MPI
More of the Story
Timothy H. Kaiser, PH.D.
tkaiser@mines.edu
Examples at

http://hpc.mines.edu/examples

To just get these examples:

mkdir examples

cd examples
curl http://hpc.mines.edu/examples/examples/mpi/mpi4py/mpi4py.tgz | tar -xz

Goal for today: quickly go over things but leave you with many well commented examples and at least one useful full example program.
Outline

- Review
- Types
- Broadcast
- Wildcards
- Using Status and Probing
- Asynchronous Communication
- More Global communications
- Advanced topics
  - "V" operations
  - Communicators
Outline: Advance examples

- Finite difference code
- Mixing mpi4py and C or Fortran
- Bag of tasks
- Passing a token
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()
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

# 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()
```

P_ex01.py, f_ex01.f90

Blocking Send and Receive
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.Send()
- comm.Recv()
- MPI.Finalize

P_ex01b.py

Blocking Send and Receive Character Data
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.Send()
- comm.Recv()
- MPI.Finalize
Our Examples

**P_ex00.py**
Hello world
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- MPI.Finalize()

**P_ex01.py**, **f_ex01.f90**
Blocking Send and Receive
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.Send()
- comm.Recv()
- MPI.Finalize

**P_ex01b.py**
Blocking Send and Receive *Character Data*
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.Send()
- comm.Recv()
- MPI.Finalize

**P_ex02.py**
Blocking Send and Receive with probe to find size
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.Send()
- comm.probe()
- mystat.Get_count()
- comm.Recv()
- MPI.Finalize

**P_ex03.py**
Nonblocking Send and Receive with wait
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.isend()
- comm.irecv()
- req.wait()
- MPI.Finalize

http://hpc.mines.edu/examples/examples/mpi/mpi4py/index.html
Our Examples

**P_ex03I.py**
Nonblocking Send and Receive with wait
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.Isend()
- comm.Irecv()
- req.wait()
- MPI.Finalize

**P_ex04.py**
Broadcast of an array of integers and a string
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.bcast()
- comm.Bcast()
- MPI.Finalize

**P_ex05.py**
This program shows how to use MPI_Scatter and MPI_Gather. Each processor gets different data from the root processor by way of mpi_scatter. The data is summed and then sent back to the root processor using MPI_Gather. The root processor then prints the global sum.
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.bcast()
- comm.Scatter()
- comm.gather()
- comm.Gather()
- MPI.Finalize
Our Examples

P_ex06.py
This program shows how to use MPI_Scatter and MPI_Reduce. Each processor gets different data from the root processor by way of mpi_scatter. The data is summed and then sent back to the root processor using MPI_Reduce. The root processor then prints the global sum.

• MPI.COMM_WORLD
• Get_rank()
• Get_size()
• comm.bcast()
• comm.Scatter()
• comm.reduce()
• comm.Reduce()
• MPI.Finalize

P_ex07.py
This program shows how to use MPI_Alltoall. Each processor send/rec a different random number to/from other processors.

• MPI.COMM_WORLD
• Get_rank()
• Get_size()
• comm.Alltoall()
• MPI.Finalize
Our Examples

**P_ex08.py**
This program shows how to use MPI_Gatherv. Each processor sends a different amount of data to the root processor. We use MPI_Gather first to tell the root how much data is going to be sent.

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.gather
- comm.Gatherv
- MPI.Finalize

**P_ex09.py**
This program shows how to use Alltoallv. Each processor gets amounts of data from each other processor. It is an extension to example P_ex07.py. In mpi4py the displacement array can be calculated automatically from the rcounts array. We show how it would be done in "normal" MPI. See also P_ex08.py for how this can be done

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.Alltoall()
- comm.Alltoallv()
- MPI.Finalize
Our Examples

P_ex10.py, ex10.in

Pass a "token" from one task to the next with a single task reading token from a file.

An extensive program. We first create a new communicator that contains every task except the zeroth one. This is done by first defining a group, new_group. We define new_group based on the group associated with mpi_comm_world, old_group and an array, will_use. Will_use contains a list of tasks to be included in the new group. It does not contain the zeroth one. The new communicator is sub_comm_world.

There are other ways to create communicator but this is one of the more general methods.

Next, we have break of the task not in the communicator to call the routine get_input. This routine will do input from a file ex10.in. The file contains a list of integers. The task will send the integer to the first task in the new communicator.

The remaining tasks which are port of sub_comm_world call the routine pass_token. Pass_token "just" receives a value from the previous processor and passes it on to the next.

There is a minor subtlety in pass_token. We are using both our new communicator and MPI_COMM_WORLD. The tasks that are port of the new communicator use it to pass data. We note that the task that is injecting values into the stream is not part of the new communicator so it must use MPI_COMM_WORLD. Thus we do a probe on WORLD, which is actually MPI_COMM_WORLD looking for a message. When we get it we send it on using the new communicator.

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- WORLD.Iprobe()
- comm.Get_group()
- old_group.Incl()
- comm.Create()
- new_group.Get_rank()
- MPI.Finalize
Our Examples

P_ex12.py
This program shows how to use mpi_comm_split
Split will create a set of communicators. All of the tasks with the same value of color will be in the same communicator. In this case we get two sets one for odd tasks and one for even tasks. Note they have the same name on all tasks, new_comm, but there are actually two different values (sets of tasks) in each one.

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.Split
- comm.bcast
- MPI.Finalize

P_ex13.py
This program shows how to use Scatterv. Each processor gets a different amount of data from the root processor. We use MPI_Gather first to tell the root how much data is going to be sent.

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.gather
- comm.Scatterv
- MPI.Finalize
Our Examples

`simple.py`, `flist`

**This is a bag-of-tasks program.** We define a manager task that distributes work to workers. Actually, the workers request input data. The manager sits in a loop calling Iprobe waiting for requests for work. In this case the manager reads input. The input is a list of file names. It will send an entry from the list as requested. When the worker is done processing it will request a new file name from the manager. This continues until the manager runs out of files to process. The manager subroutine is just "manager". The worker subroutine is "worker". It receives file names from the manager. The files in this case are outputs from an optics program tracking a laser beam as it propagates through the atmosphere. The workers read in the data and then create an image of the data by calling the routine mkview.plotit. This should work with arbitrary 2d files except the size in mkview.plotit is currently hard coded to 64 x 64.

We use the call to "Split" to create a separate communicator for the workers. This is not important in this example but could be if you wanted multiple workers to work together.

To get the data...

