MPI

More of the Story

Timothy H. Kaiser, PH.D.

tkaiser@mines.edu
Outline

- Review
- Types
- Wildcards
- Using Status and Probing
- Asynchronous Communication, first cut
- Global communications
- Advanced topics
  - "V" operations
  - Derived types
  - Communicators
http://geco.mines.edu/workshop
Six basic MPI calls

**MPI_INIT**
Initialize MPI

**MPI_COMM_RANK**
Get the processor rank

**MPI_COMM_SIZE**
Get the number of processors

**MPI_Send**
Send data to another processor

**MPI_Recv**
Get data from another processor

**MPI_FINALIZE**
Finish MPI
Send and Receive Program Fortran

program send_receive
include "mpif.h"
integer myid,ierr,numprocs,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
if(myid .eq. source)then
    buffer=5678
    Call MPI_Send(buffer, count, MPI_INTEGER,destination,&
        tag, MPI_COMM_WORLD, ierr)
    write(*,*)"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(*,*)"processor ",myid," got ",buffer
endif
call MPI_FINALIZE(ierr)
stop
end
Send and Receive Program C

```c
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int myid, numprocs, tag, source, destination, count, buffer;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    tag = 1234; source = 0; destination = 1; count = 1;
    if (myid == source){
        buffer = 5678;
        MPI_Send(&buffer, count, MPI_INT, destination, tag, MPI_COMM_WORLD);
        printf("processor %d sent %d\n", myid, buffer);
    }
    if (myid == destination){
        MPI_Recv(&buffer, count, MPI_INT, source, tag, MPI_COMM_WORLD, &status);
        printf("processor %d got %d\n", myid, buffer);
    }
    MPI_Finalize();
}
```
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>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td>-</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>-</td>
</tr>
</tbody>
</table>
## Predefined types in Fortran

<table>
<thead>
<tr>
<th>Fortran MPI Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPI_INTEGER</strong></td>
</tr>
<tr>
<td><strong>MPI_REAL</strong></td>
</tr>
<tr>
<td><strong>MPI_DOUBLE PRECISION</strong></td>
</tr>
<tr>
<td><strong>MPI_COMPLEX</strong></td>
</tr>
<tr>
<td><strong>MPI_LOGICAL</strong></td>
</tr>
<tr>
<td><strong>MPI_CHARACTER</strong></td>
</tr>
<tr>
<td><strong>MPI_BYTE</strong></td>
</tr>
<tr>
<td><strong>MPI_PACKED</strong></td>
</tr>
</tbody>
</table>

- INTEGER
- REAL
- DOUBLE PRECISION
- COMPLEX
- LOGICAL
- CHARACTER(1)
- -
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
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)`
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...
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
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_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)
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 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
MPI_Wait used to complete communication

- Request from lsend or lrecv 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_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
Exercise 3: Asynchronous Send and Receive

- Write a parallel program to send and receive data using MPI_Isend and MPI_Irecv
  - Initialize MPI
  - Have processor 0 send an integer to processor 1
  - Have processor 1 receive and integer from processor 0
  - Both processors check on message completion
- Quit MPI
MPI Broadcast call: MPI_Bcast

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

C

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

Fortran

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

- Root is node that sends the message
Exercise 4 : Broadcast

- Write a parallel program to broadcast data using MPI_Bcast
  - Initialize MPI
  - Have processor 0 broadcast an integer
  - Have all processors print the data
  - Quit MPI
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:

P_0 P_1 P_2 ... P_{n-1}
MPI_Scatter

- **C**

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

- **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
Scatter Operation using MPI_Scatter

- Scatter with Sendcnts = 2

Data in an array on root node:

\[
A(0) \ A(2) \ A(4) \ldots \ A(2N-2) \\
A(1) \ A(3) \ A(5) \ldots \ A(2N-1)
\]

Goes to processors:

\[
P_0 \quad P_1 \quad P_2 \quad \ldots \quad P_{n-1} \\
B(0) \quad B(0) \quad B(0) \quad \quad \quad \quad B(0) \\
B(1) \quad B(1) \quad B(1) \quad \quad \quad \quad B(1)
\]
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:

<table>
<thead>
<tr>
<th>P_0</th>
<th>P_1</th>
<th>P_2</th>
<th>...</th>
<th>P_{n-1}</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td>A</td>
</tr>
</tbody>
</table>

Goes to an array on root node:

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

- C
  – int MPI_Gather(&sendbuf, sendcnts, sendtype, &recvbuf, recvcnts, recvtype, root, comm);

- 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
Exercise 5: Scatter and Gather

- Write a parallel program to scatter real data using MPI_Scatter
- Each processor sums the data
- Use MPI_Gather to get the data back to the root processor
- Root processor sums and prints the data
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**
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)
Exercise 6: Global Sum with MPI_Reduce

