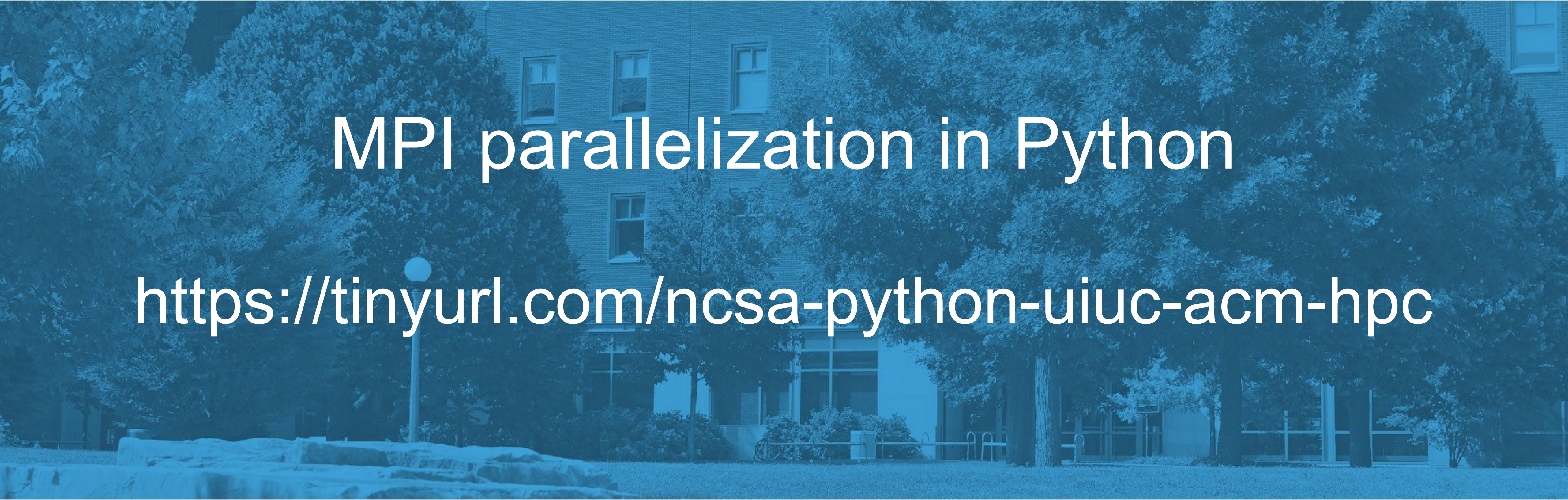


MPI parallelization in Python

https://tinyurl.com/ncsa-python-uiuc-acm-hpc

Roland Haas (NCSA / University of Illinois)
Email: rhaas@illinois.edu

ILLINOIS
NCSA | National Center for
Supercomputing Applications

Following along

- all slides and examples available on GitHub:

https://tinyurl.com/ncsa-python-uiuc-acm-hpc

- login in to `kingfisher`

```
ssh -l <NCSA-USER> kingfisher.ncsa.illinois.edu
```

- copy my prepared Python3 virtualenv

```
cp -a /home/rhaas/mpi_course $HOME/
```

- activate virtualenv

```
cd mpi_course
source bin/activate
```

- Debian / Ubuntu packages (if using your own laptop):

```
apt-get install python3-h5py-mpi python3-mpi4py
```

