the work "DONE."

Work can be distributed among tasks using a broadcast scheme.

```
>>> lhs = mpi.bcast(rhs)
```

In pyMPI, the broadcast assures that the value of lhs in all cooperating tasks is the value that rhs had on the so-called *root* task. The root task is, by default,

---

[1]The interactive input is coordinated with a broadcast and synchronizing operation

```
% cat barrier.py
import mpi
data = open('foo%02d.data'%mpi.rank).read()
print 'Work on',mpi.rank
DATA = data.upper()
open('foo%02d.DATA'%mpi.rank,'w').write(DATA)
mpi.barrier()
if mpi.rank == 0:
    print 'DONE'
% mpirun -np 3 pyMPI barrier.py
Work on 0
Work on 2
Work on 1
DONE
%
```

Figure 3: Using a barrier

the one with rank 0 though that can be overridden (e.g. mpi.bcast(rhs,3) would set task 3 to be the root task). The value of rhs is ignored (and is optional) on non-root tasks. Figure 2 illustrates this. The value of `rhs` can be any Python type that is serializable (using the pickle module). This includes basic types, tuples, lists, dictionaries, instances, and others.

Both mpi.barrier and mpi.bcast are *synchronizing* operations. Tasks will block (wait) until every other task performs the operation.

3   Reductions

Bcast is used to send information out from one task to all others. The reverse operation is a *reduction* which collects and processes data from many tasks. MPI has the concept of reduce (collect to a single task) and allreduce (collect to all processes). Like bcast, the reductions are synchronizing. Reductions require a value to operate on and a function to apply. The single task reduce assumes that the root (target) of a reduction is the rank 0 task if it is not provided as an optional third argument. The non-root tasks receive the special Python value `None`. MPI provides some built in operations (see Table 3) pyMPI also allows user defined Python functions to be used.

With these rudimentary operations, we can begin to make more interesting programs. Consider a program to integrate $\int_0^1 \frac{4}{1+x^2}$ (Note that $\frac{1}{1+x^2}$ defines a quarter unit circle in the upper right quadrant, so $\frac{4}{1+x^2}$ has the same area as a unit circle, i.e. π). We can partition the space into a number of small rectangles over which we sum the area.

Our strategy will be to have the master task (rank 0) broadcast the number of rectangles to the other tasks. Each task will create a local sum of its areas.

```
if mpi.rank == 0:
   rhs = << some computation >
   lhs = mpi.bcast(rhs)
else:
   lhs = mpi.bcast()

or

if mpi.rank = 0:
   rhs = << some computation >>
else:
   rhs = None
lhs = mpi.bcast(rhs)
```

Figure 4: Two ways of using mpi.bcast

Table 1: Predefined reductions in (py)MPI

Name	Operation	Example
BAND	Boolean AND	y = mpi.reduce(x,mpi.BAND)
BOR	Boolean OR	y = mpi.reduce(x,mpi.BOR)
BXOR	Boolean XOR	y = mpi.reduce(x,mpi.LXOR)
LAND	Logical AND	y = mpi.reduce(x,mpi.LAND)
LOR	Logical OR	y = mpi.reduce(x,mpi.LOR)
LXOR	Logical XOR	y = mpi.reduce(x,mpi.LXOR)
MAX	Maximum	y = mpi.reduce(x,mpi.MAX)
MAXLOC	First maximum	y,rank = mpi.reduce(x,mpi.MAXLOC)
MIN	Minimum	y = mpi.reduce(x,mpi.MIN)
MINLOC	First minimum	y,rank = mpi.reduce(x,mpi.MINLOC)
PROD	Product	y = mpi.reduce(x,mpi.PROD)
SUM	Sum	y = mpi.reduce(x,mpi.SUM)

```
import mpi
import string
import sys

def f(x): return 4.0/(1.0+x*x)

if mpi.rank == 0:
   n = string.atoi(sys.argv[1])
   mpi.bcast(n)
else:
   n = mpi.bcast()

h = 1.0/n
local_sum = 0.0

for i in range(mpi.rank+1,n+1,mpi.size):
  x = h*(i-0.5)
  y = f(x)
  local_sum += y

global_sum = mpi.reduce(local_sum,mpi.SUM)

if mpi.rank == 0:
  print 'PI is about',h*global_sum
```

Figure 5: Computing π in parallel

Then all tasks will contribute their local sum to a global sum (which should approximate π).