```
curl http://hpc.mines.edu/examples/laser.tgz | tar -xz
```

- `MPI.COMM_WORLD`
- `Get_rank()`
- `Get_size()`
- `comm.gather`
- `comm.Send()`
- `comm.Recv()`
- `MPI.Status()`
- `comm.Iprobe()`
- `gotfrom=status.source`
- `MPI.Get_processor_name()`
- `MPI.COMM_WORLD.barrier()`
- `MPI.Finalize`
Our Examples

<table>
<thead>
<tr>
<th>File</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ccalc.c</td>
<td>parallel</td>
</tr>
<tr>
<td>stc_03.c</td>
<td>parallel</td>
</tr>
<tr>
<td>pcalc.py</td>
<td>parallel</td>
</tr>
<tr>
<td>stp_00.py</td>
<td>serial</td>
</tr>
<tr>
<td>stp.py</td>
<td>parallel</td>
</tr>
<tr>
<td>tiny.in</td>
<td>tiny input file</td>
</tr>
<tr>
<td>small.in</td>
<td>small input file</td>
</tr>
<tr>
<td>st.in</td>
<td>regular input file</td>
</tr>
</tbody>
</table>

We have a finite difference model that will serve to demonstrate what a computational scientist needs to do to take advantage of Distributed Memory computers using MPI.

The model we are using is a two dimensional solution to a model problem for Ocean Circulation, the Stommel Model. It has Wind-driven circulation in a homogeneous rectangular ocean under the influence of surface winds, linearized bottom friction, flat bottom and Coriolis force. Solution: intense crowding of streamlines towards the western boundary caused by the variation of the Coriolis parameter with latitude.

For a description of the Fortran and C versions of this program see:

http://geco.mines.edu/prototype/Show_me_some_local_HPC_tutorials/stoma.pdf
http://geco.mines.edu/prototype/Show_me_some_local_HPC_tutorials/stomb.pdf

The python version, stp.py, follows this C version except it does a 1d decomposition.

The python version, stp.py, follows this C version except it does a 1d decomposition.

The C version is 1500x faster than the python version.

pcalc.py and ccalc.c are similar except they create a new communicator that contains N-1 tasks. These tasks do the calculation and pass data to the remaining task to be plotted. Thus we can have "C" do the heavy calculation and python do plotting.
We have a finite difference model that will serve to demonstrate what a computational scientist needs to do to take advantage of Distributed Memory computers using MPI.

The model we are using is a two dimensional solution to a model problem for Ocean Circulation, the Stommel Model. It has Wind-driven circulation in a homogeneous rectangular ocean under the influence of surface winds, linearized bottom friction, flat bottom and Coriolis force. Solution: intense crowding of streamlines towards the western boundary caused by the variation of the Coriolis parameter with latitude.

For a description of the Fortran and C versions of this program see:

http://geco.mines.edu/prototype/Show_me_some_local_HPC_tutorials/stoma.pdf
http://geco.mines.edu/prototype/Show_me_some_local_HPC_tutorials/stomb.pdf

The python version, stp.py, follows this C version except it does a 1d decomposition.

The C version is 1500x faster than the python version.

pcalc.py and ccalc.c are similar except they create a new communicator that contains N-1 tasks. These tasks do the calculation and pass data to the remaining task to be plotted. Thus we can have "C" do the heavy calculation and python do plotting.
Our Examples

**pwrite.py**
pwrite.py is a small MPI program designed to be run in conjunction with either ccalc.c or pcalc.py. If you are using mpiexec to launch your programs these might be launched together using one of the commands:

