Parallel Programming with MPI

Overview

In the last guide, we learned some of the basics of parallel programming. This guide will focus on parallel programming with the Message Passing Interface (MPI).

Step 1: Setting up the environment

The first thing we need to do is setup Python to be able to run MPI. We do this with a module called `mpi4py`. To build this module, we will need to be on our newer python version and to have the `openmpi3 lmod` module loaded.

```
[lowell@te-master mpi-examples]$ module list

Currently Loaded Modules:
  1) autotools   2) prun/1.3   3) gnu8/8.3.0   4) openmpi3/3.1.3   5) ohpc
[lowell@te-master mpi-examples]$ python --version
Python 3.7.3
[lowell@te-master mpi-examples]$ pip install mpi4py
Collecting mpi4py
Installing collected packages: mpi4py
Successfully installed mpi4py-3.0.2
```

Make the following test script just to make sure `mpi4py` is working:

```
#!/usr/bin/env python3

from mpi4py import MPI

comm = MPI.COMM_WORLD   # this is the default channel for communicating
with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool
```

```
print("I am {} of {}".format(rank, comm_size))
```

In MPI, the `rank` is a special sequential numeric ID each MPI task is given. The `comm_size` is the total number of tasks in our job.

Now let's try to run it with a couple of different settings:

```
[lowell@te-master mpi-examples]$ mpirun -np 1 mpitest.py
...
I am 0 of 1
```

```
[lowell@te-master mpi-examples]$ mpirun -np 4 mpitest.py
...
I am 0 of 4
I am 2 of 4
I am 3 of 4
I am 1 of 4
```

Looks like it's working.

Step 2: Basic MPI communications

MPI offers a number of different communications that we can do with other ranks in our system. Those ranks may live on the same node with us, or that may communicate over the network fabric (e.g. Infiniband) that MPI is using.

The communications backend protocol is called the "base transport layer", or "BTL". You'll see references to this in various debugging and warning messages.

MPI communications methods can be broken up into two categories: point-to-point communication and collective communication. First, we will go over a couple of examples that use point-to-point communication. We'll look at collective communication in the next step.

send/recv

Let's start with something very simple. Let's have 2 MPI tasks, and have the first task send the second task a message, then print it.

```
#!/usr/bin/env python3
# mpi-send-recv.py

from mpi4py import MPI

comm = MPI.COMM_WORLD   # this is the default channel for communicating
with our MPI run
```

```
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool

if rank == 0:
  msg="Hello hello... is there anybody in there?"
  comm.send(msg, dest=1, tag=42)
  print("I (0) sent this msg: " + msg)
elif rank == 1:
  rmsg = comm.recv(source=0, tag=42)
  print("I (1) got this message: " + rmsg)
else:
  print("I'm rank {}.  I don't get to play right now.".format(rank))
```

So, if we're rank 0 we send a message if we're rank 1 we receive it. Both of us print what we sent/received. The `tag=#` is not required. In more complicated situations, tags can help us be sure we're receiving the message we think we're receiving. Notice that `dest=` and `source=` take a rank number as values.

We also were able to pass a Python data object directly through this message. `mpi4py` works with `pickle` to serialize objects for transfer. If you do happen to be working with raw byte buffers (e.g. Numpy arrays), there are other routines for that in `mpi4py` which are probably a little more efficient.

Let's actually run this example:

```
[lowell@te-master mpi-examples]$ mpirun -np 2 mpi-send-recv.py
...
I (0) sent this msg: Hello hello... is there anybody in there?
I (1) got this message: Hello hello... is there anybody in there?
```

Looks like it worked just fine. If we run with more than 2 tasks we get:

```
[lowell@te-master mpi-examples]$ mpirun -np 3 mpi-send-recv.py
...
I (0) sent this msg: Hello hello... is there anybody in there?
I'm rank 2.  I don't get to play right now.
I (1) got this message: Hello hello... is there anybody in there?
```

It's worth noting that `send`/`recv` are blocking--they'll wait until they succeed. They have non-blocking equivalents, `isend`/`irecv` that will return right away even if they don't send/recieve any data.

Gather

The Gather operation collects all of a certain piece of data to a single task. Let's look at an example:

```
#!/usr/bin/env python3
# mpi-gather.py

from mpi4py import MPI
from random import randint

comm = MPI.COMM_WORLD   # this is the default channel for communicating
with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool

my_random = randint(0,100)

print("{}: my random int is: {}".format(rank,my_random))

rands = comm.gather(my_random, root=0)
if rank == 0:
  print("0: all of the randoms: {}".format(rands))
```

If we run this:

```
[lowell@te-master mpi-examples]$ mpirun -np 4 mpi-gather.py
...
0: my random int is: 42
0: all of the randoms: [42, 48, 83, 57]
1: my random int is: 48
2: my random int is: 83
3: my random int is: 57
```

There are a few things to note about this:

1. *all* of the ranks call `comm.gather`. The ones that are not `root=<#>` send data, while the one that is specified by `root=<#>` collects it. This is common to how all of the collective MPI communications work.
2. when `rank=0` gathers the data, that data always includes its own entry.