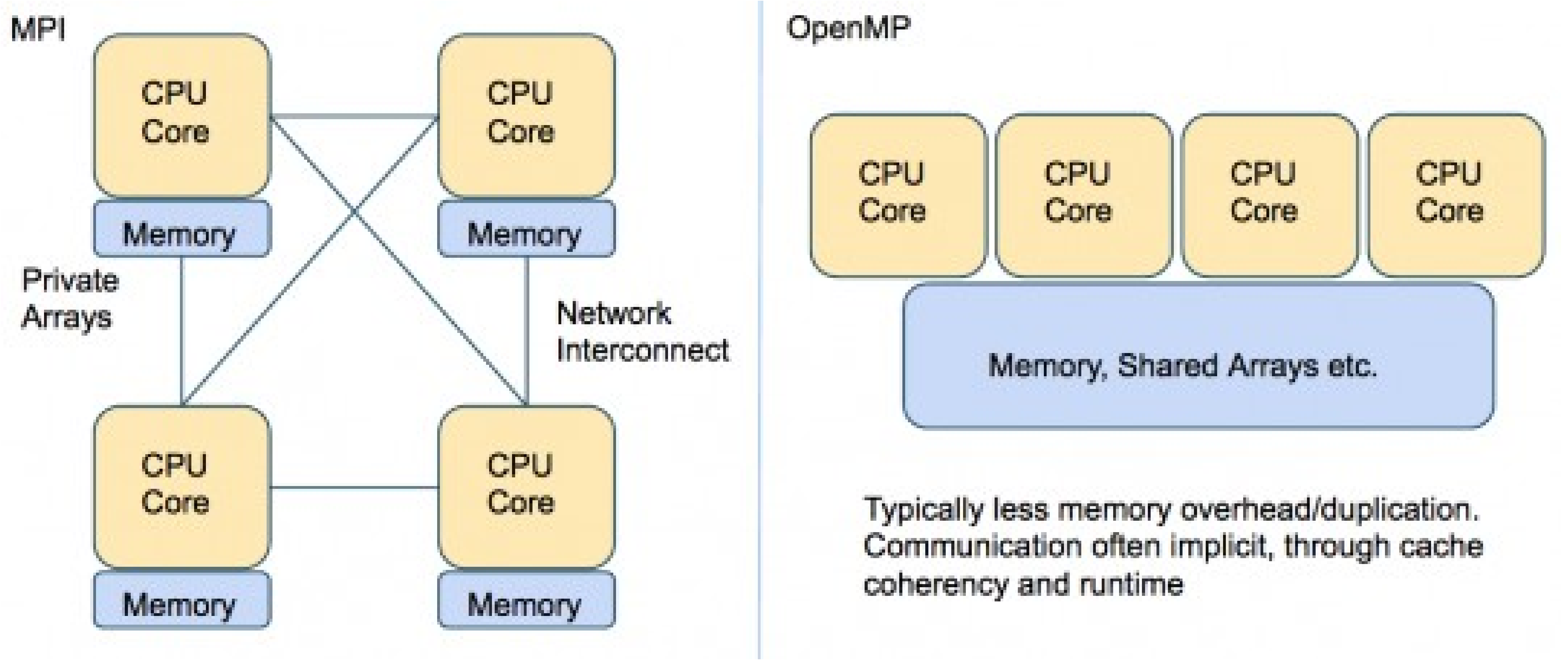
What is MPI?

- MPI is the de-facto standard for inter-node communication on distributed memory systems
- uses explicit function calls and manual parallelization
- has been around for a long time (20+ years), will be around for a while longer
- updated regularly to address new hardware developments
- comprehensive design
  - point-to-point communication
  - collective communication
  - remote-DMA
  - (parallel) IO
  - memory management
  - process management
- available for all major programming languages
  - C / C++
  - Fortran
  - **Python**, Java, R, ...

C vs. python code

- MPI was originally designed with compiled languages (Fortran 77, C) in mind
  - MPI is (mostly) object oriented, (almost) all identifiers are for opaque objects, but uses a function based interface
  - all MPI routines, preprocessor constants and data types start with `MPI_` to form a namespace
  - all functions operate on arrays, taking an array pointer, an array length and a symbolic constant describing the array datatype as arguments
- mpi4py provides pythonic interface to MPI
  - fully object oriented
  - deduce array type and length
  - provide defaults for parameters
- use python to prototype algorithm, C for production
- mpi4py provides SWIG bindings to interact with C code
  - pass communicators between languages
  - can mix C / python code in communicatioin
- multiple books cover C / Fortran API (Amazon finds 595 books for MPI)
- standards documents (complex) https://www.mpi-forum.org/docs/
- MPI API reference https://www.open-mpi.org/doc/v4.0/
- mpi4py docs at readthedocs https://mpi4py.readthedocs.io/en/stable/

Distributed vs. shared memory



Typically less memory overhead/duplication. Communication often implicit, through cache coherency and runtime

http://www.nersc.gov/assets/Uploads/_resampled/ResizedImage540228-MPIVSOPENMP.png

Hello, MPI!

