# 03.1_Distributed_Computing

September 27, 2014

# 1 3.1 Distributed Computing



## 1.1 Acknowledgements

- Much of this tutorial uses slide material from William Gropp, University of Illinois and Lisandro Dalcin, CONICET
- mpi4py is a Cythonized wrapper around MPI originally developed by Lisandro Dalcin, CONICET

## 1.2 Notebook Engine Setup

Following the recommendations in PEP 8, we import all Python modules at the beginning of the notebook.

```
In [5]: %load_ext parallelmagic
        %pylab inline --no-import-all
```

```
The parallelmagic extension is already loaded. To reload it, use:
  %reload_ext parallelmagic
Populating the interactive namespace from numpy and matplotlib
```

```
In [6]: import os
        from mpi4py import MPI
        import numpy as np
        from matplotlib import pyplot
```

## 1.3 Connecting to MPI via IPython

To run the examples in parallel, you must run the command:

```
ipcluster start --engines=MPI --n 4
```

Or use notebook Appendix_Launch_MPI_Engines (if you are using the VMs or have this configured for your environment).

Then execute the next cell:

```
In [7]: from IPython.parallel import Client
        c = Client()
        view = c[:]

        view.activate()
        view.block = True
```

There are now 5 IPython Engines (also sometimes called kernels) running: 1 directly connected to the Notebook and 4 started by `ipcluster` and connected via the Client interface.

### 1.4 %autopx and the ipcluster Engines

For interactive convenience, we will use the `autopx` magic from the `parallelmagic` extensions. We can use the `autopx` magic to execute cells on the ipcluster Engines instead of the Notebook Engines.

```
In [8]: os.getpid()
```

```
Out[8]: 22158
```

```
In [9]: %autopx
```

```
%autopx enabled
```

Recall that when we used `import` earlier it was on the Notebook Engine. Before we can call `os.getpid` on the ipcluster Engines, we need to import `os` and any other modules we would like to use.

```
In [11]: import os
         #We will use mpi4py, numpy, and math later
         from mpi4py import MPI
         import numpy as np
         import math
```

```
In [12]: os.getpid()
```

```
Out[0:3]: 22398


Out[1:3]: 22399


Out[2:3]: 22400


Out[3:3]: 22397
```

# 2 A Quick Review of Concepts of Scalability

# 3 The Multiple Forms of Parallelism

- **instruction** - multiple program instructions are simultaneously dispatched in a pipeline or to multiple execution units (superscalar)
- **data** - the same program instructions are carried out simultaneously on multiple data items (SIMD)
- **task** - different program instructions on different data (MIMD)
- **collective** - single program, multiple data, not necessarily synchronized at individual operation level (SPMD)

*This part of the tutorial focuses on data and collective parallelism*

# 4 Parallel Programming Paradigms

- a parallel programming paradigm is a specific approach to exploiting parallelism in hardware
- many programming paradigms are very tightly coupled to the hardware beneath!
- CUDA assumes large register files, Same Instruction Multiple Thread parallelism, and a mostly flat, structured memory model, matching the underlying GPU hardware
- OpenMP exposes loop level parallelism with a fork/join model, assumes the presence of shared memory and atomics
- OpenCL tries to generalize CUDA, but still assumes a 'coprocessor' approach, where kernels are shipped from a master core to worker cores

## 4.1 The Message Passing Model

- a process is (traditionally) a program counter for instructions and an address space for data
- processes may have multiple threads (program counters and associated stacks) sharing a single address space
- message passing is for communication among processes, which have separate address spaces
- interprocess communication consists of
- synchronization
- movement of data from one process's address space to another's

## 4.2 Why MPI?

- **communicators** encapsulate communication spaces for library safety
- **datatypes** reduce copying costs and permit heterogeneity
- multiple **communication modes** allow more control of memory buffer management
- extensive **collective operations** for scalable global communication
- **process topologies** permit efficient process placement, user views of process layout
- **profiling interface** encourages portable tools

It Scales!