Allgather

Allgather is similar to gather, except that all of the nodes get all of the information, i.e. N-to-N collections. Let's modify our example with this:

```
#!/usr/bin/env python3
# mpi-allgather.py

from mpi4py import MPI
from random import randint

comm = MPI.COMM_WORLD   # this is the default channel for communicating
with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool

my_random = randint(0,100)
```

```
print("{}: my random int is: {}".format(rank,my_random))

rands = comm.allgather(my_random)
print("{}: all of the randoms: {}".format(rank, rands))
```

Running this with 4 tasks:

```
[lowell@te-master mpi-examples]$ mpirun -np 4 mpi-allgather.py
...
0: my random int is: 96
1: my random int is: 21
2: my random int is: 98
0: all of the randoms: [96, 21, 98, 80]
1: all of the randoms: [96, 21, 98, 80]
3: my random int is: 80
3: all of the randoms: [96, 21, 98, 80]
2: all of the randoms: [96, 21, 98, 80]
```

MPI allgather is appealing to use if you're used to shared memory systems, because we always have all of the memory as long as we `allgather` frequently. However, this is often very inefficient, and `allgather` should probably generally be avoided.

Up until now we've just run these tests on one node. Let's actually make one of them talk over the Infiniband. This will do an `allgather` with every node generating one random number:

```
[lowell@te-master mpi-examples]$ salloc -N10 -n10 srun --mpi=pmix mpi-allgather.py
salloc: Granted job allocation 142
...
2: my random int is: 11
2: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
8: my random int is: 82
8: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
6: my random int is: 99
6: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
5: my random int is: 98
5: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
7: my random int is: 82
7: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
9: my random int is: 12
9: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
3: my random int is: 85
3: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
1: my random int is: 37
1: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
4: my random int is: 13
4: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
0: my random int is: 55
```

```
0: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
salloc: Relinquishing job allocation 142
```

Scatter

A scatter is essentially the inverse of a gather. In a scatter operation, one node has a list of objects, and randomly distributes those objects to all of the nodes. This can be very useful if you want a "work queue" style model. A controlling task can scatter work that needs to be done to worker tasks.

```
#!/usr/bin/env python3
# mpi-scatter.py

from mpi4py import MPI
from random import randint

comm = MPI.COMM_WORLD   # this is the default channel for communicating
with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool

if rank == 0:
  d = list()
  for i in range(0,comm_size):
    d.append(randint(1,100))
  print("0: generated {}".format(d))
else:
  d = None

d = comm.scatter(d, root=0)

print("{}: got {}".format(rank,d))
```

Running:

```
[lowell@te-master mpi-examples]$ salloc -N10 -n10 srun --mpi=pmix mpi-
scatter.py
salloc: Granted job allocation 143
...
1: got 18
0: generated [87, 18, 47, 39, 79, 57, 29, 65, 36, 52]
0: got 87
8: got 36
7: got 65
5: got 57
6: got 29
4: got 79
9: got 52
3: got 39
```

```
2: got 47
salloc: Relinquishing job allocation 143
```

Like the other operations, we see that `rank=0` also gets a number from the scatter list.

Broadcast

We'll look at one last operation: broadcast. A broadcast operation sends a piece of data to everyone. Let's modify our scatter to be a broadcast:

```
#!/usr/bin/env python3
# mpi-bcast.py

from mpi4py import MPI
from random import randint

comm = MPI.COMM_WORLD   # this is the default channel for communicating
with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool

if rank == 0:
  d = list()
  for i in range(0,comm_size):
    d.append(randint(1,100))
  print("0: generated {}".format(d))
else:
  d = None

d = comm.bcast(d, root=0)

print("{}: got {}".format(rank,d))
```

```
[lowell@te-master mpi-examples]$ salloc -N10 -n10 srun --mpi=pmix mpi-
bcast.py
salloc: Granted job allocation 145
...
0: generated [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
0: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
8: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
4: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
1: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
9: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
5: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
6: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
2: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
7: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
3: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
salloc: Relinquishing job allocation 145
```

Those are most of the MPI operations one would generally use, though the spec of operations is quite large. As we can see, MPI allows us to trivially pass data around. MPI also has methods we did not go over for things like task synchronization (e.g. MPI.Barrier, where all tasks pause until everyone has called Barrier). Not only does MPI make it easy to pass data around, it does it efficiently over various different transport layers like Infiniband.

