

```
1]: #need to start outside of the main directory
```

```
2]: cd mpi4py
```

```
/home/tkaiser/tut/mpi4py
```

```
3]: %load_ext slurm_magic
```

```
4]: # sprint: Use like print but returns a string instead of printing.
# clist: Returns output from the magic command %%capture as a list
# tymer: a useful tymer
from tymer import *
```

```
5]: ls *.py
```

```
future.py*  P_ex00.py*  P_ex04.py*  P_ex10.py*  q4.py*      stp.py*
h5.py*      P_ex01b.py* P_ex05.py*  P_ex12.py*  queue.py*   surface.py*
lustre.py   P_ex01.py*  P_ex06.py*  P_ex13.py*  report.py*  write_grid.py*
mkview.py*  P_ex02.py*  P_ex07.py*  ppong.py*   runMPI.py
mpi1.py*    P_ex03I.py* P_ex08.py*  primes.py   simple.py*
pcalc.py*   P_ex03.py*  P_ex09.py*  pwrite.py*  stp_00.py*
```

```
24]: srun -n 4 ./P_ex00.py
```

```
24]: 'Hello from 1 on blk Numprocs is 4\npython is not about snakes\nHello from
3 on blk Numprocs is 4\npython is not about snakes\nHello from 0 on blk
Numprocs is 4\npython is not about snakes\nHello from 2 on blk Numprocs is
4\npython is not about snakes\n'
```

```
25]: for x in _.split("\n") :
      print(x)
```

```
Hello from 1 on blk Numprocs is 4
python is not about snakes
Hello from 3 on blk Numprocs is 4
python is not about snakes
Hello from 0 on blk Numprocs is 4
python is not about snakes
Hello from 2 on blk Numprocs is 4
python is not about snakes
```

```
In [19]: %%capture out
srun -n 4 ./P_ex00.py
```

```
In [20]: out=clist(out)
for x in out:
    print(x)
```

```
Hello from 1 on blk Num
python is not about snakes
Hello from 3 on blk Num
python is not about snakes
n
Hello from 0 on blk Num
python is not about snakes
Hello from 2 on blk Numprocs is 4
python is not about snakes
```

```
#Run from within save
```

```
%load_ext slurm_magic
```

```
from slideshow import mysay
from tymer import tymer
from tymer import clist
from tymer import sprint
```

```
mpi4py=mysay(datadir="mpi01_py")
#for rmpi02_py skip over list of examples nx(slide=18)
#mpi4py=mysay(datadir="mpi02_py")
#mpi4py=mysay(datadir="bagoftasks_py")
#mpi4py=mysay(datadir="stoma_py")
#mpi4py=mysay(datadir="stomb_py")
```

```
mpi4py.nx()
```

```
%macro n 4
```

```
nx=mpi4py.nx
```

```
cd ../..
```

```
!srun -n 4 ./P_ex00.py
```

```
# In[9]:
```

```
srun -n 8 hostname
```

```
# In[10]:
```

```
!srun -n 8 hostname
```

```
# In[11]:
```

```
tymer(["-i","start"])
```

```
# In[12]:
```

```
%%capture out
srun -n 8 ./report.py
```

```
# In[13]:
```

```
tymer(["-i","done"])
```

```
# In[14]:
```

```
for x in clist(out) :
    print(x)
```

```
nx(slide=1)
```

notes for getting things going

# Parallel Programming

## Basic MPI

### With MPI4py

Timothy H. Kaiser, Ph.D.  
[tkaiser2@nrel.gov](mailto:tkaiser2@nrel.gov)



The background of the slide is a scenic photograph of a mountain range. In the foreground, there is a dense forest of green coniferous trees. Behind the trees, a series of rugged, rocky mountain peaks are visible, some with patches of snow or light-colored rock. The sky is a pale, hazy blue. The text is overlaid on the center of the image.

**Slides at:**  
**<https://github.com/timkphd/slides>**



# Talk Overview

- Background on MPI
- Documentation
- Hello world in MPI
- Some differences between mpi4py and normal MPI
- Basic communications
- Simple send and receive program
- How you build and run



# Examples at

```
git clone https://github.com/timkphd/examples  
cp -r examples/mpi/mpi4py .  
cd mpi4py
```

Goal for today: quickly go over things but leave you with many well commented examples and at some useful full example programs.



# Running Today

- We are going to run most examples today in a Jupyter notebook
- It is useful to install the "slurm" magics
- See: <https://github.com/NERSC/slurm-magic>
  - "This package implements **magic commands** for interacting with the **SLURM** workload manager. SLURM magic simply wraps command-line executables and the commands themselves should look like their command-line counterparts. "
  - It's easy to extend or modify for your purposes
- `%load_ext slurm_magic`



# Examples run on:

- CU/CSU Summit
- NREL Eagle
- Mac laptop and desktop
- But today...