C code

```c
#include <mpi.h>
int main(int argc, char **argv){
 int rank;
 MPI_Init(&argc, &argv);
 MPI_Comm_rank(MPI_COMM_WORLD,
               &rank);
 printf("Hello from %d\n", rank);
 MPI_Finalize();
}
```

- MPI_Init, MPI_Finalize should be first and last lines in code
- (almost) all MPI functions return a status code `ierr` which is typically ignored
- MPI is **defined** in its **standard**, not via a canonical implementation

Python code

```python
import mpi4py.MPI as MPI
rank = MPI.COMM_WORLD.Get_rank()
print ("Hello from %d" % rank)
```

- all processes execute the same binary and code
- processes are assigned a unique `rank` unique among all nodes in a job
- `mpi.h` provides C prototypes
- `import mpi4py` for python

Key concepts

- MPI uses communicators to group ranks
  - `MPI.COMM_WORLD`
  - new subgroups can be created
- calls to MPI routines must match in all involved MPI ranks
  - deadlocks (may) occur if violated
  - this does include the data type and length of arrays used
- messages from the **same** rank are received in the order sent
  - between ranks, explicit synchronization is required
- messages can be tagged by a (small) integer and cherry-picked

- MPI operates on data, not byte streams
- MPI calls come in two flavors
  - collective calls
  - point-to-point communication
  - remote-memory access
- one-to-many and many-to-one communications designate a root rank
- MPI calls do not wait for the receiver to confirm receipt
  - Send calls return to sender when last bit of data is handed over to the network
  - use `comm.Barrier` to synchronize ranks

Assigning work when all are equal...

- all ranks execute the very same code
- unless explicitly request, no synchronization takes place
- they are distinguishable **only** by their `rank` number

```
if (my_pe_num == 0):
  Routine_SpaceInvaders()
else if (my_pe_num == 1):
  Routine_CrackPasswords()
else:
  Routine_WeatherForecast()
```

- more commonly use rank to distribute spatial domain
- or have one controller (rank 0) assign work to multiple workers
- MPI even offers help to choose work based on the physical location of the rank in the network
- for many problems load-balancing becomes an issue

```
Comm.Get_rank()

Comm.Get_size()

Comm.Barrier()
```

Passing the buck – how to serialize code

```
import mpi4py.MPI as MPI

rank = MPI.COMM_WORLD.Get_rank()
sz = MPI.COMM_WORLD.Get_size()

for i in range(sz):
  if(i == rank):
    print ('Hello from %d' % rank)
  MPI.COMM_WORLD.Barrier()
```

- serializing code is useful to
  - disentangle output to screen or file
  - reduce load on IO system when reading or writing data
  - manipulating global state (typically files)
  - reduce required buffers for many-to-one communication
- code shown is not very scalable
  - size^2 communications
  - run time increases linearly with size
- can be sped up by passing a “token” (or “baton” or the “buck”) to your neighbour once done
  - only size communications
  - same runtime

Running the serialization example

```
cd examples
mpirun -n 1 python3 ./mpihello.py
mpirun -n 12 python3 ./mpihello.py
mpirun -n 1 python3 ./serialmpi.py
mpirun -n 12 python3 ./serialmpi.py
```

- randomness of output is more obvious if one runs more MPI ranks than there are CPUs
  - need to allow OpenMPI to do so (some clusters may never let you)

MPI_Bcast, MPI_Reduce

- `Comm.Bcast` broadcasts data to all ranks
- `Comm.Reduce` combines data from all ranks to one rank
  - `MPI.SUM, MPI.PROD`
  - `MPI.MIN, MPI.MAX`
  - `MPI.LAND, MPI.LOR, MPI.LXOR`
- `Comm.Allreduce` bcast's result to all ranks
- same MPI function for all data types
  - Python deduces type and length automatically

- Reductions of arrays are done **horizontally** (element-by-element)

