CENTRE FOR HIGH PERFORMANCE COMPUTING

www.chpc.ac.za

High Performance Computing with Python and mpi4py

Kevin Colville

science & technology

Department:
Science and Technology
REPUBLIC OF SOUTH AFRICA

CSIR

our future through science

High Performance Computing

HPC is:

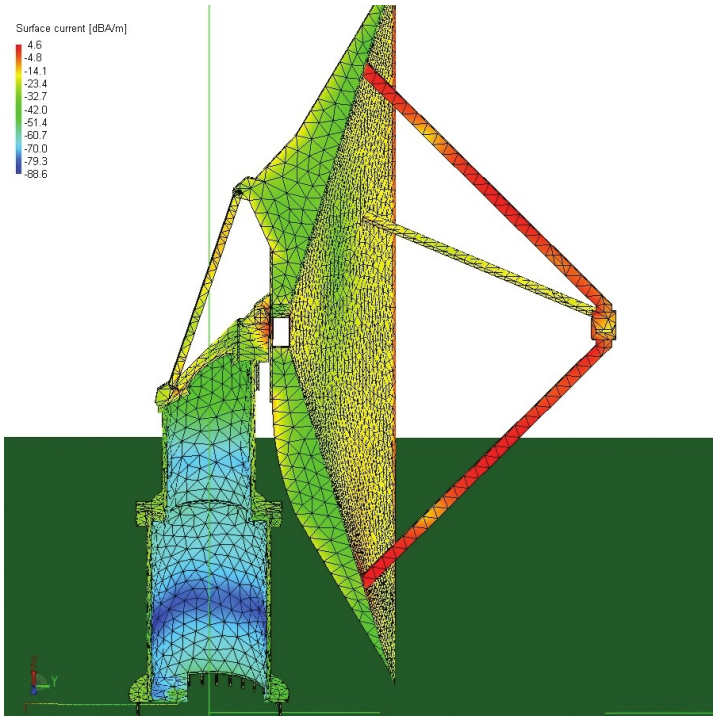
- use of parallel processing
- for running advanced application programs
- efficiently, reliably and quickly

HPC systems function above 1 teraflops:

> 10^{12} floating-point operations per second

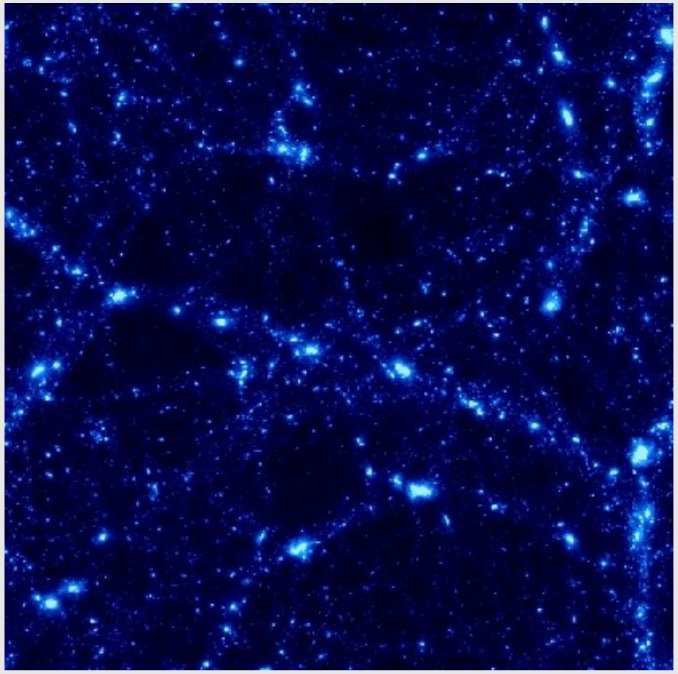
up to 16 petaflops (10^{15})