## 4.3 MPI - Quick Review

- processes can be collected into **groups**
- each message is sent in a **context**, and must be received in the same context
- a **communicator** encapsulates a context for a specific group
- a given program may have many communicators with any level of overlap
- two initial communicators
- `MPI_COMM_WORLD` (all processes)
- `MPI_COMM_SELF` (current process)

In Python, these communicators are `MPI.COMM_WORLD and MPI.COMM_SELF`

## 4.4 Communicator, Rank, and Size Setup

We'll be using the `MPI.COMM_WORLD` communicator as `comm` for the remainder of this notebook. It's also very common to use the communicator's associated rank and size attributes as `rank` and `size`. We'll assign these variables now to simplify the readability of the code.

```
In [18]: comm = MPI.COMM_WORLD
         size = comm.Get_size()
         rank = comm.Get_rank()
```

## 4.5 First Example: Hello World

```
In [19]: # set up basic variables
         name = MPI.Get_processor_name()
         pid = os.getpid()
```

```
In [20]: print("Hello World! I am process %d of %d on %s with pid %d.\n" % (rank, size, name, pid))
```

```
[stdout:0]
Hello World! I am process 1 of 4 on Arons-MacBook-Pro.local with pid 22398.

[stdout:1]
Hello World! I am process 2 of 4 on Arons-MacBook-Pro.local with pid 22399.

[stdout:2]
Hello World! I am process 3 of 4 on Arons-MacBook-Pro.local with pid 22400.

[stdout:3]
Hello World! I am process 0 of 4 on Arons-MacBook-Pro.local with pid 22397.
```

Note that the MPI rank is not necessarily synchronized with the IPython view rank.

## 4.6 Communicators

- processes can be collected into **groups**
- each message is sent in a **context**, and must be received in the same context
- a **communicator** encapsulates a context for a specific group
- a given program may have many communicators with any level of overlap
- two initial communicators
- `MPI_COMM_WORLD` (all processes)
- `MPI_COMM_SELF` (current process)

## 4.7 Datatypes

- the data in a message to send or receive is described by address, count and datatype
- a datatype is recursively defined as:
- predefined, corresponding to a data type from the language (e.g., `MPI_INT`, `MPI_DOUBLE`)
- a contiguous, strided block, or indexed array of blocks of MPI datatypes
- an arbitrary structure of datatypes
- there are MPI functions to construct custom datatypes

## 4.8 Tags

- messages are sent with an accompanying user-defined integer tag to assist the receiving process in identifying the message
- messages can be screened at the receiving end by specifying the expected tag, or not screened by using `MPI_ANY_TAG`

## 4.9 mpi4py Functionality

- Implements up to MPI-3 with underlying MPI implementation support
- Generic API with lowercase function names, e.g. `Comm.send`
- Efficient API with titlecase function names, e.g. `Comm.Send`
- The efficient API can still handle default arguments and type discovery for NumPy arrays and PEP-3118 buffers

## 4.10 Job Startup

- To launch: mpirun –np NP python script_name
- IPython automatically handles calling `mpirun` for you with the `ipcluster` command

## 4.11 Initialization

`mpi4py` automatically calls `MPI_Init()` and `MPI_Finalize()`

- `MPI_Init()` is called when you import the `MPI` module from `mpi4py`
- `MPI_Finalize()` is called before the Python process ends
- If you need explicit control, use the `mpi4py.rc` module to configure before importing `MPI`

## 4.12 MPI Basic (Blocking) Send

**C**

```
int MPI_Send(void* buf, int count, MPI_Datatype type,
int dest, int tag, MPI_Comm comm)
```

**mpi4py**

```
Comm.Send(self, buf, dest=0, tag=0)
Comm.send(self, obj=None, dest=0, tag=0)
```

## 4.13 MPI Basic (Blocking) Recv

**C**

```
int MPI_Recv(void* buf, int count, MPI_Datatype type,
int source, int tag, MPI_Comm comm, MPI_Status status)
```

**mpi4py**

```
comm.Recv(self, buf, source=0, tag=0, status=None)
comm.recv(self, obj=None, source=0, tag=0, status=None)
```

## 4.14 Send/Receive Example

### 4.14.1 Generic

```
In [23]: if rank == 0:
             data = {'a': 7, 'b': 3.14}
             comm.send(data, dest=1, tag=11)
         elif rank == 1:
             data = comm.recv(source=0, tag=11)
             print data
```

```
[stdout:0] {'a': 7, 'b': 3.14}
```

