Introduction to MPI using `mpi4py`

Prof. Jeremy Roberts

Fall 2015

Resources

- https://mpi4py.scipy.org/docs/
- W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed., MIT Press (1999)
- “Parallel Programming for Multicore Machines Using OpenMP and MPI” as taught by Dr. Evangelinos, MIT OpenCourseWare (2010)
  - ocw.mit.edu/courses/ earth-atmospheric-and-planetary-sciences/ 12-950-parallel-programming-for-multicore-machines-usi

Getting MPI

It’s as easy as:

```
conda install mpi4py
```

Check that it works: open the Python interpreter and try `from mpi4py import MPI`. If you get no errors, you have MPI.

MPI Programming Model

- MPI defines an application programmer interface (API) for distributed-memory, parallel programming based on the "message-passing" model
- Processes operate within *different* address spaces (i.e., distributed memory)
- Coarse- (and some fine-grain) parallelism is supported
- Each process is a unique instance of the executable, and all transfer of data is done by explicit sends, receives, and their equivalents.
- Long-term development and support for C/C++ and Fortran (i.e., MPI will be around for a long time)

Hello World

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

print "hello from rank ", rank, " of ", size
```

hello.py

To run with 2 processes: `mpirun -np 2 python hello.py`

Point-to-Point Communication

```python
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
   data = {'a': 7, 'b': 3.14, 'c': np.ones(3)}
   comm.send(data, dest=1, tag=11)
elif rank == 1:
   data = comm.recv(source=0, tag=11)
   print "rank 1 has data: ", data
```

point_to_point.py

Point-to-Point Communication for NumPy Arrays

```
from mpi4py import MPI
import numpy as np
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
# pass explicit MPI datatypes
if rank == 0:
   data = np.linspace(0, 1, 10, dtype=np.float64)
   comm.Send([data, MPI.DOUBLE], dest=1, tag=77)
elif rank == 1:
   data = np.empty(10, dtype=np.float64)
   comm.Recv([data, MPI.DOUBLE], source=0, tag=77)
   print "rank 1 has ", data
# automatic MPI datatype discovery
if rank == 0:
   data = np.arange(10, dtype=np.float64)
   comm.Send(data, dest=1, tag=13)
elif rank == 1:
   data = np.empty(10, dtype=np.float64)
   comm.Recv(data, source=0, tag=13)
   print "rank 1 has ", data
```

point_to_point_numpy.py

Exercise: For a large array (e.g., $n = 10^5$), compare the speed of `send/recv` to `Send/Recv`.

Collective Communication (a Convenience, Really)

```
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank, size = comm.Get_rank(), comm.Get_size()
# initialize data to be communicated
if rank == 0:
   dataB = 123, [(i+1)**2 for i in range(size)]
else:
   dataB, dataS = None, None
dataG = (rank+1)**3
# collectively move the data
dataB = comm.bcast(dataB, root=0)
dataS = comm.scatter(dataS, root=0)
dataG = comm.gather(dataG, root=0)
# print the results on a single process
if rank == 0 :
    print dataB, dataS, dataG
```

collective.py

Exercise: Using two processes, observe the output produced if process 0 or process 1 prints the output. Can you determine from that what `bcast`, `scatter`, and `gather` are doing?
Exercise: Implement `bcast`, `scatter`, and `gather` using the `send` and `recv`.

Simplest Real-World Use: Sum an Array