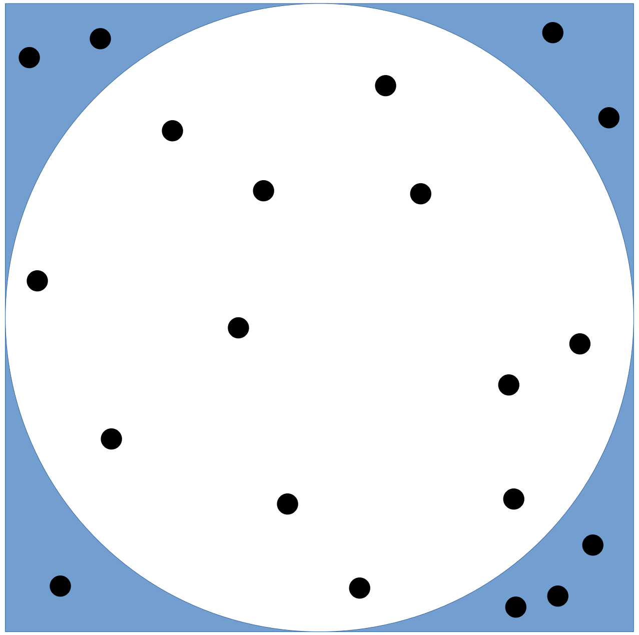
```
Comm.Bcast(buf, root = 0)
```

```
Comm.Reduce(sendbuf, recvbuf,
 op = MPI.SUM, root = 0)
```

Computing π

- use Monte-Carlo integration to compute π from the area of a disk

$$A = \pi r^2$$



```
# random seeds for MC integration
my_seed = np.empty(shape=1, \
                   dtype='i')
if (rank == 0):
 my_seed[:] = seed
MPI.COMM_WORLD.Bcast(my_seed, 0)
random.seed(int(my_seed[0]) + rank)
…
# combine all rank results
global_inside = \
  np.empty_like(local_inside)
MPI.COMM_WORLD.Reduce(local_inside, \
  global_inside, MPI.SUM, 0)
my_pi = (4.0*global_inside) / \
        (sz * local_points)

if (rank == 0):
  print("pi is %g" % my_pi);
```

Computing π output

```
cd examples

mpirun -n 1 python3 ./pi.py

pi is approximated as 3.14177
real pi is 3.14159 diff -0.000179569
Took 5427.55 ms

mpirun -n 3 python3 ./pi.py

pi is approximated as 3.14122
real pi is 3.14159 diff 0.000374876
Took 3036.15 ms
```

Point-to-point communication

- MPI provides `MPI_Send` and `MPI_Recv` as communication primitives

```
numbertosend = np.array([4.0])
numbertoreceive = np.empty_like(numbertosend)
status = MPI.Status()

if (rank == 0):
  MPI.COMM_WORLD.Recv(numbertoreceive, status=status)

if (rank == 1):
  MPI.COMM_WORLD.Send(numbertosend, 0)
```

- can be used to build arbitrary communication patterns

```
Comm.Send(buf, dest, tag = 0)

Comm.Recv(buf, source = MPI.ANY_SOURCE, tag = MPI.ANY_TAG,
          status = None)

Status.Get_source()
Status.Get_count()
Status.Get_tag()
```

Manager - worker schemes using MPI

- one rank, typically rank 0, is designated the manager and hands out work orders to the workers
- workers process each order and return result to master
- manager aggregates the results

```
cd examples

mpirun -n 3 python3 ./worker.py

PE 1 received 0.294665 and
computed 0.294665

Received 0.294665 from PE 1

PE 2 received 0.530587 and
computed 1.061174

Received 1.061174 from PE 2

Total is 1.355839
```

- manager

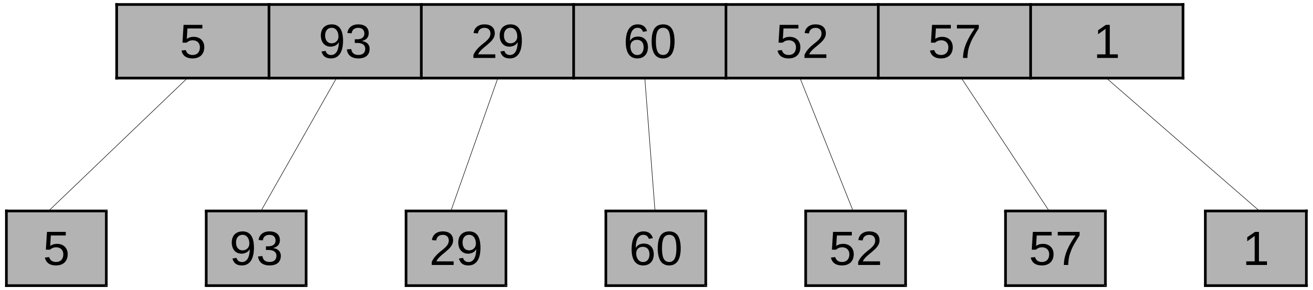
```
for i in range(1,sz):
  comm.Send( \
    numberstosend[i-1], I)

result = np.array([0.])
for i in range(1,sz):
  numbertoreceive = \
    np.empty_like(result)
  comm.Recv(numbertoreceive)
  result += numbertoreceive
```