We first write a small Python function that implements the function we are trying to integrate:

```
def f(x): return 4.0/(1.0+x*x)
```

We can have the master get the number of rectangles from the command line:

```
   n = string.atoi(sys.argv[1])
```

where each rectangle will be $h = \frac{1}{n}$ units wide. So the first rectangle's base runs from $[0, h]$, the second from $[h, 2 \cdot h]$, and so on up to the last rectangle at $[(n-1)h, n \cdot h]$. If we choose to evaluate the function at the center of each rectangle, we will be computing $h \cdot f(.5 \cdot h) + h \cdot f(1.5 \cdot h) + \ldots +$ or with a bit of factoring $h \cdot \sum_1^n f((i - .5) \cdot h)$. We partition the rectangles so that the rank 0 task gets $i = 1, 1 + p, 1 + 2p, \ldots$, the rank 1 task gets $2, 2 + p, 2 + 2p, \ldots$, etc. . . (where p is the number of tasks, or mpi.size).

In Figure 3 we bring all these components together to compute π. We can do

5

Table 2: Send/Recv examples

TASK 0	TASK 3	Comments
mpi.send(msg,3)	msg,status = mpi.recv()	Any sender/tag
mpi.send(msg,3,tag=22)	msg,status = mpi.recv()	Any sender/tag
mpi.send(msg,3,tag=22)	msg,status = mpi.recv(0)	Any tag from task 0
mpi.send(msg,3,tag=22)	msg,status = mpi.recv(0,22)	Only tag 22 from task 0
mpi.send(msg,3,tag=22)	msg,status = mpi.recv(tag=22)	Any tag 22 message

somewhat better if we, say, choose to iterate until the error drops to some specific value. Consider Figure 3 in which the master starts with a small number of rectangles (1) and keeps doubling the rectangle count until the computed value stops changing by 1e-6 which happens about 512 rectangles. This version of the program uses `allreduce` instead of `reduce`. This assures that the global_sum is computed on each task. Reduce will return None on non-root tasks, so it can't be multiplied. Notice also that the tasks are synchronizing in each reduce, so a barrier isn't needed to make the printing deterministic.

4 Point-to-point communications

MPI provides explicit point-to-point messaging operations. Messages can be blocking or non-blocking. The primitives are `send`, `recv` (receive), and `sendrecv`. The simplest form of `send` specifies a value and a destination. It returns no value.

```
mpi.send(msg,3)
```

On the receiving end, the receiver can specify:

```
msg, status = mpi.recv()
```

The status value is a structure holding the rank of the sending task and an integer *tag* value which was defaulted to 0 in this example. You may choose to specify the tag when you send. You can also choose to limit reception to a particular sender or for messages with a particular tag. Table 4 shows some examples of how Task 0 and Task 3 can choose to pass messages. In each case, status.source will hold the sender's rank id (here 0) and the tag (0 if it wasn't specified).