- Write a program to sum data from all processors
Global Sum with MPI_Reduce

2d array spread across processors

<table>
<thead>
<tr>
<th>NODE 0</th>
<th>X(0)</th>
<th>X(1)</th>
<th>X(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>

NODE 0: A0 + A1 + A2
NODE 1: B0 + B1 + B2
NODE 2: C0 + C1 + C2
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
  ```c
  int MPI_Alltoall(&sendbuf, sendcnts, sendtype, recvbuf, recvcnts, recvtype, root, MPI_Comm);
  ```
- Fortran
  ```fortran
  call MPI_Alltoall(sendbuf,sendcnts,sendtype,recvbuf,recvcnts,recvtype,root,comm,ierror)
  ```
All to All with MPI_Alltoall

- Parameters
  - Sendcnts # of elements sent to each processor
  - Sendbuf is an array of size sendcnts
  - 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
Things Left

- “V” operations
- Communicators
- Derived typed
- Parallel IO
A collection of very powerful but difficult to setup global communication routines

- **MPI_Gatherv**: Gather different amounts of data from each processor to the root processor
- **MPI_Alltoallv**: Send and receive different amounts of data from 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** and **MPI_Alltoallv**
MPI_Gatherv

• C

  - int MPI_Gatherv (&sendbuf, sendcnts, sendtype, &recvbuf, &recvvcnts, &rdispls, recvtype, root, comm);

• Fortran

  - MPI_Gatherv (sendbuf, sendcnts, sendtype, recvbuf, recvvcnts, rdispls, recvtype, root, comm, ierror)

• Parameters:

  - Recvcnts is now an array

  - Rdispls is a displacement
MPI_Gather

• **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**

```plaintext
recvcnts=...

rdispls(0)=0
do I=1,n-1
   rdispls(I) = rdispls(I-1) + recvnts(I-1)
endo
doi
```
MPI_Gatherv Example

- 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.
• Send and receive different amounts of data from all processors

• C

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

• Fortran

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

```fortran
recvcnts=...
Sendcnts=...
rdispls(0)=0
Sdispls(0)=0
do I=1,n-1
   rdispls(I) = rdispls(I-1) + recvcnts(I-1)
sdispls(I) = sdispls(I-1) + sendcnts(I-1)
Enddo
```
MPI_Alltoally example

- Each processor send/rec a different and random amount of data to/from other processors.
- We use MPI_Alltoall first to tell how much data is going to be sent.
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 count blocks each of length blocklength and stride displacement between blocks

- C
  - MPI_TYPE_VECTOR(count, blocklength, stride, old_type, *new_type)

- Fortran
  - 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 MPI_DOUBLE_PRECISION
10 MPI_INTEGER
2 MPI_LOGICAL

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
  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
• 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

- A communicator is a parameter in all MPI message passing routines
- 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

  ```c
  #int MPI_Comm_create(old_comm, group, &newcomm)
  ```

- Fortran

  ```fortran
  MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
  ```

- 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 provides several functions to manipulate existing groups.
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
  
  ```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_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]

• C

  - int MPI_Group_excl(old_group, n, &ranks, MPI_Group &new_group)

• Fortran

  - Call MPI_GROUP_EXCL(OLD_GROUP, N, RANKS, NEW_GROUP, IERROR)
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

• 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
Note, I have sorted the output

```
[mbpro:~] tkaiser% mpiexec -np 8 split.exe | sort
old_id= 0 new_id= 0
old_id= 1 new_id= 0
old_id= 2 new_id= 1
old_id= 3 new_id= 1
old_id= 4 new_id= 2
old_id= 5 new_id= 2
old_id= 6 new_id= 3
old_id= 7 new_id= 3
part of even processor communicator
part of even processor communicator
part of even processor communicator
part of even processor communicator
part of odd processor communicator
part of odd processor communicator
part of odd processor communicator
part of odd processor communicator
```

[mbpro:~] tkaiser%
 MPI_Comm_split output with task labels

- Split odd and even processors into 2 communicators

0: part of even processor communicator
0: old_id= 0 new_id= 0

2: part of even processor communicator
2: old_id= 2 new_id= 1

1: part of odd processor communicator
1: old_id= 1 new_id= 0

3: part of odd processor communicator
3: old_id= 3 new_id= 1
This program is designed to show how to set up a new communicator. We set up a communicator that includes all but one of the processors. The last processor is not part of the new communicator, TIMS_COMM_WORLD.

We use the routine MPI_Group_rank to find the rank within the new communicator. For the last processor the rank is MPI_UNDEFINED because it is not part of the communicator. For this processor we call get_input. The processors in TIMS_COMM_WORLD pass a token between themselves in the subroutine pass_token. The remaining processor gets input, i, from the terminal and passes it to processor 1 of MPI_COMM_WORLD. If i > 100 the program stops.