```
mpiexec -n 5 ./ccalc : -n 1 ./pwrite.py < cut.in
mpiexec -n 5 ./pcalc.py : -n 1 ./pwrite.py < cut.in
```

pcalc.py and ccalc.c are versions of the finite difference program discussed above. pcalc.py and ccalc.c create a new communicator that contains N-1 tasks. These tasks do the calculation and pass data to the remaining task to be plotted. The remaining task is pwrite.py.

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- world.Get_group()
- old_group.Incl()
- world.Create()
- world.barrier()
- MPI.Finalize
Our Examples

**write_grid.py**
write_grid.py contains three procedures write_each, write_one, write_extra, plot_extra. These are different output routines for the finite difference code discussed above.

**write_each**
Each MPI task writes its portion of the grid in a separate file. Could be called from stp.py

**write_one**
Each MPI task sends its portion of the grid to a single task and it is written as a single file. Could be called from stp.py

**write_extra**
This could be called from the "extra" MPI task pwrite.py. This routine collects the data from all other tasks and prints it.

**plot_extra**
This could be called from the "extra" MPI task pwrite.py. This routine collects the data from all other tasks and plots it using mkview.py

Write_one, write_extra, and plot_extra all work the same way. They collect data to a single task a line at a time using a combination of Gather and Gatherv. For a give line, each processor tells the writing processor how much, if any of the line it holds using the Gather. The the Gatherv is used to actually transfer the data. For write_one and write_extra each line is printed as it is gathered. The routine plot_extra collects the whole grid before plotting it. Write_each opens a file with the name based on the task id. Each task writes its portion of the grid to its file.

- comm.Get_rank()
- comm.Get_size()
- comm.Gather()
- comm.Gatherv()
MPI Types

- MPI has many different predefined data types
- Can be used in any communication operation
## Predefined types in C

<table>
<thead>
<tr>
<th>C MPI Types</th>
<th>C Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_CHAR</code></td>
<td>signed char</td>
</tr>
<tr>
<td><code>MPI_SHORT</code></td>
<td>signed short int</td>
</tr>
<tr>
<td><code>MPI_INT</code></td>
<td>signed int</td>
</tr>
<tr>
<td><code>MPI_LONG</code></td>
<td>signed long int</td>
</tr>
<tr>
<td><code>MPI_UNSIGNED_CHAR</code></td>
<td>unsigned char</td>
</tr>
<tr>
<td><code>MPI_UNSIGNED_SHORT</code></td>
<td>unsigned short int</td>
</tr>
<tr>
<td><code>MPI_UNSIGNED</code></td>
<td>unsigned int</td>
</tr>
<tr>
<td><code>MPI_UNSIGNED_LONG</code></td>
<td>unsigned long int</td>
</tr>
<tr>
<td><code>MPI_FLOAT</code></td>
<td>float</td>
</tr>
<tr>
<td><code>MPI_DOUBLE</code></td>
<td>double</td>
</tr>
<tr>
<td><code>MPI_LONG_DOUBLE</code></td>
<td>long double</td>
</tr>
<tr>
<td><code>MPI_BYTE</code></td>
<td>-</td>
</tr>
<tr>
<td><code>MPI_PACKED</code></td>
<td>-</td>
</tr>
</tbody>
</table>
## Predefined types in Fortran

<table>
<thead>
<tr>
<th>Fortran MPI Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>-</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>-</td>
</tr>
<tr>
<td>Predefined types in mpi4py (91)</td>
<td></td>
</tr>
<tr>
<td>--------------------------------</td>
<td></td>
</tr>
<tr>
<td>AINT</td>
<td></td>
</tr>
<tr>
<td>BOOL</td>
<td></td>
</tr>
<tr>
<td>BYTE</td>
<td></td>
</tr>
<tr>
<td>CHAR</td>
<td></td>
</tr>
<tr>
<td>CHARACTER</td>
<td></td>
</tr>
<tr>
<td>COMPLEX</td>
<td></td>
</tr>
<tr>
<td>COMPLEX16</td>
<td></td>
</tr>
<tr>
<td>COMPLEX32</td>
<td></td>
</tr>
<tr>
<td>COMPLEX4</td>
<td></td>
</tr>
<tr>
<td>COMPLEX8</td>
<td></td>
</tr>
<tr>
<td>COUNT</td>
<td></td>
</tr>
<tr>
<td>CXX_BOOL</td>
<td></td>
</tr>
<tr>
<td>CXX_DOUBLE_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>CXX_FLOAT_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>CXX_LONG_DOUBLE_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>C_BOOL</td>
<td></td>
</tr>
<tr>
<td>C_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>C_DOUBLE_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>C_FLOAT_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>C_LONG_DOUBLE_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>DATATYPE_NULL</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_INT</td>
<td></td>
</tr>
<tr>
<td>DOUBLE_PRECISION</td>
<td></td>
</tr>
<tr>
<td>FLOAT</td>
<td></td>
</tr>
<tr>
<td>FLOAT_INT</td>
<td></td>
</tr>
<tr>
<td>F_BOOL</td>
<td></td>
</tr>
<tr>
<td>F_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>F_DOUBLE</td>
<td></td>
</tr>
<tr>
<td>F_DOUBLE_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>F_FLOAT</td>
<td></td>
</tr>
<tr>
<td>F_FLOAT_COMPLEX</td>
<td></td>
</tr>
<tr>
<td>F_INT</td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td></td>
</tr>
<tr>
<td>INT16_T</td>
<td></td>
</tr>
<tr>
<td>INT32_T</td>
<td></td>
</tr>
<tr>
<td>INT64_T</td>
<td></td>
</tr>
<tr>
<td>INT8_T</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
</tr>
<tr>
<td>INTEGER1</td>
<td></td>
</tr>
<tr>
<td>INTEGER16</td>
<td></td>
</tr>
<tr>
<td>INTEGER2</td>
<td></td>
</tr>
<tr>
<td>INTEGER4</td>
<td></td>
</tr>
<tr>
<td>INTEGER8</td>
<td></td>
</tr>
<tr>
<td>INT_INT</td>
<td></td>
</tr>
<tr>
<td>LB</td>
<td></td>
</tr>
<tr>
<td>LOGICAL</td>
<td></td>
</tr>
<tr>
<td>LOGICAL1</td>
<td></td>
</tr>
<tr>
<td>LOGICAL2</td>
<td></td>
</tr>
<tr>
<td>LOGICAL4</td>
<td></td>
</tr>
<tr>
<td>LOGICAL8</td>
<td></td>
</tr>
<tr>
<td>LONG</td>
<td></td>
</tr>
<tr>
<td>LONG_DOUBLE</td>
<td></td>
</tr>
<tr>
<td>LONG_DOUBLE_INT</td>
<td></td>
</tr>
<tr>
<td>LONG_INT</td>
<td></td>
</tr>
<tr>
<td>LONG_LONG</td>
<td></td>
</tr>
<tr>
<td>OFFSET</td>
<td></td>
</tr>
<tr>
<td>PACKED</td>
<td></td>
</tr>
<tr>
<td>REAL</td>
<td></td>
</tr>
<tr>
<td>REAL16</td>
<td></td>
</tr>
<tr>
<td>REAL2</td>
<td></td>
</tr>
<tr>
<td>REAL4</td>
<td></td>
</tr>
<tr>
<td>REAL8</td>
<td></td>
</tr>
<tr>
<td>SHORT</td>
<td></td>
</tr>
<tr>
<td>SHORT_INT</td>
<td></td>
</tr>
<tr>
<td>SIGNED_CHAR</td>
<td></td>
</tr>
<tr>
<td>SIGNED_INT</td>
<td></td>
</tr>
<tr>
<td>SIGNED_LONG</td>
<td></td>
</tr>
<tr>
<td>SIGNED_LONG_LONG</td>
<td></td>
</tr>
<tr>
<td>SIGNED_SHORT</td>
<td></td>
</tr>
<tr>
<td>SINT16_T</td>
<td></td>
</tr>
<tr>
<td>SINT32_T</td>
<td></td>
</tr>
<tr>
<td>SINT64_T</td>
<td></td>
</tr>
<tr>
<td>SINT8_T</td>
<td></td>
</tr>
<tr>
<td>TWOINT</td>
<td></td>
</tr>
<tr>
<td>UB</td>
<td></td>
</tr>
<tr>
<td>UINT16_T</td>
<td></td>
</tr>
<tr>
<td>UINT32_T</td>
<td></td>
</tr>
<tr>
<td>UINT64_T</td>
<td></td>
</tr>
<tr>
<td>UINT8_T</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_CHAR</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_INT</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_LONG</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_LONG_LONG</td>
<td></td>
</tr>
<tr>
<td>UNSIGNED_SHORT</td>
<td></td>
</tr>
<tr>
<td>WCHAR</td>
<td></td>
</tr>
<tr>
<td>_typedict</td>
<td></td>
</tr>
<tr>
<td>_typedict_c</td>
<td></td>
</tr>
<tr>
<td>_typedict_f</td>
<td></td>
</tr>
</tbody>
</table>
MPI Broadcast call: MPI_Bcast

- All nodes call MPI_Bcast
- One node (root) sends a message all others receive the message

- C
  
  ```c
  MPI_Bcast(&buffer, count, datatype, root, communicator);
  ```

- Fortran
  
  ```fortran
  call MPI_Bcast(buffer, count, datatype, root, communicator, ierr)
  ```

- Root is node that sends the message
MPI Broadcast call: MPI_Bcast

- All nodes call MPI_Bcast
- One node (root) sends a message all others receive the message

- **Bcast**(self, buf, int root=0)
- **bcast**(self, obj, int root=0)
Broadcast

P_ex04.py

Broadcast of an array of integers and a string

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.bcast()
- comm.Bcast()
- MPI.Finalize
Wildcards

- Allow you to not necessarily specify a tag or source

- Example

```c
MPI_Status status;
int buffer[5];
int error;
error = MPI_Recv(&buffer[0], 5, MPI_INT,
                  MPI_ANY_SOURCE, MPI_ANY_TAG,
                  MPI_COMM_WORLD,&status);
```

- MPI_ANY_SOURCE and MPI_ANY_TAG are wild cards

- Status structure is used to get wildcard values
Wildcards

- Allow you to not necessarily specify a tag or source
- Example

```python
status = MPI.Status()