Care must be taken to prevent deadlock in sending and receiving messages. Whether or not a particular message actually blocks is a function of the underlying MPI implementation and the size of the message. For instance, sending a small loop back message (i.e. a message to oneself) may work if it is buffered off:

```
>>> mpi.send("hi",0)
>>> msg,status = mpi.recv()
```

```
import mpi
import string
import sys

def f(x): return 4.0/(1.0+x*x)

def computePi(rectangles):
    n = mpi.bcast(rectangles)

    h = 1.0/n
    local_sum = 0.0

    for i in range(mpi.rank+1,n+1,mpi.size):
        x = h*(i-0.5)
        y = f(x)
        local_sum += y

    global_sum = mpi.allreduce(local_sum,mpi.SUM)
    pi = h*global_sum
    return pi


last_pi = 0
n = 1
while 1:
    if mpi.rank == 0:
        print 'Try computing with',n,'rectangles'
    pi = computePi(n)
    error = abs(last_pi - pi)
    if mpi.rank == 0:
        print 'Error is',error
    if error < 1e-6: break
    last_pi = pi
    n *= 2

if mpi.rank == 0:
    print 'Pi is',pi
```

Figure 6: Error bounded π

```
>>> print msg
hi
```

If the string is sufficiently long, then the internal eager buffer will be too small to hold it and pyMPI will block trying to finish the send.

Similarly, consider:

Task 0	Task 3
mpi.send('message 1',3)	mpi.send('message 2',0)
msg,status = mpi.recv(3)	msg,status = mpi.recv(0)

This may block trying to send messages to 3 or 0 since no receive is yet posted for the associated message. Common techniques to fix this issue are to use the MPI function **sendrecv** or to alternate the send/recv pattern.

Task 0	Task 3
msg,status = mpi.sendrecv('message 1',3)	msg,status = mpi.sendrecv('message 2',0)

or

Task 0	Task 3
mpi.send('message 1',3)	msg,status = mpi.recv(0)
msg,status = mpi.recv(3)	mpi.send('message 2',0)

Note that **sendrecv** allows the user to specify both the destination and source in a call:

```
# Send to n+1 and receive from n-1
msg,status = mpi.sendrecv('hi Mom',destination=n-1,source=n+1)
```

5 Non-blocking send and recv

In some cases, a programmer may choose to post a receive well in advance of its receipt (this gives the system a chance to preallocate buffers for instance). In MPI, this is done with the `MPI_Isend` and `MPI_Irecv` calls. In pyMPI, these are exposed as mpi.isend and mpi.irecv. These calls return a *request* object instead of the normal None for send and message/status pair for recv. These request objects are lazy in that they do not block until a message is actually requested. Additionally, a programmer can `test` to see if a message has actually arrived or `wait` which blocks until the message arrives. When a request object is used in a boolean context (e.g. in an if-test), it returns 0 if the message has not completed, and 1 if it does.

Suppose Task 0 has or will send a message to Task 3. Task 3 is expecting messages from task 0 and task 1, so it posts early requests:

```
task0_request = mpi.irecv(0)
task1_request = mpi.irecv(1)
```

Now, Task 3 can test to see if a message has arrived and act accordingly.

```
if task0_request:
  do_zero(task0_request.message)
if task1_request:
  do_one(task1_request.message)
```

```
import mpi
import crypt

if mpi.rank == 0:
  words = open('/usr/dict/words').read().split()
else:
  words = []

local_words = mpi.scatter(words)

target = 'xxaGcwiAKoYgc'
for word in local_words:
  if crypt.crypt(word,target[:2]) == target:
    print 'the word is',word
    break
```

Figure 7: Cracking a password with scatter

The equivalent of MPI_Testany and MPI_Waitany are not wrapped in the current version of pyMPI (1.2a7), but should be in the next release. In the meantime, the programmer can poll:

```
while not ( task0_request or task1_request ):
  if task0_request:
    do_zero(task0_request.message)
  if task1_request:
    do_one(task1_request.message)
```

Care must be taken when requests are destroyed (or go out of scope) before a message is received/sent. Because an internal buffer has been given to MPI, the objects cannot be destroyed until the operation completes or is canceled (e.g. request.cancel()). Send operations cannot be portably canceled. If an non-canceled or incomplete request object is destroyed, then pyMPI will block until the operation completes.

6 Gather/Scatter

Another simple way to achieve parallelism is with gather/scatter parallelism. A scatter operation will take a container, split it into equal (or nearly equal) parts that are messaged to various slave tasks. A gather reverses that and collects sub-containers together into one larger Python list.