### 4.14.2 Efficient

#### Explicit MPI Datatypes

```
In [27]: if rank == 0:
             data = np.arange(10, dtype='i')
             comm.Send([data, MPI.INT], dest=1, tag=77)
         elif rank == 1:
             data = np.empty(10, dtype='i')
             comm.Recv([data, MPI.INT], source=0, tag=77)
             print data
```

```
[stdout:0] [0 1 2 3 4 5 6 7 8 9]
```

#### Automatic MPI Datatype Discovery (NumPy arrays)

```
In [29]: 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 data
```

```
[stdout:0] [ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
```

## 4.15 Synchronization

### C

```
int MPI_Barrier(MPI_Comm comm)
```

### mpi4py

```
comm.Barrier(self)
comm.barrier(self)
```

```
In [30]: for r_id in range(comm.Get_size()):
             if rank == r_id:
                 print "Hello from proc:", rank
             comm.Barrier()
```

```
[stdout:0] Hello from proc: 1
[stdout:1] Hello from proc: 2
[stdout:2] Hello from proc: 3
[stdout:3] Hello from proc: 0
```

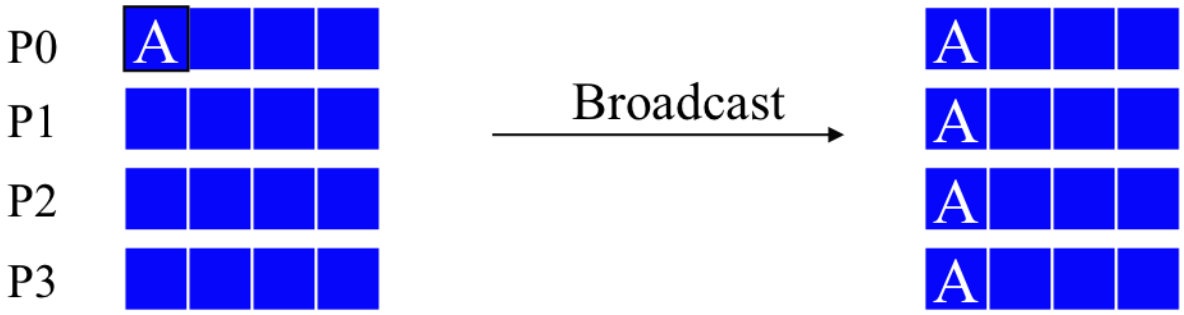
### 4.16 Timing and Profiling

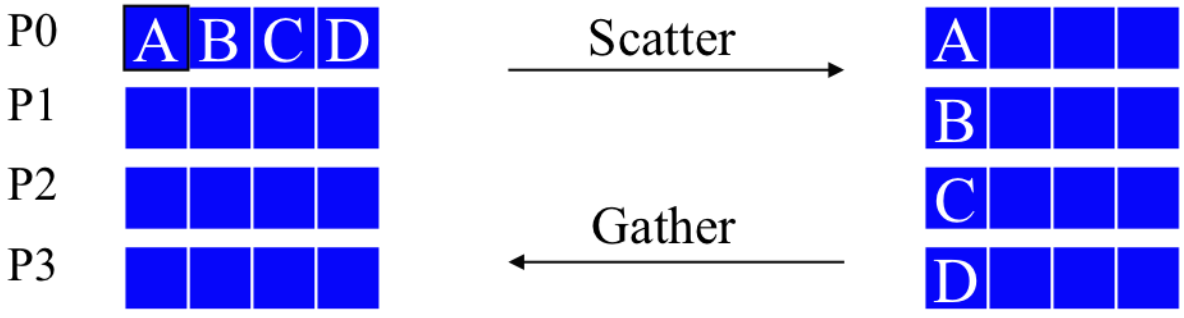
The elapsed (wall-clock) time between two points in an MPI program can be computed using MPI_Wtime:

```
In [31]: t1 = MPI.Wtime()
         t2 = MPI.Wtime()
         print("time elapsed is: %e\n" % (t2-t1))
```

```
[stdout:0]
time elapsed is: 9.230500e-05

[stdout:1]
time elapsed is: 9.876500e-05

[stdout:2]
time elapsed is: 9.861000e-05

[stdout:3]
time elapsed is: 4.349900e-05
```