comm.Recv([i, MPI.INT],
           source=MPI.ANY_SOURCE,
           tag=MPI.ANY_TAG,
           status=mystat)
```

- MPI_ANY_SOURCE and MPI_ANY_TAG are wild cards
- Status object is used to get wildcard values
Status

- The status parameter returns additional information for some MPI routines
  - Additional Error status information
  - Additional information with wildcard parameters
- C declaration: a predefined struct
  - `MPI_Status status;`
- Fortran declaration: an array is used instead
  - `INTEGER STATUS(MPI_STATUS_SIZE)`
- mpi4py: an class object
  - `status=MPI.Status()`
Accessing status information

- The tag of a received message
  - C : status.MPI_TAG
  - Fortran : STATUS(MPI_TAG)
- The source of a received message
  - C : status.MPI_SOURCE
  - Fortran : STATUS(MPI_SOURCE)
- The error code of the MPI call
  - C : status.MPI_ERROR
  - Fortran : STATUS(MPI_ERROR)
- Other uses...
Accessing status information mpi4py

class Status(builtins.object):
    Status

    Methods defined here:

    Get_count(...)
      Status.Get_count(self, Datatype datatype=BYTE)
      Get the number of *top level* elements

    Get_elements(...)
      Status.Get_elements(self, Datatype datatype)
      Get the number of basic elements in a datatype

    Get_error(...)
      Status.Get_error(self)
      Get message error

    Get_source(...)
      Status.Get_source(self)
      Get message source

    Get_tag(...)
      Status.Get_tag(self)
      Get message tag

    Is_cancelled(...)
      Status.Is_cancelled(self
      Test to see if a request was cancelled
MPI_Probe

- MPI_Probe allows incoming messages to be checked without actually receiving.
- The user can then decide how to receive the data.
- Useful when different action needs to be taken depending on the "who, what, and how much" information of the message.
MPI_Probe

- C
  - int MPI_Probe(source, tag, comm, &status)
- Fortran
  - MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)

- Parameters
  - Source: source rank, or MPI_ANY_SOURCE
  - Tag: tag value, or MPI_ANY_TAG
  - Comm: communicator
  - Status: status object
MPI_Probe

- mpi4py

- **Probe**(self, int source=\texttt{ANY\_SOURCE}, int tag=\texttt{ANY\_TAG}, Status status=\texttt{None})

- Parameters
  - Source: source rank, or \texttt{MPI\_ANY\_SOURCE}
  - Tag: tag value, or \texttt{MPI\_ANY\_TAG}
  - Comm: communicator
MPI_Probe example (part 1) f_ex02.f

! How to use probe and get_count
! to find the size of an incoming message
program probe_it
include 'mpif.h'
integer myid,numprocs
integer status(MPI_STATUS_SIZE)
integer mytag,icount,ierr,iray(10)
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
mytag=123; iray=0; icount=0
if(myid .eq. 0)then
! Process 0 sends a message of size 5
  icount=5
  iray(1:icount)=1
  call MPI_SEND(iray,icount,MPI_INTEGER, 1,mytag,MPI_COMM_WORLD,ierr)
endif
MPI_Probe example (part 2)

if(myid .eq. 1)then
  ! process 1 uses probe and get_count to find the size
  call mpi_probe(0,mytag,MPI_COMM_WORLD,status,ierr)
  call mpi_get_count(status,MPI_INTEGER,icount,ierr)
  write(*,*)"getting ", icount," values"
  call mpi_recv(iray,icount,MPI_INTEGER,0, &
                 mytag,MPI_COMM_WORLD,status,ierr)
endif
write(*,*)iray
write(*,*)iray
call mpi_finalize(ierr)
stop
End
MPI_Probe example

**P_ex02.py**

Blocking Send and Receive with probe to find size

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.Send()
- comm.probe()
- mystat.Get_count()
- comm.Recv()
- MPI.Finalize
MPI_BARRIER

- Blocks the caller until all members in the communicator have called it.
- Used as a synchronization tool.

- **C**
  - `MPI_Barrier(comm)`

- **Fortran**
  - `Call MPI_BARRIER(COMM, IERROR)`

- **Parameter**
  - Comm communicator (`MPI_COMM_WORLD`)
MPI_BARRIER

- Blocks the caller until all members in the communicator have called it.
- Used as a synchronization tool.
- mpi4py
  - `Barrier(self)`
Asynchronous Communication

- Asynchronous send: send call returns immediately, send actually occurs later
- Asynchronous receive: receive call returns immediately. When received data is needed, call a wait subroutine
- Asynchronous communication used in attempt to overlap communication with computation (usually doesn’t work)
- Can help prevent deadlock (not advised)
Asynchronous Send with MPI_Isend

- C
  - MPI_Request request
  - int MPI_Isend(&buffer, count, datatype, dest, tag, comm, &request)

- Fortran
  - Integer REQUEST
  - MPI_ISEND(BUFFER,COUNT,DATATYPE, DEST, TAG, COMM, REQUEST, IERROR)

- Request is a new output Parameter
- Don't change data until communication is complete
Asynchronous Send with MPI_Isend

- mpi4py
  - `isend(self, obj, int dest, int tag=0)`
  - `Isend(self, buf, int dest, int tag=0)`

- They return a communication Request which is an object with various methods

- Don't change data until communication is complete
Asynchronous Receive with MPI_Irecv

- **C**
  - `MPI_Request request;`
  - `int MPI_Irecv(&buf, count, datatype, source, tag, comm, &request)`

- **Fortran**
  - `Integer request`
  - `MPI_Irecv(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)`

- **Parameter Changes**
  - Request: communication request
  - Status parameter is missing
  - Don't use data until communication is complete
Asynchronous Receive with MPI_Irecv

- mpi4y
  - `Irecv`(self, buf, int source=ANY_SOURCE, int tag=ANY_TAG)
  - `irecv`(self, buf=None, int source=ANY_SOURCE, int tag=ANY_TAG)

- Parameter Changes
  - They return a communication Request which is an object with various methods
  - Status parameter is missing
  - Don't use data until communication is complete
MPI_Wait used to complete communication

- Request from Isend or Irecv is input
- The completion of a send operation indicates that the sender is now free to update the data in the send buffer
- The completion of a receive operation indicates that the receive buffer contains the received message
- MPI_Wait blocks until message specified by "request" completes
MPI_Wait used to complete communication

- C
  - MPI_Request request;
  - MPI_Status status;
  - MPI_Wait(&request, &status)

- Fortran
  - Integer request
  - Integer status(MPI_STATUS_SIZE)
  - MPI_WAIT(REQUEST, STATUS, IERROR)

- MPI_Wait blocks until message specified by "request" completes
MPI_Wait used to complete communication

- Very different in mpi4py
- Wait is a method of the class object “Request”
- Where req is the Request returned by the Isend/Irecv the call is:
  - Irecv
  - req.wait()
  - req.Wait()
  - irecv()
  - buffer=req.wait()
Asynchronous Send and Receive with MPI_Wait used to complete communication

**P_ex03.py**
Nonblocking Send and Receive with wait
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.isend()
- comm.irecv()
- req.wait()
- MPI.Finalize

**P_ex03I.py**
Nonblocking Send and Receive with wait
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.Isend()
- comm.Irecv()
- req.wait()
- MPI.Finalize
MPI_Test

- Similar to MPI_Wait, but does not block
- Value of flags signifies whether a message has been delivered

- C
  - `int flag`
  - `int MPI_Test(&request,&flag, &status)`

- Fortran
  - `LOGICAL FLAG`
  - `MPI_TEST(REQUEST, FLAG, STATUS, IER)`
Non blocking send example

call MPI_Isend (buffer,count,datatype,dest,
tag,comm, request, ierr)
10 continue

   Do other work ...

call MPI_Test (request, flag, status, ierr)
if (.not. flag) goto 10
MPI_Test

- Very different in mpi4py
  - req.test returns a tuple (flag, buffer), works with both irecv and Irecv
  - req.Test just returns a flag, works only with Irecv