- worker

```
numbertoreceive = np.empty( \
  shape=(1), dtype=float)
comm.Recv(numbertoreceive, 0)

result = numbertoreceive * rank

comm.Send(result, 0)
```

Scatter and gather data

- `Comm.Scatter` and `Comm.Gather` are similar to `comm.Bcast` and `Comm.Reduce`
- scatter or gather the components of an array among the MPI ranks
- `Comm.Allgather` broadcasts the resulting array to all ranks (like `Comm.Allreduce`)
- useful if one rank prepares work assignments then scatters them to the others



5	93	29	60	52	57	1

5

93

29

60

52

57

1

```
Comm.Scatter(sendbuf, recvbuf,
 root = 0)

Comm.Gather(sendbuf, recvbuf,
 root = 0)

Comm.Allgather(sendbuf, recvbuf)
```

MPI_Gather, MPI_Scatter examples

```
myvals = numpy.empty(2)

if(rank == 0):
  values = numpy.arange(2.*sz)
  comm.Scatter(values, myvals, \
    root=0)
else:
  comm.Scatter(None, myvals, \
    root=0)

myvals = myvals **2

if(rank == 0):
  values = numpy.empty(2*sz)
  comm.Gather(myvals, values, root=0)
  print("Got squares: "+str(values))
else:
  comm.Gather(myvals, None, root=0)
```

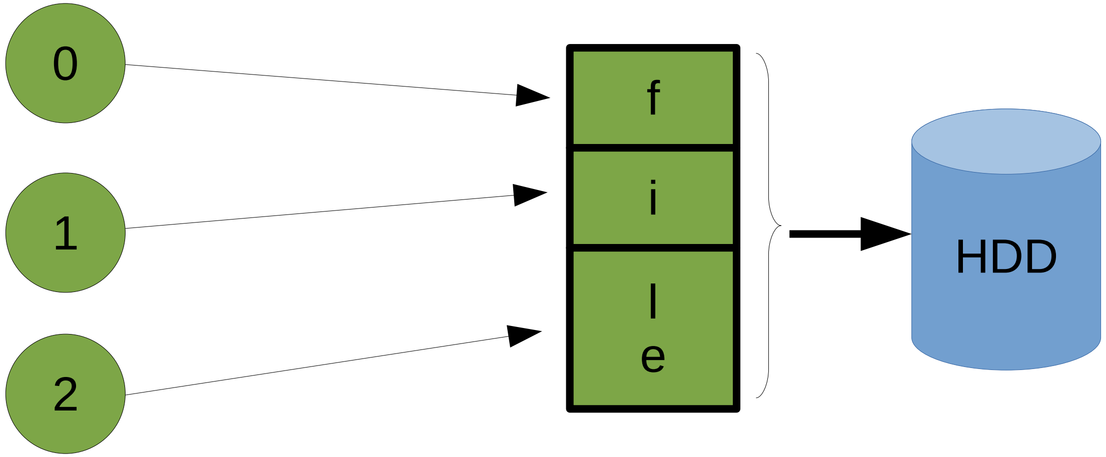
- large arrays only exist on root rank
  - length of array must be multiple of `sz`
  - use `Comm.Scatterv` otherwise (complicated)
- and similar for gather...

```
cd examples

mpirun -n 1 python3 ./gather.py
Got squares: [0. 1.]

mpirun -n 4 python3 ./gather.py
Got squares: [ 0.  1.  4.  9.
16. 25. 36. 49.]
```