MeerKAT RFI simulation



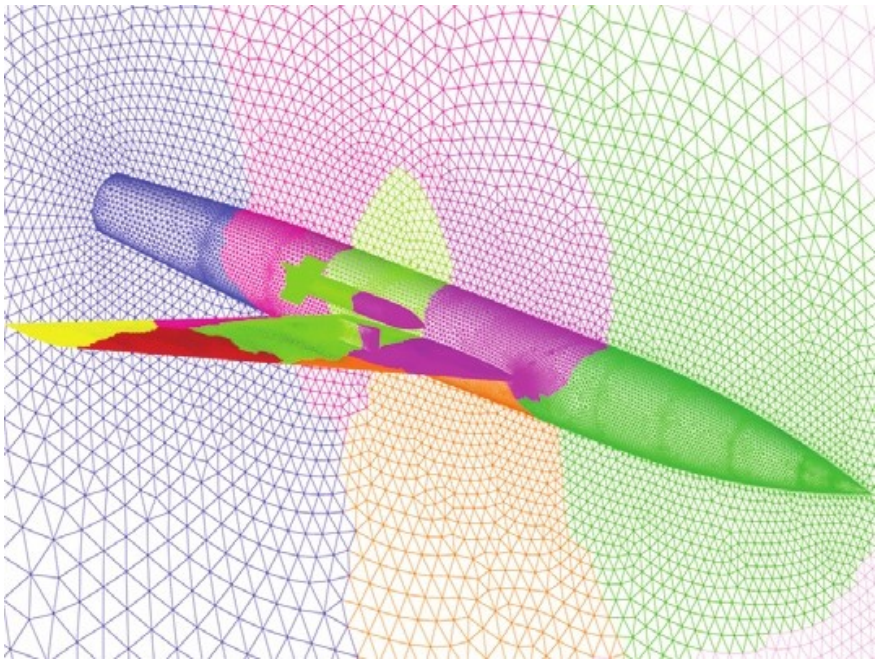
CHPC

Big Science

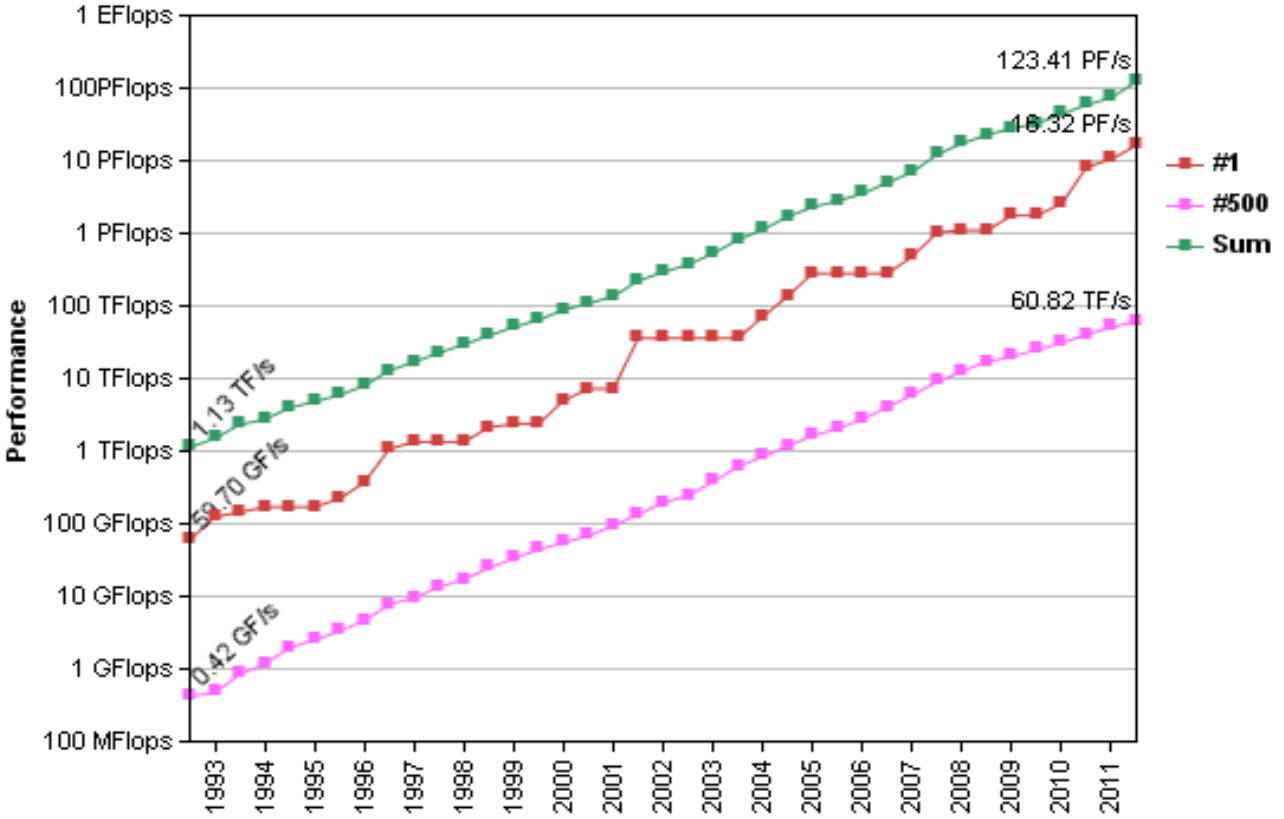


Cosmology

Computational Fluid Dynamics



Top 500 supercomputers



top500.org

Cluster Supercomputer

CHPC Sun Constellation

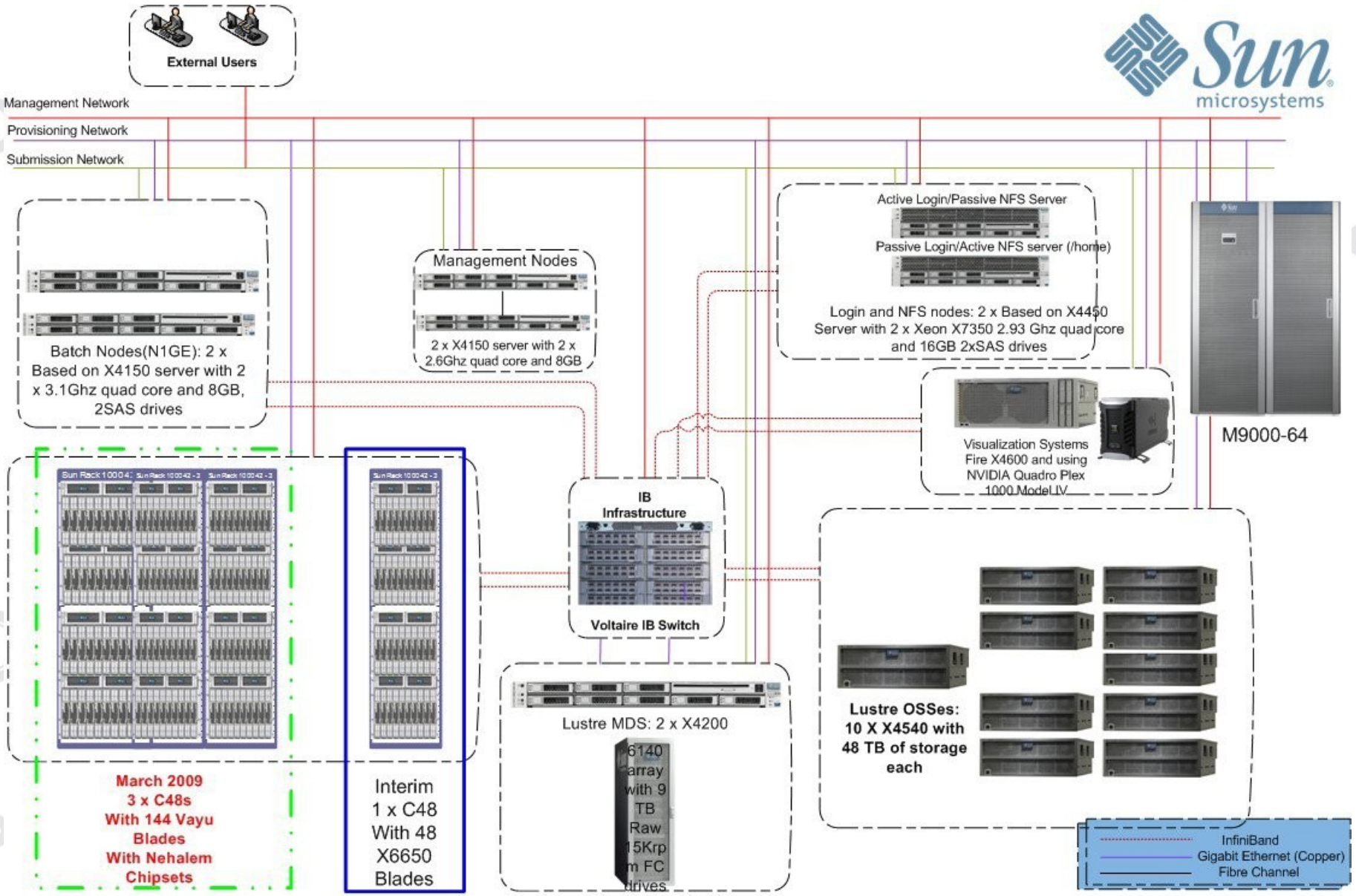
2 304 Intel Nehalem cores

3 456 GB RAM

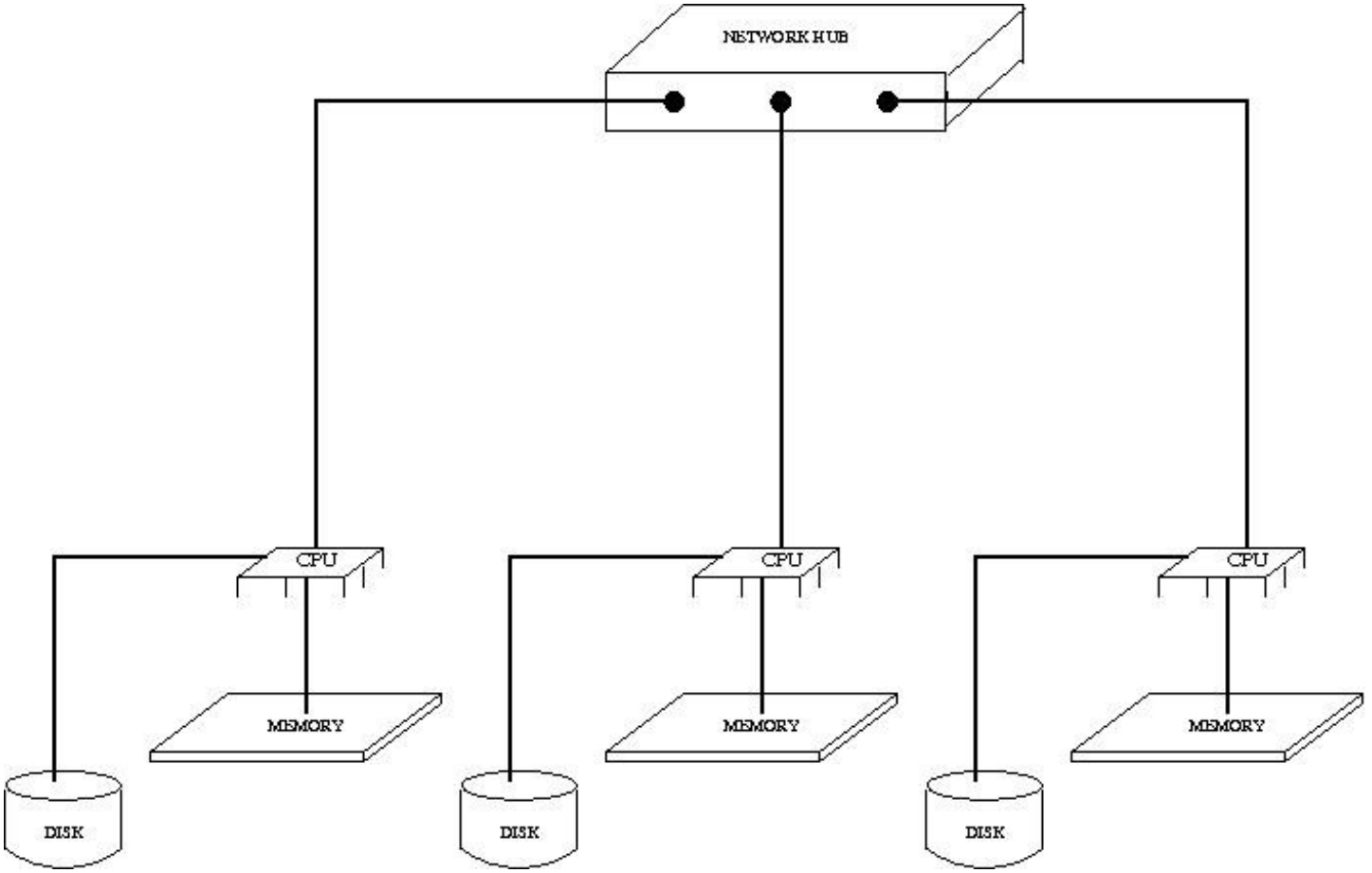
QDR Infiniband

24 Tflops

Sun Constellation cluster


External Users
Management Network
Provisioning Network
Submission Network
Batch Nodes(N1GE): 2 x Based on X4150 server with 2 x 3.1Ghz quad core and 8GB, 2SAS drives
Management Nodes
2 x X4150 server with 2 x 2.6Ghz quad core and 8GB
Active Login/Passive NFS Server
Passive Login/Active NFS server (/home)
Login and NFS nodes: 2 x Based on X4450 Server with 2 x Xeon X7350 2.93 Ghz quad core and 16GB 2xSAS drives
M9000-64
Visualization Systems Fire X4600 and using NVIDIA Quadro Plex 1000 Model IV
IB Infrastructure
Voltaire IB Switch
March 2009
3 x C48s
With 144 Vayu Blades
With Nehalem Chipsets
Interim
1 x C48
With 48
X6650
Blades
Lustre MDS: 2 x X4200
6140 array with 9 TB Raw 15Krpm FC drives
Lustre OSSes: 10 X X4540 with 48 TB of storage each
InfiniBand
Gigabit Ethernet (Copper)
Fibre Channel