# My Raspberry Pi Cluster

- Currently two 4 core nodes
  - clr & blk
- Ethernet interconnect
- lmod
- gcc 10.x
- OpenMPI & MPICH
- R 4.0.5
- Python 3.9.4
  - Local version of slurm magics
  - Local version of srun
- spack





# Background on MPI

- MPI - Message Passing Interface
  - Library standard defined by a committee of vendors, implementers, & parallel programmers
  - Used to create parallel programs based on message passing
- Portable: one standard, many implementations
- Available on almost all parallel machines in C and Fortran
- Python is not one for the officially supported languages
- Over 100 advanced routines but 6 basic



# Documentation

- MPI home page (contains the library standard):  
[www.mcs.anl.gov/mpi](http://www.mcs.anl.gov/mpi)
- Books
  - "MPI: The Complete Reference" by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press (also in Postscript and html)
  - "Using MPI" by Gropp, Lusk and Skjellum, MIT Press
- Tutorials
  - many online, just do a search



# MPI Implementations

- Most parallel supercomputer vendors provide optimized implementations
- Some vendor versions
  - Intel
  - Portland Group
  - mpt
- OpenMPI



# MPI Implementations

- MPICH:
  - <http://www-unix.mcs.anl.gov/mpi/mpich1/download.html>
  - <http://www.mcs.anl.gov/research/projects/mpich2/index.php>
- MVAPICH & MVAPICH2
  - Infiniband optimized version of MPICH
  - <http://mvapich.cse.ohio-state.edu/index.shtml>



# mpi4py

- A python module
- One of a number of python wrappers of MPI, calls C MPI underneath
- Main Page:
  - <http://mpi4py.scipy.org>
- Start of documentation:
  - <https://mpi4py.readthedocs.io/en/stable/>



# Key Concepts of MPI

- Used to create parallel programs based on message passing
- Normally the same program is running on several different processors
- Processors communicate using message passing
- Typical methodology:

```
start job on n processors
do i=1 to j
    each processor does some calculation
    pass messages between processor
end do
end job
```



# Messages

- Simplest message: an array of data of one type.
- Predefined types correspond to commonly used types in a given language
  - MPI\_REAL (Fortran), MPI\_FLOAT (C)
  - MPI\_DOUBLE\_PRECISION (Fortran), MPI\_DOUBLE (C)
  - MPI\_INTEGER (Fortran), MPI\_INT
  - mpi4py types match C but adds pickled data for “special” calls
- User can define more complex types and send packages.



# Communicators

- Communicator
  - A collection of processors working on some part of a parallel job
  - Used as a parameter for most MPI calls
  - For mpi4py the communicator is an object with methods
  - MPI\_COMM\_WORLD includes all of the processors in your job
  - Processors within a communicator are assigned numbers (ranks) 0 to n-1



# Include files

- The MPI include file
  - C: `mpi.h`
  - Fortran: `mpif.h` (a `f90` module is a good place for this)
  - `mpi4py: import mpi4py`
- Defines many constants used within MPI programs
- In C defines the interfaces for the functions
- Compilers know where to find the include files



# mpi4py differences

- The communicator is not an argument to calls but an object with methods
- `MPI_COMM_WORLD` is created when you import the module
- Instead of a function call:

```
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

- we have:

```
comm=MPI.COMM_WORLD  
myid=comm.Get_rank()
```



# mpi4py differences

- If a parameter to a call can be determined from context it is optional
- If there is a reasonable default value for a parameter (e.g. root for reduction) it is optional



# mpi4py differences and observations

- Two versions of most communication calls
- Upper case
  - More like regular MPI routines
  - Data transported are numpy arrays
  - Can enclose (optional) descriptions in square brackets
- Lower case
  - Automatically pickle and unpickle data
  - Can send Python objects
  - Transported values are return values of method calls
  - Can optionally have received values as a parameter



# Python is **!!!!MUCH!!!!** slower

- Our “big” example calculation, `scp.py/stc_03.c`
  - Python 4720 seconds
  - C/Fortran 3 seconds
- You can however, call C and Fortran compiled subroutines from Python
- You can mix Python and C/Fortran in MPMD fashion
  - MPI tasks 0 to n-2 might be Fortran or C
  - MPI task n-1 could be a Python graphics program
    - Have example
    - `mpiexec -n 3 ./ccalc : -n 1 pwrite.py < small.in`
  - Probably would not work with pickled data



# Minimal MPI program

- Every MPI program needs these...

- C version

```
/* the mpi include file */
#include <mpi.h>
int nPES, ierr, iam;
/* Initialize MPI */
ierr=MPI_Init(&argc, &argv);
/* How many processors (nPES) are there? */
ierr=MPI_Comm_size(MPI_COMM_WORLD, &nPES);
/* What processor am I (what is my rank)? */
ierr=MPI_Comm_rank(MPI_COMM_WORLD, &iam);
...
ierr=MPI_Finalize();
```

In C MPI routines are functions and return an error value



# Minimal MPI program

- Every MPI program needs these...

- Fortran version