Step 3: The MPI of `gas.py`

Perhaps the easiest way to see how this message passing can be used to make a large application spread over lots of systems is to simply look at an example. Fortunately, we have one. There is an MPI + Threading version of `gas.py` in the `mpi` branch:

```
comm = MPI.COMM_WORLD   # this is the default channel for communicating
with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool


class Sym(threading.Thread):
    def boundary_resolve(self):
        c = 0
        for p in iter(self.w.oob):
            p.boundary_resolve_hard(self.w.box)
            c += 1
        print("\treflected {} particles of {} in the oob list".format(c,
len(self.w.oob)), file=sys.stderr)

    def clear_oob(self):
        self.w.oob.clear()
        print("oob list cleared")

    def run_cell(self, c):
        print("\t{:0.4f} comb particles into cells".format(since(self.w)),
file=sys.stderr)
        c.comb_particles()
        print("waiting for comb to complete")
        self.b.wait() # once we're done combing, we have to clear all
leftover oob particles; they must have gone to a different rank
        print("\t{:0.4f} start integrate".format(since(self.w)),
file=sys.stderr)
        c.integrate()
        print("\t{:0.4f} start collide".format(since(self.w)),
file=sys.stderr)
        c.collide()
        print("\t{:0.4f} start boundary check".format(since(self.w)),
file=sys.stderr)
        c.boundary_check()

    def print(self):
        for c in self.cells:
            c.print(self.w.out)
    def run(self):
        for i in range(0,threads):
            zrange = (self.w.box.z.high - self.w.box.z.low)/threads
            yrange = (self.w.box.y.high - self.w.box.y.low)/comm_size # we
want things comm_size * threads
            # We slice the z-dimension by number of threads, and the y
dimension by comm size
            c = Cell(i, self.w, int(parts/threads), Box(self.w.box.x,
Range(rank*yrange, (rank+1)*yrange), Range(i*zrange, (i+1)*zrange)))
            self.cells.append(c)

        # create a cell
        print("Running with {} threads, {} comm_size".format(threads,
comm_size))
        print("My MPI rank is {}".format(rank))
        print("{:0.4f} running with {} particles for {} steps, output to
{}".format(since(self.w), parts, steps, out.name), file=sys.stderr)

        print("--- ts=0", file=out)
        self.print()

        for i in range(1, steps+1):
            tds = list()
            for c in self.cells:
                print("{:0.4f} timestep={}, particles=
{}".format(since(self.w), i, len(c.parts)), file=sys.stderr)
                tds.append(threading.Thread(target=self.run_cell, args=
(c,)))

            self.b.reset()
            for td in tds:
                print("starting thread")
                td.start()

            for td in tds:
                td.join()

            all_oob = comm.gather(self.w.oob, root=0)
            print(all_oob)
            self.w.oob: List[Particle] = []
            for i in all_oob or []:
                for j in iter(i) or []:
                    self.w.oob.append(j)

            if rank == 0:   # rank 0 gathers all oob and computes boundary
conditions for the whole world
                print("\t{:0.4f} resolve out-of-bounds
particles".format(since(self.w)), file=sys.stderr)
                self.boundary_resolve()

            # how broadcast the list to everyone else, post boundary
conditions
            self.w.oob = comm.bcast(self.w.oob, root=0)
            print("\t{:0.4f} start print".format(since(self.w)),
file=sys.stderr)
            print("--- ts={}".format(i), file=out)
            self.print()

        print("{:0.4f} done".format(since(self.w)), file=sys.stderr)
        out.close()


    def __init__(self):
        self.b = threading.Barrier(threads, action=self.clear_oob)
        self.cells = list()
        self.w = World(steps=steps, out=out, parts=parts, cells=cells)
```

Our simple `gas.py` simulator clearly got a little more complicated, but not dramatically so. There are a couple of things we had to do to map in MPI:

1. we divide cells now not only in the z-axis for threads, but also in the y-axis for rank (in exactly the same way). So, our cells are like long rectangular rods.
2. when a particle gets on the out-of-bounds list when the cell does a boundary check, we need to potentially transfer it to a neighbor. Here's how we do that:
    1. we gather all `oob` lists to `rank=0`
    2. `rank=0` does total world size boundary conditions
    3. once boundary conditions are satisfied, `rank=0` broadcasts the (potentially modified) `oob` list to all ranks
    4. each cell combs for its particles as usual
    5. ...but there's a complication

When the cells comb for their particles, because some particles will belong to other ranks, any particular rank will not have an empty list of `oob` particles by the end of the combing process. For this reason, we have to clear the list manually. But, to do this, we need to be sure that all cells (threads) in our rank are done combing.

To ensure this, we implement a thread "barrier". The barrier, once called, blocks until all threads have called it. Once all threads have called it, it executes the `action=` specified by:

```
self.b = threading.Barrier(threads, action=self.clear_oob)
```

This is just a function that clears the `oob` list.

So, we see that going to MPI + Threads required a bit more bookkeeping, but ultimately isn't that much harder than our earlier versions.

Here's how the lines of code break down for code in `Sym`:

- single-threaded: 29 lines
- multi-threaded: 61 lines

- MPI + threaded: 84 lines

Finally, let's run the `gas.py` in MPI mode.

```
[lowell@te-master gas]$ time mpirun -np 2 ./gas.py -t2 -p 1024 -s4 -o sym
...
real    0m12.349s
user    0m19.839s
sys    0m0.416s
```

Try running it across more than one node. Does it work best to use MPI or threads or both?

Some experimenting shows it doesn't scale very well to large ranks. Trying to run across the whole cluster, for instance, tends to break.