```python
while (not req.Test()) :
    time.sleep(0.5)
    print("dest", req.Test(), req.test()[0])
    print("processor ", destination, " got ", buffer)

while buffer == (False, None) :
    buffer=req.test()
    print("dest", req.Test(), buffer)
    time.sleep(0.5)
    buffer=buffer[1]
    print("processor ", destination, " got ", buffer)

dest False (False, None)
dest False (False, None)
dest False (False, None)
source True (True, None)
dest True (True, None)
processor 1 got [5678]
```

dest False (False, None)
dest False (False, None)
dest False (False, None)
source True (True, None)
processor 0 sent 5678
dest True (True, 5678)
processor 1 got 5678
Scatter Operation using MPI_Scatter

- Similar to Broadcast but sends a section of an array to each processors

Data in an array on root node:

A(0)  A(1)  A(2)  . . .  A(N-1)

Goes to processors:

P₀  P₁  P₂  . . .  Pₙ₋₁
MPI_Scatter

- **C**
  ```c
  int MPI_Scatter(&sendbuf, sendcnts, sendtype, &recvbuf, recvcnts, recvtype, root, comm );
  ```
- **Fortran**
  ```fortran
  MPI_Scatter(sendbuf,sendcnts,sendtype, recvbuf,recvcnts,recvtype,root,comm,ierror)
  ```
- **Parameters**
  - Sendbuf is an array of size (number processors*sendcnts)
  - Sendcnts number of elements sent to each processor
  - Recvcnts number of elements obtained from the root processor
  - Recvbuf elements obtained from the root processor, may be an array
**MPI_Scatter**

- mpi4py
  - `scatter(self, sendobj, int root=0)`
  - `Scatter(self, sendbuf, recvbuf, int root=0)`

**Parameters**

- Sendbuf is an array of size (number processors*sendcnts)
- Sendcnts number of elements sent to each processor (not needed)
- Recvcnts number of elements obtained from the root processor (not needed)
- Recvbuf elements obtained from the root processor, may be an array
Scatter Operation using MPI_Scatter

- Scatter with Sendcnts = 2

Data in an array on root node:

\[
\begin{align*}
\text{P}_0 & \\
A(0) & A(2) & A(4) & \ldots & A(2N-2) \\
A(1) & A(3) & A(5) & \ldots & A(2N-1) \\
\end{align*}
\]  

Goes to processors:

\[
\begin{align*}
\text{P}_0 & \quad \text{P}_1 & \quad \text{P}_2 & \quad \ldots & \quad \text{P}_{n-1} \\
B(0) & B(0) & B(0) & B(0) & B(0) \\
B(1) & B(1) & B(1) & B(1) & B(1) \\
\end{align*}
\]
Gather Operation using MPI_Gather

- Used to collect data from all processors to the root, inverse of scatter
- Data is collected into an array on root processor

Data from various Processors:

\[
\begin{align*}
\text{P}_0 & \quad \text{A} \\
\text{P}_1 & \quad \text{A} \\
\text{P}_2 & \quad \text{A} \\
\vdots & \quad \vdots \\
\text{P}_{n-1} & \quad \text{A}
\end{align*}
\]

Goes to an array on root node:

\[
\begin{align*}
\text{A}(0) & \quad \text{A}(1) \quad \text{A}(2) \quad \ldots \quad \text{A}(N-1)
\end{align*}
\]
MPI_Gather

- C

```c
int MPI_Gather(&sendbuf, sendcnts, sendtype, &recvbuf, recvcnts, recvtype, root, comm);
```

- Fortran