MPI-IO using h5py

- HDF5 is standardized, binary, self-describing file format to store numerical data
  - essentially a file system in a file
  - automatic endianess adjustment
  - type conversion `float` <> `double` etc.
  - very nice Python interface in `h5py`
- MPI-IO provides functions to
  - write to the same file from multiple ranks in parallel
  - optimize reads / writes for speed and reduces impact on the file system (not in `h5py`)
  - asynchronous I/O
- HDF5 provides a convenient interface to MPI-IO

- collective operations (all ranks must participate):
  - opening HDF5 files for writing
  - creating data sets
- independent operations (each rank does its own thing):
  - opening HDF5 files for reading
  - reading data from data sets
  - writing data to data sets



MPI-IO examples

```
fh = h5py.File('data.h5', 'w', \
       driver='mpio', comm=comm)

dset = \
 fh.create_dataset('alldata', \
   shape=(sz+1, 3), dtype=float)

if rank == sz-1:
 dset[rank:rank+2] = \
  10.*rank + np.arange(0.,6.).\
                reshape((2,3))
else:
 dset[rank] = \
  10.*rank + np.arange(0.,3.).\
                reshape((1,3))


h5py.File(name, mode,
 driver='mpiio', comm=Comm)
```

```
cd examples

mpirun -n 1 python3 ./mpiio.py
alldata:
 [[0. 1. 2.]
 [3. 4. 5.]]

mpirun -n 4 python3 ./mpio.py
alldata:
 [[ 0.  1.  2.]
 [10. 11. 12.]
 [20. 21. 22.]
 [30. 31. 32.]
 [33. 34. 35.]]
```

Why not use MPI?

- You will likely have to rewrite portions in all areas of your code.
  - Old, dusty subroutines written by a long-departed grad student.
- You will have to understand almost all of your code.
  - Old, dusty subroutines written by a long-departed grad student.
- You can’t do it incrementally.
  - Major data structures have to be decomposed up front.
- Debugging will be “different".
  - You aren’t just finding the bad line of code. You sometimes need to find the bad PE.

taken from https://www.psc.edu/hpc-workshop-series/mpi

Further reading

- more MPI functions
  - non-blocking communication
  - multiple communicators
  - MPI shared memory
  - custom data types
- multiple books cover C / Fortran API (Amazon finds 595 books for MPI)
- standards documents (complex) https://www.mpi-forum.org/docs/
- MPI API reference https://www.open-mpi.org/doc/v4.0/
- mpi4py docs at readthedocs https://mpi4py.readthedocs.io/en/stable/
- h5py docs https://docs.h5py.org/en/stable/
- MPI course https://www.hpc-training.org/xsede/moodle/enrol/index.php?id=34
- MPI for grid based algorithms, e.g. the Laplace equation

Question?

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

ILLINOIS

NCSA | National Center for
Supercomputing Applications

OpenMP task constructs

- `omp task` spins off a new task
  - variables are `private` by default
  - use `firstprivate` to pass arguments, `shared` causes in a race condition
  - use `shared` to return values
- `omp single` to spin off recursion
- `omp taskwait` waits for immediate child tasks to finish
- `omp critical` with a name if shared variables are modified
- `omp taskgroup` if result of tasks is not required
- many more clauses and constructs
  - `taskyield`
  - `final, untied, mergable, depend`

- Compiling the example code

```
cd examples/
make fib
```

- Run the example code

```
export OMP_NUM_THREADS=1
./fib
```

```
fib(30)=832040
Took 248.556 ms
```

```
export OMP_NUM_THREADS=2
./fib
```

```
fib(30)=832040
Took 1036.49 ms
```

- this is indeed slower in parallel

Compiling and running

```
cd examples
mpicc -o mpihello mpihello.c

mpirun -n 1 ./mpihello

mpirun -n 4 ./mpihello

mpirun -n 1 python mpihello.py

mpirun -n 4 python mpihello.py
```

- compiler wrappers take care of linking against MPI runtime library
  - mpicc, mpicxx, mpif90 (usually)
  - mpiicc, mpiicpc, mpiifort (Intel compiler)
  - cc, CC, ftn (Cray machines...)

- executable must be run via mpirun
  - some MPI stacks (OpenMPI) let you run a serial version without using mpirun
  - some cluster won't let you run on the login nodes
- each MPI rank corresponds to a UNIX process
- when one of the ranks terminates, (usually) mpirun will terminate the whole run
- console output from all ranks is collected by mpirun
- rank 0 receives mpirun's input stream