We start with a simple example. Suppose we have a list of words we want to use to try to crack a password. We can scatter the list across the tasks and crack the password in parallel. Figure 6 shows one way to crack the password that encodes as xxaGcwiAKoYgc. We have the master read in words from the

```
import mpi

if mpi.rank == 0:
  words = open('/usr/dict/words').read().split()
else:
  words = []
local_words = mpi.scatter(words)

target = 'xxaGcwiAKoYgc'
hits = []
for word in local_words:
  for vowel in 'aeiou':
      if word.count(vowel) != 1: break
  else:
     hits.append(word)

all = mpi.gather(hits)
if mpi.rank == 0:
  for word in all:  print word
```

Figure 8: Search for words using aeiou exactly once

standard dictionary which are then scattered to all the tasks. Each word is tried with the *salt* value of xx (the first two characters of the target are used in the decryption). This program will print out "the word is snake."

The gather operation reverses the scatter with each task providing a sublist that is catenated into the full list. Gather is available as `mpi.gather` and `mpi.allgather` where the former returns the result to the root task and the later returns the value to all tasks. Consider searching the dictionary for words with all five vowels exactly once. See Figure 6 for the code. We scatter the dictionary words as in password cracking example. Here, however, each task is likely to find some words that match the criteria. These are collected in the variable, `hits`. After the search completes, the results are gathered back to the master task which prints out the list which might include such words as ambidextrous, aureomycin, bimolecular, businesswoman, cauliflower, colatitude, communicable, communicate, and consanguine.

7 Basic console output control

pyMPI provides two mechanisms to control output to the console output devices (stdout and stderr). The first is a convenience function that will print its output in rank order. In the earlier examples, output is generated in unpredictable order because the tasks are not synchronous. Examine Figure 7. In this example, when the normal print is encountered, there is no predictable order that the

```
>>> print 'Rank',mpi.rank,'x =',mpi.rank*10
Rank 1 Rank 0 x = 10
x = 0
Rank 3 x = 30 Rank 2

x = 20
>>> mpi.synchronizedWrite('Rank',mpi.rank,'x =',mpi.rank*10,'\n')
Rank 0 x = 0
Rank 1 x = 10
Rank 2 x = 20
Rank 3 x = 30
>>>
```

Figure 9: Using synchronizing write function

tasks will output results. Furthermore, there is no guarantee that the lines
will be atomically output on each task (hence the interspersed output). In the
second clause, the results are guaranteed to be output in rank order by task
0. The synchronizedWrite operation is synchronizing and blocking (hence the
name to alert the programmer that all tasks must call together). Each argument
to the function is converted to a string and joined, space separated. The user
is responsible for including a terminating newline if desired.

The second control that pyMPI provides allows the user to either queue or
discard input on slave tasks (task 0 always outputs to its console).

```
>>> mpi.synchronizeQueuedOutput('/dev/null')
```

This clause will cause output to be discarded except on task 0 (after the flushing
step explained below).

```
>>> mpi.synchronizeQueuedOutput('foobar')
```

This clause will cause output to be queued into the files 'foobar.out.1' ... 'foobar.out.¡size-
1¿' (again, after the flushing step explained below). This keeps the output
clutter low (important if running on hundreds or thousands of processors).

Using `mpi.synchronizeQueuedOutput(None)` will restore the original con-
sole output stream.

If output had already been queued to a local file (the 'foobar' option above)
when `synchronizeQueuedOutput` is invoked, then any output that had been
stored in local files is sent to the master task and output. The temporary files
where data were queued are deleted and then the new mode takes effect.