```fortran
MPI_Gather(sendbuf, sendcnts, sendtype, recvbuf, recvcnts, recvtype, root, comm, ierror)
```

- Parameters

  - Sendcnts # of elements sent from each processor
  - Sendbuf is an array of size sendcnts
  - Recvcnts # of elements obtained from each processor
  - Recvbuf of size Recvcnts*number of processors
MPI_Scatter

- **mpi4py**
  
  - `scatter`(self, sendobj, int root=0)
  
  - `Scatter`(self, sendbuf, recvbuf, int root=0)

- **Parameters**
  
  - Sendbuf is an array of size (number processors*sendcnts)
  
  - Sendcnts number of elements sent to each processor (not needed)
  
  - Recvcnts number of elements obtained from the root processor (not needed)
  
  - Recvbuf elements obtained from the root processor, may be an array
This program shows how to use MPI_Scatter and MPI_Gather. Each processor gets different data from the root processor by way of mpi_scatter. The data is summed and then sent back to the root processor using MPI_Gather. The root processor then prints the global sum.

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.bcast()
- comm.Scatter()
- comm.gather()
- comm.Gather()
- MPI.Finalize
Reduction Operations

- Used to combine partial results from all processors
- Result returned to root processor
- Several types of operations available
- Works on single elements and arrays
MPI routine is MPI_Reduce

- **C**
  - `int MPI_Reduce(&sendbuf, &recvbuf, count, datatype, operation, root, communicator)`

- **Fortran**
  - `call MPI_Reduce(sendbuf, recvbuf, count, datatype, operation, root, communicator, ierr)`

- **Parameters**
  - Like MPI_Bcast, a root is specified.
  - Operation is a type of mathematical operation
MPI_Reduce

- **mpi4py**
  - `reduce`(self, sendobj, op=SUM, int root=0)
  - `Reduce`(self, sendbuf, recvbuf, Op op=SUM, int root=0)

- **Parameters**
  - Like MPI_Bcast, a root is specified.
  - Operation is a type of mathematical operation
Operations for MPI_Reduce

- **MPI_MAX**: Maximum
- **MPI_MIN**: Minimum
- **MPI_PROD**: Product
- **MPI_SUM**: Sum
- **MPI_LAND**: Logical and
- **MPI_LOR**: Logical or
- **MPI_LXOR**: Logical exclusive or
- **MPI_BAND**: Bitwise and
- **MPI_BOR**: Bitwise or
- **MPI_BXOR**: Bitwise exclusive or
- **MPI_MAXLOC**: Maximum value and location
- **MPI_MINLOC**: Minimum value and location
Global Sum with MPI_Reduce

C

double sum_partial, sum_global;
sum_partial = ...;
    ierr = MPI_Reduce(&sum_partial, &sum_global,
                      1, MPI_DOUBLE_PRECISION,
                      MPI_SUM,root,
                      MPI_COMM_WORLD);

Fortran

double precision sum_partial, sum_global
sum_partial = ...
    call MPI_Reduce(sum_partial, sum_global,
                    1, MPI_DOUBLE_PRECISION,
                    MPI_SUM,root,
                    MPI_COMM_WORLD, ierr)
Global Sum with MPI_Reduce

mpi4py

# each processor does a local sum
total=0
for i in range(0, count):
    total=total+myray[i]
print("myid=",myid,"total=",total)

# reduce back to the root and print
# reduce(self, sendobj, op=SUM, int root=0)
# Reduce(self, sendbuf, recvbuf, Op op=SUM, int root=0)
if lower:
    back_ray=comm.reduce(total)
else:
    back_ray=empty(2,"i")  # Why does this need to be 2?
    # Maybe to support the max_loc operation?
    # However, MAXLOC does not work?
    comm.Reduce(total,back_ray,op=MPI.SUM,root=mpi_root)
if myid == mpi_root:
    print("results from all processors=",back_ray)
Global Sum with MPI_Reduce

P_ex06.py
This program shows how to use MPI_Scatter and MPI_Reduce. Each processor gets different data from the root processor by way of mpi_scatter. The data is summed and then sent back to the root processor using MPI_Reduce. The root processor then prints the global sum.

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.bcast()
- comm.Scatter()
- comm.reduce()
- comm.Reduce()
- MPI.Finalize
Global Sum with MPI_Reduce

2d array spread across processors

<table>
<thead>
<tr>
<th></th>
<th>X(0)</th>
<th>X(1)</th>
<th>X(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE 0</td>
<td>A0</td>
<td>B0</td>
<td>C0</td>
</tr>
<tr>
<td>NODE 1</td>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>NODE 2</td>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
</tbody>
</table>

Global Sum with MPI_Reduce

2d array spread across processors

<table>
<thead>
<tr>
<th></th>
<th>X(0)</th>
<th>X(1)</th>
<th>X(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE 0</td>
<td>A0+A1+A2</td>
<td>B0+B1+B2</td>
<td>C0+C1+C2</td>
</tr>
<tr>
<td>NODE 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NODE 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
All Gather and All Reduce

- Gather and Reduce come in an "ALL" variation
- Results are returned to all processors
- The root parameter is missing from the call
- Similar to a gather or reduce followed by a broadcast
Global Sum with MPI_AllReduce

2d array spread across processors

<table>
<thead>
<tr>
<th></th>
<th>X(0)</th>
<th>X(1)</th>
<th>X(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE 0</td>
<td>A0</td>
<td>B0</td>
<td>C0</td>
</tr>
<tr>
<td>NODE 1</td>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>NODE 2</td>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Y(0)</th>
<th>Y(1)</th>
<th>Y(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE 0</td>
<td>A0+A1+A2</td>
<td>B0+B1+B2</td>
<td>C0+C1+C2</td>
</tr>
<tr>
<td>NODE 1</td>
<td>A0+A1+A2</td>
<td>B0+B1+B2</td>
<td>C0+C1+C2</td>
</tr>
<tr>
<td>NODE 2</td>
<td>A0+A1+A2</td>
<td>B0+B1+B2</td>
<td>C0+C1+C2</td>
</tr>
</tbody>
</table>
All to All communication with MPI_Alltoall

- Each processor sends and receives data to/from all others

C

- int MPI_Alltoall(&sendbuf, sendcnts, sendtype,
  &recvbuf, recvcnts, recvtype, comm);

Fortran

- call MPI_Alltoall(sendbuf, sendcnts, sendtype,
  recvbuf, recvcnts, recvtype, comm, ierror)
All to All communication with MPI_Alltoall

- mpi4py
  - `alltoall(self, sendobj)`
  - `Alltoall(self, sendbuf, recvbuf)`
  - Parameters
- Each processor sends and receives the same amount of data to/from all others
All to All with MPI_Alltoall

- Parameters
  - Sendcmts # of elements sent to each processor
  - Sendbuf is an array of size sendcmts
  - Recvcnts # of elements obtained from each processor
  - Recvbuf of size Recvcnts*number of processors
  - Note that both send buffer and receive buffer must be an array of (size of the number of processors)*N
All to All with MPI_Alltoall

P_ex07.py
This program shows how to use MPI_Alltoall. Each processor send/rec a different random number to/from other processors.

• MPI.COMM_WORLD
• Get_rank()
• Get_size()
• comm.Alltoall()
• MPI.Finalize
Things Left

- “V” operations
- Communicators
- Derived typed
- Parallel IO
- See simple example
- http://mpi4py.scipy.org/docs/usrman/tutorial.html#mpi-io
- Real life examples
  - Finite Difference Code
  - Bag of tasks
The dreaded “V” or variable or operators

• A collection of very powerful but difficult to setup global communication routines (actually easier in mpi4py)

• MPI_Gatherv: Gather different amounts of data from each processor to the root processor

• MPI_Alltoallv: Send and receive different amounts of data form all processors

• MPI_Allgatherv: Gather different amounts of data from each processor and send all data to each

• MPI_Scatterv: Send different amounts of data to each processor from the root processor

• We discuss MPI_Gatherv, MPI_Scatterv, and MPI_Alltoallv
MPI_Gatherv

Task 0

Different amounts of data are sent from each task to the root
Each task must know how much data is being sent
The root must know how much and where to put it.
MPI_Gatherv

- This is legal also

Task 0 | Task 1 | Task 2
---|---|---
1 2 3 | 10 20 30 40 50 60 | 100 200 300 400 500

- Different amounts of data are sent from each task to the root
- Each task must know how much data is being sent
- The root must know how much and where to put it.
MPI_Gatherv

• C

```c
int MPI_Gatherv (&sendbuf, sendcnts, sendtype, &recvbuf, &rcvcnts, &rdispls, recvtype, root, comm);
```

• Fortran

```fortran
MPI_Gatherv (sendbuf, sendcnts, sendtype, recvbuf, rcvcnts, rdispls, recvtype, root, comm, ierror)
```

• Parameters:

  – `Recvcnts` is now an array
  – `Rdispls` is a displacement
MPI_Gatherv

- **Recvcnts**
  - An array of extent Recvcnts(0:N-1) where Recvcnts(N) is the number of elements to be received from processor N

- **Rdispls**
  - An array of extent Rdispls(0:N-1) where Rdispls(N) is the offset, in elements, from the beginning address of the receive buffer to place the data from processor N

- **Typical usage**

  ```
  recvcnts=...
  rdispls(0)=0
  do I=1,n-1
     rdispls(I) = rdispls(I-1) + recvcnts(I-1)
  enddo
  ```
MPI_Gatherv

- mpi4py
  
  - **Gatherv** (self, sendbuf, recvbuf, int root=0) Parameters
    
    - Like MPI_Bcast, a root is specified.
    - Operation is a type of mathematical operation

- Parameters:
  
  - Recvcnts is now an array
  
  - Rdispls is a displacement
  
  - WHERE ARE THEY?
MPI_Gatherv

- `Recvcnts` is now an array
- `Rdispls` is a displacement
- **WHERE ARE THEY?**

**Added as a tuple to the recvbuf:**

```
recvbuf=[allray, (counts,displacements), MPI.INT]
```
MPI_Gatherv Example

**P_ex08.py**
This program shows how to use MPI_Gatherv. Each processor sends a different amount of data to the root processor. We use MPI_Gather first to tell the root how much data is going to be sent.

- MPI.COMM_WORLD.Get_rank()
- Get_size()
- comm.gather
- comm.Gatherv
- MPI.Finalize
MPI_Alltoallv

- Send and receive different amounts of data form all processors

- C

  ```
  int MPI_Alltoallv (&sendbuf, &sendcnts, &sdispls, sendtype, &recvbuf, &recvcnts, &rdispls, recvtype, comm);
  ```

- Fortran

  ```
  Call MPI_Alltoallv(sendbuf, sendcnts, sdispls, sendtype, recvbuf, recvcnts, rdispls,recvtype, comm,ierror);
  ```
MPI_Alltoallv

- We add **sdispls** parameter
  - An array of extent **sdispls(0:N-1)** where **sdispls(N)** is the offset, in elements, from the beginning address of the send buffer to get the data for processor N

- Typical usage

  ```
  recvcnts=...
  Sendcnts=...
  rdispl(0)=0
  Sdispl(0)=0
  do I=1,n-1
  rdispl(I) = rdispl(I-1) + recvnt(I-1)
  sdispl(I) = sdispl(I-1) + sendcnt(I-1)
  Enddo
  ```
MPI_Alltoallv Example

We just showed how to calculate the displacement arrays but in mpi4py they are optional if you do the "normal" thing of placing data in task order.