```
! MPI include file
  include 'mpif.h'
! The mpi module can be used for Fortran 90 instead of mpif.h
!   use mpi
  integer nPEs, ierr, iam
! Initialize MPI
  call MPI_Init(ierr)
! How many processors (nPEs) are there?
  call MPI_Comm_size(MPI_COMM_WORLD, nPEs, ierr)
! What processor am I (what is my rank)?
  call MPI_Comm_rank(MPI_COMM_WORLD, iam, ierr)
  ...
  call MPI_Finalize(ierr)
```

In Fortran, MPI routines are subroutines, and  
last parameter is an error value



# Minimal MPI program

- Every MPI program needs these...
- mpi4py version

```
#!/usr/bin/env python
# numpy is required
import numpy
from numpy import *

# mpi4py module
from mpi4py import MPI

# Initialize MPI
comm=MPI.COMM_WORLD

# What processor am I (what is my rank)?
myid=comm.Get_rank()

# How many processors (nPEs) are there?
numprocs=comm.Get_size()

print("Hello from ",myid," Numprocs is ",numprocs)

print("python is not about snakes")

# Shut down MPI
MPI.Finalize()
```

## P\_ex00.py

Hello world

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- MPI.Finalize()



# Basic Communication

- Data values are transferred from one processor to another
  - One processor sends the data
  - Another receives the data
- Synchronous
  - Call does not return until the message is sent or received
- Asynchronous
  - Call indicates a start of send or receive, and another call is made to determine if finished



# Synchronous Send

- C
  - `MPI_Send(&buffer, count ,datatype, destination, tag,communicator);`
- Fortran
  - Call `MPI_Send(buffer, count, datatype, destination,tag,communicator, ierr)`
- mpi4py
  - `Send(self, buf, int dest, int tag=0)`
- Call blocks until message on the way



**Call MPI\_Send(buffer, count, datatype,  
destination, tag, communicator, ierr)**

- **Buffer**: The data array to be sent
- **Count** : Length of data array (in elements, 1 for scalars)
- **Datatype** : Type of data, for example : MPI\_DOUBLE\_PRECISION, MPI\_INT, etc
- **Destination** : Destination processor number (within given communicator)
- **Tag** : Message type (arbitrary integer)
- **Communicator** : Your set of processors
- **Ierr** : Error return (Fortran only)



# Synchronous Receive

- C
  - `MPI_Recv(&buffer, count, datatype, source, tag, communicator, &status);`
- Fortran
  - Call `MPI_RECV(buffer, count, datatype, source, tag, communicator, status, ierr)`
- mpi4py
  - `Recv(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG, Status status=None)`
- Call blocks the program until message is in buffer
- Status - contains information about incoming message
  - C - `MPI_Status status;`
  - Fortran - `Integer status(MPI_STATUS_SIZE)`
  - mpi4py - object



**Call MPI\_Recv(buffer, count, datatype,  
source, tag, communicator,  
status, ierr)**

- **Buffer**: The data array to be received
- **Count** : Maximum length of data array  
(in elements, 1 for scalars)
- **Datatype** : Type of data, for example :  
MPI\_DOUBLE\_PRECISION, MPI\_INT, etc
- **Source** : Source processor number  
(within given communicator)
- **Tag** : Message type (arbitrary integer)
- **Communicator** : Your set of processors
- **Status**: Information about message
- **Ierr** : Error return (Fortran only)



# Exercise 2 : Basic Send and Receive

```
#!/usr/bin/env python
# numpy is required
import numpy
from numpy import *

# mpi4py module
from mpi4py import MPI

# Initialize MPI and print out hello
comm=MPI.COMM_WORLD
myid=comm.Get_rank()
numprocs=comm.Get_size()
print("hello from ",myid," of ",numprocs)

# Tag identifies a message
mytag=1234

# Process 0 is going to send the data
mysource=0
```

[P\\_ex01.py](#) , [f\\_ex01.f90](#)

Blocking Send and Receive

- MPI.COMM\_WORLD
- Get\_rank()
- Get\_size()
- comm.Send()
- comm.Recv()
- MPI.Finalize

```
# Process 1 is going to send the data
mydestination=1

# Sending a single value each time
count=1
for k in range(1,4):
    if myid == mysource:
# For the upper case calls we need to send/recv numpy arrays
        buffer=array(k+5678,"i")
# We are sending a integer, size is optional, to mydestination
        comm.Send([buffer, MPI.INT], dest=mydestination, tag=mytag)
        print("Python processor ",myid," sent ",buffer)

    if myid == mydestination:
# We are receiving an integer, size is optional, from mysource
        if(k == 1) : buffer=empty((1),"i")
        comm.Recv([buffer, MPI.INT], source=mysource, tag=mytag)
        print("Python processor ",myid," got ",buffer)