Distributed Memory



commons.wikimedia.org

CHPC

Message Passing Interface

- MPI's prime goals are:
  - Provide source-code portability.
  - Allow efficient implementation.
- MPI also offers:
  - A great deal of functionality.
  - Support for heterogeneous parallel architectures
  - C/C++ and Fortran APIs
- Python: mpi4py
  - mpi4py.scipy.org

MPI Communications

- Point to point:
  - involves a sender and a receiver
  - only two processes participate
- Collective communication:
  - all processors within a communicator participate
  - barrier, reduction operations, gather, scatter…

Minimal MPI

```
import mpi4py.MPI as MPI
# MPI.Init()
MPI.COMM_WORLD.Get_size()
MPI.COMM_WORLD.Get_rank()
MPI.COMM_WORLD.Send()
MPI.COMM_WORLD.Recv()
MPI.Finalize()
```

MPIHelloWorld.py

```
import mpi4py.MPI as MPI

# my_rank = rank of process
# np      = number of processes

my_rank = MPI.COMM_WORLD.Get_rank()

np = MPI.COMM_WORLD.Get_size()

print "Hello, world!  I am rank %d of %d processes" % (my_rank, np)

# MPI.Finalize()
```

```
$ mpirun -np 4 python MPIHelloWorld.py
Hello, world!  I am rank 3 of 4 processes
Hello, world!  I am rank 0 of 4 processes
Hello, world!  I am rank 2 of 4 processes
Hello, world!  I am rank 1 of 4 processes
$
```

MPIHelloEveryone.py

```
import numpy
import mpi4py.MPI as MPI

# source  = rank of sender
# dest    = rank of receiver
tag = 0
message = numpy.zeros(100, dtype='c')
status = MPI.Status()

my_rank = MPI.COMM_WORLD.Get_rank()
p = MPI.COMM_WORLD.Get_size()

if (my_rank != 0):
    s = "Greetings from process %d!" % my_rank
    message[:len(s)] = s
    dest = 0
    MPI.COMM_WORLD.Send( [message, len(s)+1, MPI.CHAR], dest, tag )
else:
    for source in range(1,p):
        MPI.COMM_WORLD.Recv( [message, 100, MPI.CHAR], source, tag, status )
        print "%s\n" % message

# MPI.Finalize()
```

Blocking point-to-point

Using numpy arrays — *fast*

- MPI_Send

```
communicator.Send(...)
```

- MPI_Recv

```
communicator.Recv(...)
```

Any python object — *uses pickle*

```
communicator.send(...)
communicator.recv(...)
```

Non-blocking point-to-point

- MPI_Isend

```
request = communicator.Isend(...)
```

- MPI_Irecv

```
request = communicator.Irecv(...)
```

— *fast versions using numpy arrays*

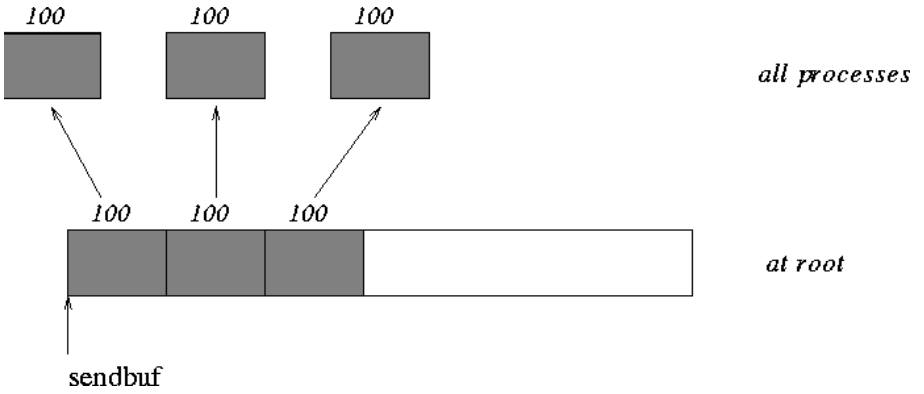
- Check status of Request object:
  - `Test()`, `Wait()`, and `Cancel()` methods