```
comm.Alltoallv(sendbuf=[sray, scounts, MPI.INT], recvbuf=[rec, rcounts, MPI.INT])
```

**P_ex09.py**
This program shows how to use Alltoallv Each processor gets amounts of data from each other processor. It is an extension to example P_ex07.py. In mpi4py the displacement array can be calculated automatically from the rcounts array. We show how it would be done in "normal" MPI. See also P_ex08.py for how this can be done

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.Alltoall()
- comm.Alltoallv()
- MPI.Finalize
MPI_Scatterv

- The root must know how much and where to put it.
- Each task must know how much data is being sent.
- Different amounts of data are sent from each task to the root.

Task 0

```plaintext
0 10 20 30 40 50 60
100 200 300 400 500
1 2 3
```

- MPI_Gatherv
P_ex13.py
This program shows how to use Scatterv. Each processor gets a different amount of data from the root processor. We use MPI_Gather first to tell the root how much data is going to be sent.
- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.gather
- comm.Scatterv
- MPI.Finalize
Derived types

- C and Fortran 90 have the ability to define arbitrary data types that encapsulate reals, integers, and characters.
- MPI allows you to define message data types corresponding to your data types.
- Can use these data types just as default types.
Derived types, Three main classifications:

- Contiguous Vectors: enable you to send contiguous blocks of the same type of data lumped together
- Noncontiguous Vectors: enable you to send noncontiguous blocks of the same type of data lumped together
- Abstract types: enable you to (carefully) send C or Fortran 90 structures, don't send pointers
Derived types, how to use them

- Three step process
  - Define the type using
    - `MPI_TYPE_CONTIGUOUS` for contiguous vectors
    - `MPI_TYPE_VECTOR` for noncontiguous vectors
    - `MPI_TYPE_STRUCT` for structures
  - Commit the type using
    - `MPI_TYPE_COMMIT`
  - Use in normal communication calls
    - `MPI_Send(buffer, count, MY_TYPE, destination, tag, MPI_COMM_WORLD, ierr)`
MPI_TYPE_CONTIGUOUS

- Defines a new data type of length count elements from your old data type

- C
  
  `MPI_TYPE_CONTIGUOUS(int count, old_type, &new_type)`

- Fortran
  
  `Call MPI_TYPE_CONTIGUOUS(count, old_type, new_type, ierror)`

- Parameters
  
  - `Old_type`: your base type
  - `New_type`: a type count elements of `Old_type`
MPI_TYPE_VECTOR

- Defines a datatype which consists of \texttt{count} blocks each of length \texttt{blocklength} and \texttt{stride} displacement between blocks

- **C**
  - \texttt{MPI_TYPE_VECTOR(count, blocklength, stride, old_type, *new_type)}

- **Fortran**
  - \texttt{Call MPI_TYPE_VECTOR(count, blocklength, stride, old_type, new_type, ierror)}

- We will see examples later
MPI_TYPE_STRUCT

• Defines a MPI datatype which maps to a user defined derived datatype

• C

  – int MPI_TYPE_STRUCT(count, 
    &array_of_blocklengths, 
    &array_of_displacement, &array_of_types, 
    &newtype);

• Fortran

  – Call MPI_TYPE_STRUCT(count, 
    array_of_blocklengths, 
    array_of_displacement, array_of_types, 
    newtype,ierror)
MPI_TYPE_STRUCT

- **Parameters:**
  - **[IN count]** # of old types in the new type (integer)
  - **[IN array_of_blocklengths]** how many of each type in new structure (integer)
  - **[IN array_of_types]** types in new structure (integer)
  - **[IN array_of_displacement]** offset in bytes for the beginning of each group of types (integer)
  - **[OUT newtype]** new datatype (handle)

- Call `MPI_TYPE_STRUCT(count, array_of_blocklengths, array_of_displacement, array_of_types, newtype, ierror)`
Consider the data type or structure consisting of:

- 3 MPIDOUBLEPRECISION
- 10 MPIINTEGER
- 2 MPILOGICAL

Creating the MPI data structure matching this C/Fortran structure is a three step process:

1. Fill the descriptor arrays:
   - B - blocklengths
   - T - types
   - D - displacements

2. Call MPI_TYPE_STRUCT to create the MPI data structure

3. Commit the new data type using MPI_TYPE_COMMIT

Derived Data type Example
Derived Data type Example

- Consider the data type or structure consisting of
  - 3 MPI_DOUBLE_PRECISION
  - 10 MPI_INTEGER
  - 2 MPI_LOGICAL

- To create the MPI data structure matching this C/Fortran structure
  - Fill the descriptor arrays:
    - B - blocklengths
    - T - types
    - D - displacements

- Call MPI_TYPE_STRUCT
Derived Data type Example (continued)

! t contains the types that make up the structure
  t(1)=MPI_DOUBLE_PRECISION
  t(2)=MPI_INTEGER
  t(3)=MPI_LOGICAL
! b contains the number of each type
  b(1)=3; b(2)=10; b(3)=2
! d contains the byte offset of the start of each type
! the start of each type
  d(1)=0; d(2)=24; d(3)=64

call MPI_TYPE_STRUCT(3,b,d,t,
    MPI_CHARLES,mpi_err)

MPI_CHARLES is our new data type
MPI_Type_commit

- Before we use the new data type we call MPI_Type_commit
- C
  - MPI_Type_commit(MPI_CHARLES)
- Fortran
  - Call MPI_Type_commit(MPI_CHARLES,ierr)
Communicators

- In “normal” MPI a communicator is a parameter in all MPI message passing routines.
- In mpi4py a communicator is a class object that has message passing routines as methods.
- A communicator is a collection of processors that can engage in communication.
- MPI_COMM_WORLD is the default communicator that consists of all processors.
- MPI allows you to create subsets of communicators.
Why Communicators?

- Isolate communication to a small number of processors
- Useful for creating libraries
- Different processors can work on different parts of the problem
- Useful for communicating with "nearest neighbors"
MPI_Comm_create

- MPI_Comm_create creates a new communicator newcomm with group members defined by a group data structure.
- C
  - int MPI_Comm_create(comm, group, &newcomm)
- Fortran
  - Call MPI_COMM_CREATE(comm, GROUP, NEWCOMM, IERROR)
- mpi4py
  - newcom=comm.Create(group)
- How do you define a group?
MPI_Comm_group

- Given a communicator, MPI_Comm_group returns in group associated with the input communicator.

- C
  - int MPI_Comm_group(comm, &group)

- Fortran
  - Call MPI_COMM_GROUP(COMM, GROUP, IERROR)
MPI_Comm_group

- Given a communicator, MPI_Comm_group returns in group associated with the input communicator
- mpi4py
  - old_group=comm.Get_group()
- As we have seen comm is an object. Get_group is a method
- Groups “old_group” is also an object with a collection of methods
MPI_Group_incl

- MPI_Group_incl creates a group new_group that consists of the n processes in old_group with ranks rank[0],..., rank[n-1]

- C
  
  ```c
  int MPI_Group_incl(group, n, &ranks, &new_group)
  ```

- Fortran
  
  ```fortran
  Call MPI_GROUP_INCL(GROUP, N, RANKS, NEW_GROUP, IERROR)
  ```
MPI_Group_incl

- Fortran
  - Call MPI_GROUP_INCL(old_GROUP, N, RANKS, NEW_GROUP, IERROR)

- Parameters
  - old_group: your old group
  - N: number of elements in array ranks (and size of new_group) (integer)
  - Ranks: ranks of processes in group to appear in new_group (array of integers)
  - New_group: new group derived from above, in the order defined by ranks
MPI_Group_incl

- MPI_Group_incl creates a group `new_group` that consists of the `n` processes in `old_group` with ranks `rank[0],..., rank[n-1]`

- mpi4py
  - `new_group=old_group.Incl(ranks)`
Create communicator...