MPI.Finalize()
```



# Exercise 2 : Basic Send and Receive

```
module fmpi
  include "mpif.h"
end module

!*****
! This is a simple send/receive program in MPI
! Processor 0 sends an integer to processor 1,
! while processor 1 receives the integer from proc. 0
!*****

program hello
  use fmpi
!  include "mpif.h"
  integer myid, ierr,numprocs
  integer tag,source,destination,count
  integer buffer
  integer status(MPI_STATUS_SIZE)
  call MPI_INIT( ierr )
  call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
  call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
  tag=1234
  source=0
  destination=1
  count=1
  do i =1,3
    if(myid .eq. source)then
      buffer=5678-i
      Call MPI_Send(buffer, count, MPI_INTEGER,destination,&
        tag, MPI_COMM_WORLD, ierr)
      write(*,*)"Fortran processor ",myid," sent ",buffer
    endif
    if(myid .eq. destination)then
      Call MPI_Recv(buffer, count, MPI_INTEGER,source,&
        tag, MPI_COMM_WORLD, status,ierr)
      write(*,*)"Fortran processor ",myid," got ",buffer
    endif
  enddo
  call MPI_FINALIZE(ierr)
end
```



# Summary

- MPI is used to create parallel programs based on message passing
- Usually the same program is run on multiple processors
- The 6 basic calls in MPI are:
  - `INIT()` “not required”
  - `comm=MPI.COMM_WORLD`
  - `comm.Get_rank()`
  - `comm.Get_size()`
  - `comm.Send(buf, dest, tag=0)`
  - `comm.Recv(buf, source=ANY_SOURCE, tag=ANY_TAG, Status status=None)`
  - `MPI.Finalize()`



# Compiling

- Most everywhere:
  - mpif77 mpif90
  - mpicc mpiCC
- Use Intel compilers
  - mpiicc mpiifort
- Most MPI compilers are actually just scripts that call underlying Fortran or C compilers
- module load "something"
  - gives access to the compilers



# mpi4py

- Compiling not required
- Need to have MPI on your system
- `pip install mpi4py`
- You might want to install you own copy of python
- Or use conda to create one
- Heresy:
  - I recommend using conda to create your environment
  - **Then** use `pip install mpi4py`



# Running

- Most often you will use a batch system
- Write a batch script file.
- You must tell the system how many copies to run
- On some systems you must tell where to run the program
- The command `mpiexec` or `mpirun` is used to start the program
- On many systems use **srun** instead of `mpiexec` or `mpirun`



# Running

- `sbatch my_script`
  - Submits the script to the scheduler
- `queue -u $USER`
  - Shows what jobs you have running
- `scancel #####`
  - kills job #####
- `scancel -u $USER`
  - kills all of your jobs



# A very simple Slurm Script

```
#!/bin/bash -x
#SBATCH --job-name="hybrid"
#comment = "glorified hello world"
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=8
#SBATCH --ntasks=16
#SBATCH --exclusive
#SBATCH --export=ALL
#SBATCH --time=10:00:00

# Go to the directory from which our job was launched
cd $SLURM_SUBMIT_DIR

# run an application
srun $SLURM_SUBMIT_DIR/hello.c

# You can also use the following format to set
# --nodes          - # of nodes to use
# --ntasks-per-node - ntasks = nodes*ntasks-per-node
# --ntasks         - total number of MPI tasks
#srun --nodes=$NODES --ntasks=$TASKS --ntasks-per-node=$TPN $EXE > output.$SLURM_JOBID
```



# A mpi4py test script

```
#!/bin/bash
#SBATCH --job-name="hybrid"
#comment = "script for mpi4py tests"
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --exclusive
#SBATCH --export=ALL
#SBATCH --time=10:00:00

# These module commands are machine specific
export MODULEPATH=/sw/mfiles:$MODULEPATH
module purge
module load Compiler/intel/18.0
module load MPI/impi/2018.1/intel
module load Compiler/python/2/2.7/comercial/intel/2018_1

# Go to the directoy from which our job was launched
cd $SLURM_SUBMIT_DIR

# These are the examples that do not require stdin
EXES="P_ex00.py P_ex01b.py P_ex01.py P_ex02.py
P_ex03I.py P_ex03.py P_ex04.py P_ex05.py
P_ex06.py P_ex07.py P_ex08.py P_ex09.py
P_ex10.py P_ex12.py P_ex13.py"

for X in $EXES ; do
    OUT=`echo $X | sed "s/py/out/"`
    echo running $X
    srun ./$X > $OUT
done
```