This feature is commonly used to force non-task 0 output (or error stream
output) to files for postmortem viewing of errors. synchronizeQueuedOutput
takes an optional second argument for controlling the stderr stream. The op-
eration is synchronizing and so all tasks must call together. It is not, however,
necessary that all tasks use the same options. So, one could have the even tasks
output to the console and the odd tasks output to files with

```
>>> if mpi.rank % 2:
...     mpi.synchronizeQueuedOutput('odd_file')
... else:
...     mpi.synchronizeQueuedOutput(None)
...
>>>
```

8 Communicators

In MPI, communicators are handles (opaque labels) that refer to a collection of cooperating MPI tasks. Typical MPI calls in C or FORTRAN pass the handle to an MPI function to perform a task on or across that collection. In pyMPI, on the other hand, communicators are *objects* which define attributes and methods (actions). The most important communicator is the WORLD communicator found at `mpi.WORLD`. Since many simple MPI programs use only the world communicator, the methods of the WORLD communicator are exposed as if they were functions. That is, `mpi.send` is really just `mpi.WORLD.send`. This way simple programs can ignore the communicators. Note that WORLD is **not** the same as MPI_COMM_WORLD to keep Python operations from interfering with MPI messaging in MPI parallel extension code. The COMM_WORLD (i.e. `mpi.COMM_WORLD`) maps onto the standard world communicator.

Communicators have an integer representation that can be passed to C and FORTRAN extensions and used as a MPI_Comm handle. For example, MPICH defines MPI_COMM_WORLD as the integer 91.

```
>>> print int(mpi.COMM_WORLD)
91
>>>
```

Communicators also act like container objects that hold the rank ids of all the tasks. For instance, to send a point-to-point message from task 0 to all *odd* tasks:

```
for rank in mpi.WORLD:
  if rank % 2: mpi.send(msg,rank)
```

You can create sub-communicators using the comm_create method. For instance, build a communicator of size 10 on the first 10 tasks of the world communicator:

```
first_ten = mpi.WORLD.comm_create(mpi.WORLD[:10])
```

Comm_create can be invoked with any sequence object that delivers integers. For example

```
odd_tasks = mpi.WORLD.comm_create( range(1,mpi.size,2) )
```
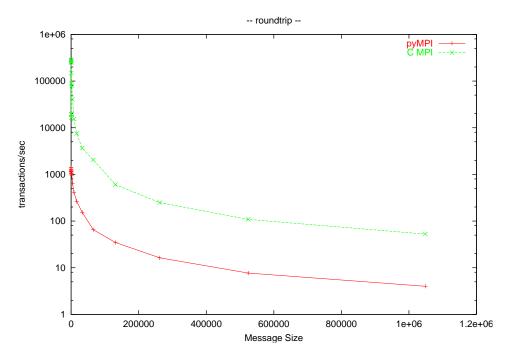
Figure 10: Ping-pong test in C MPI and pyMPI

Do note that creating communicators is a synchronizing operation, so all tasks on a particular communicator must participate (regardless of whether or not they will participate in the new communicator). The comm_create task returns `None` on tasks where the new communicator is not active.

9 Efficiency

pyMPI messages are quite a bit more heavy weight than traditional C or FOR-TRAN messages. Part of this overhead is due to the need for Python to serialized (pickle) every message before sending it. This requires a heap allocation, a copy, and additional processing. However, the pyMPI programmer never needs to explicitly define MPI types! Additionally, since pyMPI cannot preallocate buffers (since the size of messages are not predetermined), it sometimes has to send **two** messages. pyMPI uses a small initial message size intended to be within the MPI eager limit so that small messages can go through without blocking. For large messages, the pyMPI internal buffers are cut into two parts: one inside the eager limit which encodes the size of the message that follows and a second carrying the rest of the message. As Figure 9 shows, pyMPI is about two orders of magnitude slower in sending messages. This is, however, about the same degree of slowness experienced in Python scripts in general.

10   Parting words

pyMPI is still under heavy development. The next generation will encode the remaining MPI-1 functions that make sense to Python and some MPI-2 and MPI-IO features as well. The latest release of the source code is available as open source through

`http://pympi.sourceforge.net`