```python
# get our old group from MPI_COMM_WORLD
old_group=comm.Get_group()

# create a new group from the old group
# containing a subset of the processors
num_used=... will_use=zeros(num_used,"i")
for ijk in range(0, num_used):
    will_use[ijk]=...

new_group=old_group.Incl(will_use)

# create the new communicator
sub_comm_world=comm.Create(new_group)
```
Create communicator Example

P_ex10.py, ex10.in
Pass a "token" from one task to the next with a single task reading token from a file.

An extensive program. We first create a new communicator that contains every task except the zeroth one. This is done by first defining a group, new_group. We define new_group based on the group associated with mpi_comm_world, old_group and an array, will_use. Will_use contains a list of tasks to be included in the new group. It does not contain the zeroth one. The new communicator is sub_comm_world.

There are other ways to create communicator but this is one of the more general methods.

Next, we have break of the task not in the communicator to call the routine get_input. This routine will do input from a file ex10.in. The file contains a list of integers. The task will send the integer to the first task in the new communicator.

The remaining tasks which are part of sub_comm_world call the routine pass_token. Pass_token "just" receives a value from the previous processor and passes it on to the next.

There is a minor subtlety in pass_token. We are using both our new communicator and MPI_COMM_WORLD. The tasks that are part of the new communicator use it to pass data. We note that the task that is injecting values into the stream is not part of the new communicator so it must use MPI_COMM_WORLD. Thus we do a probe on WORLD, which is actually MPI_COMM_WORLD looking for a message. When we get it we send it on using the new communicator.

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- WORLD.Iprobe()
- comm.Get_group()
- old_group.Incl()
- comm.Create()
- new_group.Get_rank()
- MPI.Finalize
MPI_Group_excl

- MPI_Group_excl creates a group of processes new_group that is obtained by deleting from old_group those processes with ranks ranks[0], ..., ranks[n-1]
MPI_Comm_split

- Provides a short cut method to create a collection of communicators
- All processors with the "same color" will be in the same communicator
- Index gives rank in new communicator

- Fortran
  - call MPI_COMM_SPLIT(OLD_COMM, color, index, NEW_COMM, mpi_err)

- C
  - MPI_Comm_split(OLD_COMM, color, index, &NEW_COMM)
MPI_Comm_split

- Provides a short cut method to create a collection of communicators
- All processors with the "same color" will be in the same communicator
- Index gives rank in new communicator
- mpi4py
  - `new_comm = old_comm.Split(color, index)`
MPI_Comm_split

• Split odd and even processors into 2 communicators

Program comm_split
include "mpif.h"
Integer color,zero_one

call MPI_INIT( mpi_err )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numnodes, mpi_err )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, mpi_err )
color=mod(myid,2) !color is either 1 or 0
call MPI_COMM_SPLIT(MPI_COMM_WORLD,color,myid,NEW_COMM,mpi_err)
call MPI_COMM_RANK( NEW_COMM, new_id, mpi_err )
call MPI_COMM_SIZE( NEW_COMM, new_nodes, mpi_err )
Zero_one = -1
If(new_id==0)Zero_one = color
Call MPI_Bcast(Zero_one,1,MPI_INTEGER,0, NEW_COMM,mpi_err)
If(zero_one==0)write(*,*)"part of even processor communicator"
If(zero_one==1)write(*,*)"part of odd processor communicator"
Write(*,*)"old_id=", myid, "new_id=", new_id
Call MPI_FINALIZE(mpi_error)
End program
• Split odd and even processors into 2 communicators

Program comm_split
include "mpif.h"
Integer color,zero_one
call MPI_INIT( mpi_err )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numnodes, mpi_err )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, mpi_err )
color=mod(myid,2) !color is either 1 or 0
call MPI_COMM_SPLIT(MPI_COMM_WORLD,color,myid,NEW_COMM,mpi_err)
call MPI_COMM_RANK( NEW_COMM, new_id, mpi_err )
call MPI_COMM_SIZE( NEW_COMM, new_nodes, mpi_err )
Zero_one = -1
If(new_id==0)Zero_one = color
Call MPI_Bcast(Zero_one,1,MPI_INTEGER,0, NEW_COMM,mpi_err)
If(zero_one==0)write(*,*)"part of even processor communicator"
If(zero_one==1)write(*,*)"part of odd processor communicator"
Write(*,*)"old_id=", myid, "new_id=", new_id
Call MPI_FINALIZE(mpi_error)
End program
MPI_Comm_split

- Split odd and even processors into 2 communicators

**P_ex12.py**

This program shows how to use mpi_comm_split

Split will create a set of communicators. All of the tasks with the same value of color will be in the same communicator. In this case we get two sets one for odd tasks and one for even tasks. Note they have the same name on all tasks, new_comm, but there are actually two different values (sets of tasks) in each one.

- MPI.COMM_WORLD
- Get_rank()
- Get_size()
- comm.Split
- comm.bcast
- MPI.Finalize
MPI_Comm_split example output

• Note, I have sorted the output

```bash
osage:mpi4py tkaiser$ mpiexec -n 6 ./P_ex12.py | sort
color to integer= {'blue ': 0, 'green ': 1, 'red ': 2, 'yellow': 3} and
integer to color= {0: 'blue ', 1: 'green ', 2: 'red ', 3: 'yellow'}
hello from 0 of 6
hello from 1 of 6
hello from 2 of 6
hello from 3 of 6
hello from 4 of 6
hello from 5 of 6
myid= 0 color integer = 0 color name = blue
myid= 1 color integer = 1 color name = green
myid= 2 color integer = 0 color name = blue
myid= 3 color integer = 1 color name = green
myid= 4 color integer = 0 color name = blue
myid= 5 color integer = 1 color name = green
new id is 0 in the blue communicator or 0.blue original id is 0 id bcast from root 0
new id is 0 in the green communicator or 0.green original id is 1 id bcast from root 1
new id is 1 in the blue communicator or 1.blue original id is 2 id bcast from root 0
new id is 1 in the green communicator or 1.green original id is 3 id bcast from root 1
new id is 2 in the blue communicator or 2.blue original id is 4 id bcast from root 0
new id is 2 in the green communicator or 2.green original id is 5 id bcast from root 1
osage:mpi4py tkaiser$
```