```python
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank, size = comm.Get_rank(), comm.Get_size()
# compute the sum s of the numbers 0...n-1
n, s = 1000, 0.0
# compute starting and ending indices
i_s = rank*(n/size)
i_e = i_s+(n/size)
if rank == size - 1 :
    i_e = n
# perform the sum and have process 0 get the sum
for i in range(i_s, i_e) :
    s += i
comm.reduce(s, s, op=MPI.SUM, root=0)
# print the result on process 0
if rank == 0 :
    print "sum is ", s, " expected ", n*(n-1)/2
```

sum.py

Exercise: Note that the last process can have up to nearly *double* the work of all other processes. Modify this code so that the extra work is spread evenly across the processes.

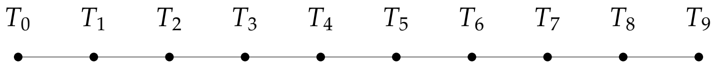
Another Application: Domain Decomposition

Recall the finite-difference approximation we used to solve the 1-D heat equation:

$$-k\frac{d^2T}{dx^2} = Q \Longrightarrow -\frac{T_{i+1} - 2T_i + T_{i-1}}{\Delta^2} = \frac{Q}{k}$$

with appropropriate boundary conditions. For simplicity, let's assume $T_0 = T_n = 0$.

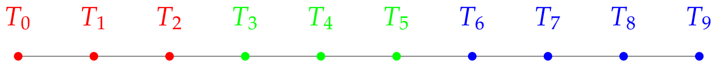
For $n = 9$, the mesh for the problem can be visualized as follows:



What we need is a way to *decompose* this mesh so that several processes can do the work.

Another Application: Domain Decomposition

For three processes, the mesh can be divided up as follows:



Consider T_2. The finite difference equation indicates that process 0 needs the value of T_3 to update T_2, but process 1 is in charge of T_3!

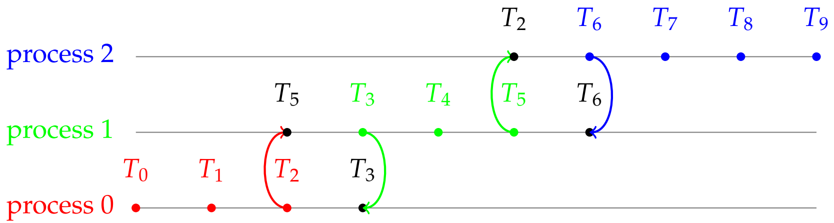
Options:

1. Each process has a full T array and communicates it to neighbors after each iteration.
2. Each process has an array of size equal to the number of temperatures it computes (here, 5), plus one or two extra "ghost" cells for neighbor data.

Option 1 is easy to code but (a) sends way too much data every time and (b) requires more memory-per-process than is needed. Option 2 is harder to code but minimizes communication and memory requirements. *We'll, of course, go with the second option.*

Another Application: Domain Decomposition

A schematic for decomposing the mesh and communicating boundary information is shown below:



Therefore, each process but the last must send to its right, and each process but the first must send to its left.

Likewise, each process but the last must receive from its right neighbor, and each process but the first must receive from its left neighbor.

Handling Ghost Cells: a Python Snippet

```
    # send left
    if rank > root :
        comm.send(T[i_s], dest=rank-1, tag=111)
    if rank < last :
        T[i_e] = comm.recv(source=rank+1, tag=111)
    # send right
    if rank < last :
        comm.send(T[i_e-1], dest=rank+1, tag=222)
    if rank > root :
        T[i_s-1] = comm.recv(source=rank-1, tag=222)
```

ghost_cell_snippet.py

where `i_s` and `i_e` are starting and ending indices of the *local temperature array*. For example, process 3 computes temperatures `T[1:4]` (where the local 1 and 4 map onto the global 3 and 6, respectively). Then, process 3 sends `T[1]` and `T[3]` (globally, T_3 and T_5), and receives `T[0]` and `T[4]` (globally T_2 and T_6).

Exercise: Switch the first two if statements and run your code with three processes. What happens and why? The phenomenon is called *deadlock* and is as common a bug for MPI programs as are race conditions in OpenMP programs.

Hints for the Full Implementation

- Think carefully about how to handle (the physical) boundary conditions. Not all processes will have such a condition to handle.
- If you terminate the iteration when the difference of successive temperatures is below some threshold τ (e.g., $\max(|\mathbf{T} - \mathbf{T}^{\text{old}}|) < \tau$), then each process needs the same difference for comparison to a tolerance. Otherwise, some processes might keep iterating, and your code will hang when they try to send to and receive from processes no longer iterating.
- Ideally, you'll get the full, final temperature array onto the root process for plotting and such. You've seen a few functions for doing this, but `gather` might be a clean approach.
- In 2-D, you'll need whole rows of ghost cells, so think carefully about your indices.