### 4.17 Basic Collectives: Broadcast, Scatter, and Gather




**C**

```
int MPI_Bcast(void *buf, int count, MPI_Datatype type,
int root, MPI_Comm comm)
```

**mpi4py**

```
comm.Bcast(self, buf, root=0)
comm.bcast(self, obj=None, root=0)
```

## 4.18 Broadcast Example

```
In [54]: if rank == 0:
             data = {'key1' : [7, 2.72, 2+3j],
                     'key2' : ( 'abc', 'xyz')}
         else:
             data = None
         data = comm.bcast(data, root=0)
         print "bcast finished and data \
          on rank %d is: "%comm.rank, data
```

```
[stdout:0] bcast finished and data  on rank 0 is:  {'key2': ('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
[stdout:1] bcast finished and data  on rank 1 is:  {'key2': ('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
[stdout:2] bcast finished and data  on rank 3 is:  {'key2': ('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
[stdout:3] bcast finished and data  on rank 2 is:  {'key2': ('abc', 'xyz'), 'key1': [7, 2.72, (2+3j)]}
```

## 4.19 Scatter Example:

```
In [55]: if rank == 0:
             data = [(i+1)**2 for i in range(size)]
         else:
             data = None
         data = comm.scatter(data, root=0)
         assert data == (rank+1)**2
         print "data on rank %d is: "%comm.rank, data
```

```
[stdout:0] data on rank 0 is:  1
[stdout:1] data on rank 1 is:  4
[stdout:2] data on rank 3 is:  16
[stdout:3] data on rank 2 is:  9
```

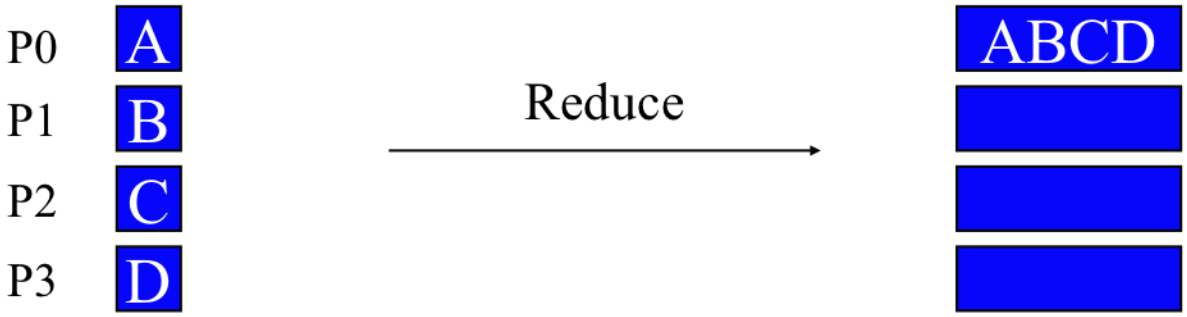
## 4.20 Gather (and Barrier) Example:

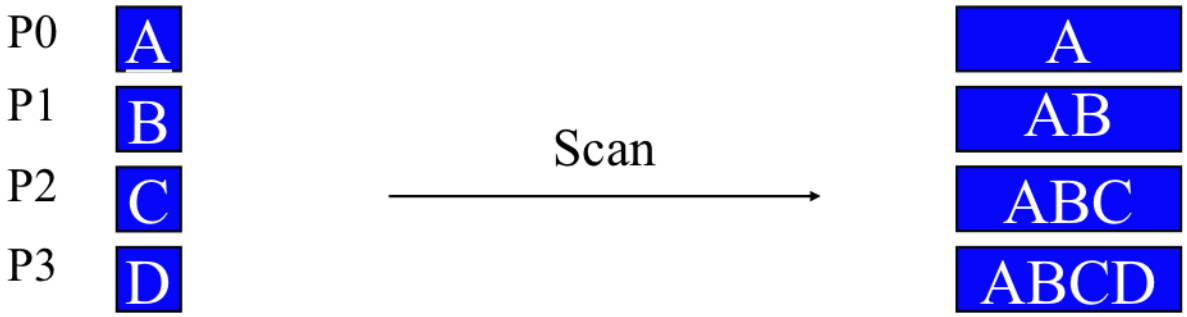
```
In [56]: data = (rank+1)**2
         print "before gather, data on \
           rank %d is: "%rank, data

         comm.Barrier()
         data = comm.gather(data, root=0)
         if rank == 0:
             for i in range(size):
                 assert data[i] == (i+1)**2
         else:
             assert data is None
         print "data on rank: %d is: "%rank, data
```

```
[stdout:0]
before gather, data on   rank 0 is:  1
data on rank: 0 is:  [1, 4, 9, 16]
[stdout:1]
before gather, data on   rank 1 is:  4
```

```
data on rank: 1 is:  None
[stdout:2]
before gather, data on   rank 3 is:  16
data on rank: 3 is:  None
[stdout:3]
before gather, data on   rank 2 is:  9
data on rank: 2 is:  None
```