Usable combinations

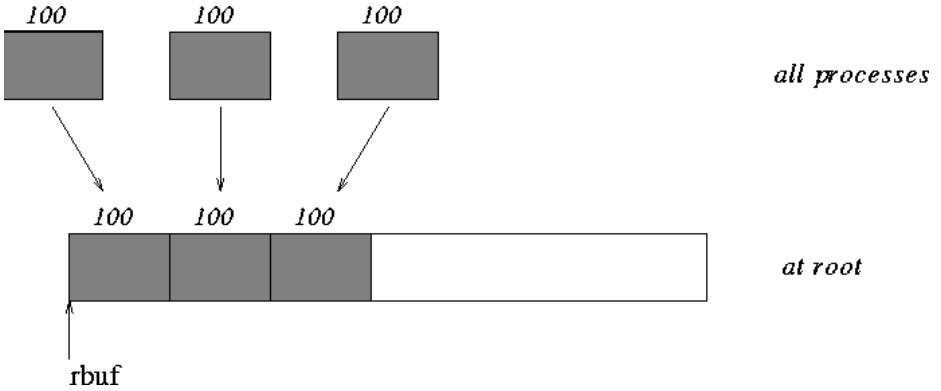
MPI_Send	→	MPI_Recv
MPI_Send	→	MPI_Irecv
MPI_Isend	→	MPI_Recv
MPI_Isend	→	MPI_Irecv
MPI_Sendrecv	↔	MPI_Sendrecv
MPI_Alltoall	↔	MPI_Alltoall

Collective communication

- MPI_Scatter



- MPI_Gather



mpi-forum.org

Scatter

```
from mpi4py import MPI

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

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
```

Gather

```
from mpi4py import MPI

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

data = (rank+1)**2
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
```

Collective communication

- MPI_Reduce
  - MPI_MAX
  - MPI_MIN
  - MPI_MAXLOC
  - MPI_MINLOC
  - MPI_SUM
  - MPI_PROD
  - MPI_LAND
  - MPI_LOR
  - MPI_LXOR
  - MPI_BAND
  - MPI_BOR
  - MPI_BXOR

MPIDotProduct.py

```
import numpy
import mpi4py.MPI as MPI

def Serial_dot(x,y,n):
    sum = 0.0
    for i in xrange(0,n):
        sum = sum + x[i]*y[i]
    return sum

vec1 = numpy.ones(100, 'd')
vec2 = numpy.ones(100, 'd')

my_rank = MPI.COMM_WORLD.Get_rank()
p = MPI.COMM_WORLD.Get_size()
n_bar = int(len(vec1)/p)

my_start = my_rank*n_bar
my_end = (my_rank+1)*n_bar
local_x = vec1[my_start:my_end]
local_y = vec2[my_start:my_end]
dot = 0.0
local_dot = Serial_dot(local_x, local_y, n_bar)
dot = MPI.COMM_WORLD.Reduce(local_dot, None, MPI.SUM, 0)

if (my_rank == 0):
    print "Dot Product completed: product = %f" % dot
```

Dynamic Processes

```
from mpi4py import MPI
import numpy
import sys

comm = MPI.COMM_SELF.Spawn(sys.executable,
                           args=['cpi.py'],
                           maxprocs=3)

N = numpy.array(100, 'i')
comm.Bcast([N, MPI.INT], root=MPI.ROOT)

PI = numpy.array(0.0, 'd')
comm.Reduce(None, [PI, MPI.DOUBLE], op=MPI.SUM,
            root=MPI.ROOT)
print(PI)

comm.Disconnect()
```

cpi.py

```
#!/usr/bin/env python
from mpi4py import MPI
import numpy

comm = MPI.Comm.Get_parent()
size = comm.Get_size()
rank = comm.Get_rank()

N = numpy.array(0, dtype='i')
comm.Bcast([N, MPI.INT], root=0)

h = 1.0 / N;    s = 0.0
for i in range(rank, N, size):
    x = h * (i + 0.5)
    s += 4.0 / (1.0 + x**2)

PI = numpy.array(s * h, dtype='d')
comm.Reduce([PI, MPI.DOUBLE], None, op=MPI.SUM, root=0)

comm.Disconnect()
```

Thank You

Part 2 follows . . .