### 4.21 Reduce and Scan




### 4.22 Reduce Example

The lower-case `reduce` implemented in `mpi4py` is not designed to be particularly scalable. If you need to perform a reduce on thousands of processes or more, it is recommended that you either switch to the non-generic Reduce, or utilize the scalable reduce provided in `mpi4py/demo/reductions/reductions.py`

```
In [57]: sendmsg = comm.rank
         recvmsg1 = comm.reduce(sendmsg, op=MPI.SUM, root=0)
         recvmsg2 = comm.allreduce(sendmsg)
         print recvmsg2

[stdout:0] 6
[stdout:1] 6
```

```
[stdout:2] 6
[stdout:3] 6
```

## 4.23 Compute Pi Example

The following example is completely self-contained to simplify reuse in another script. You can switch between running the code in parallel and serial by executing an %autopx cell.

```
In [33]: from mpi4py import MPI
         import math

         def compute_pi(n, start=0, step=1):
             h = 1.0 / n
             s = 0.0
             for i in range(start, n, step):
                 x = h * (i + 0.5)
                 s += 4.0 / (1.0 + x**2)
             return s * h

         comm = MPI.COMM_WORLD
         nprocs = comm.Get_size()
         myrank = comm.Get_rank()
         if myrank == 0:
             n = 10
         else:
             n = None

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

         mypi = compute_pi(n, myrank, nprocs)

         pi = comm.reduce(mypi, op=MPI.SUM, root=0)

         if myrank == 0:
             error = abs(pi - math.pi)
             print ("pi is approximately %.16f\nerror is %.16f" % (pi, error))
```

```
[stdout:3]
pi is approximately 3.1424259850010983
error is 0.0008333314113051
```

## 4.24 Mandelbrot Set Example

The following example is completely self-contained to simplify reuse in another script.

```
In [41]: from mpi4py import MPI
         import numpy as np

         def mandelbrot (x, y, maxit):
             c = x + y*1j
             z = 0 + 0j
             it = 0
             while abs(z) < 2 and it < maxit:
                 z = z**2 + c
                 it += 1
```

```
        return it

x1, x2 = -2.0, 1.0
y1, y2 = -1.0, 1.0
w, h = 250, 200
maxit = 127

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

# number of rows to compute here
N = h // size + (h % size > rank)

# first row to compute here
start = comm.scan(N)-N

# array to store local result
Cl = np.zeros([N, w], dtype='i')

# compute owned rows

dx = (x2 - x1) / w
dy = (y2 - y1) / h

for i in range(N):
    y = y1 + (i + start) * dy
    for j in range(w):
        x = x1 + j * dx
        Cl[i, j] = mandelbrot(x, y, maxit)

# gather results at root (process 0)
counts = comm.gather(N, root=0)
C = None
if rank == 0:
    C = np.zeros([h, w], dtype='i')

# here we create a custom datatype for sending/receiving rows of data.
rowtype = MPI.INT.Create_contiguous(w)
rowtype.Commit()

comm.Gatherv(sendbuf=[Cl, MPI.INT], recvbuf=[C, (counts, None), rowtype],root=0)

rowtype.Free()
```

We can't inline plots from the `ipcluster` Engines where we just performed the computations. Instead, we use the Notebook Engine to get a copy of the data on MPI rank 0, then plot as before.

First, we switch off `%autopx` to enable computing on the Notebook Engine.

```
In [42]: %autopx
```

```
%autopx disabled
```

Then we collect the array and rank data from the `ipcluster` Engines using the `view` object.

In [43]:
```
# CC is a list of C from all ranks
CC = view['C']
# Similarly, ranks is a list of MPI ranks from all ipcluster processes
ranks = view['rank']

# Do the plotting
# We use the IPython index of the ipcluster process with rank 0 as
# the index into CC
pyplot.imshow(CC[ranks.index(0)])
pyplot.spectral()
pyplot.show()
```