Parallel Programming with MPI
(Introduction to Advanced)

Timothy H. Kaiser, Ph.D.
tkaiser@sdsc.edu
San Diego Supercomputer Center
Overview

- Background
- Hello world in MPI
- Basic communications
- Types
- Wildcards
- Reductions
- Advanced topics
  - Communicators
  - Derived types
  - AlltoAllV
Background

- MPI - Message Passing Interface
- Library standard defined by committee of vendors, implementers, and parallel programmers
- Available on almost all parallel machines in C and Fortran
- Used to create parallel SPMD programs based on 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
```
Documentation

• SDSC MPI Documentation
  http://www.npaci.edu/Resources/Applications/MPI

• MPI home page
  http://www.mcs.anl.gov/mpi

• Books
  http://www.epm.ornl.gov/~walker/mpi/books.html
  "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
Initialization

• Every MPI program needs these initialization lines.

• C

```c
#include <mpi.h>
/* Initialize MPI */
MPI_Init(&argc, &argv);
/* How many total PEs are there */
MPI_Comm_size(MPI_COMM_WORLD, &nPEs);
/* What node am I (what is my rank? */
MPI_Comm_rank(MPI_COMM_WORLD, &iam);
...
MPI_Finalize();
```

• Fortran

```fortran
include 'mpif.h'
c Initialize MPI
call MPI_Init(ierr)
c Find total number of PEs
call MPI_Comm_size(MPI_COMM_WORLD, nPEs, ierr)
c Find the rank of this node
call MPI_Comm_rank(MPI_COMM_WORLD, iam, ierr)
...
call MPI_Finalize(ierr)
```
Exercise 1 : Hello World

Exercise 1 : Hello World

• write a parallel hello world program
  Initialize MPI
  Have each node print out its node number
  Quit MPI

• C Solution : http://www.sdsc.edu/~tkaiser/source/hello.c

• Fortran Solution : http://www.sdsc.edu/~tkaiser/source/hello.f
Communication Basics

• Bytes transferred from one processor to another

• Specify destination, data buffer, and message ID (called a tag)

• Synchronous send: send call does not return until the message is sent

• Asynchronous send: send call returns immediately, send occurs during other calculation ideally

• Synchronous receive: receive call does not return until the message has been received (may involve a significant wait)

• 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
Synchronous Send

• Send a message to a processor

• C

    MPI_Send(&buffer, count, 
            datatype, 
            destination, tag, 
            communicator);

• Fortran

    call MPI_Send(buffer, count, 
                  datatype, 
                  destination, tag, 
                  communicator, ierr)

• Execution blocked until message in channel
MPI_Send Parameters

MPI_Send Parameters

• **buffer**: Beginning address of data
• **count**: Length of source array (in elements)
• **datatype**: Type of data, for example: MPI_DOUBLE_PRECISION, MPI_INT, etc
• **destination/source**: Logical processor number of destination/source processor
• **tag**: Message type (arbitrary integer)
• **communicator**: Signifies a set of processors to whom the message is sent
• **ierr**: Error return (Fortran only)
Synchronous Receive

- Blocking receive of a message from another processor

- C

  ```c
  MPI_Recv(&buffer, COUNT, datatype, source, tag, communicator, &status);
  ```

- Fortran

  ```fortran
  call MPI_Recv(buffer, COUNT, datatype, source, tag, communicator, status, ierr)
  ```
# MPI types

## C MPI Types

<table>
<thead>
<tr>
<th>MPI Type</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_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>

## Fortran MPI Types

<table>
<thead>
<tr>
<th>MPI Type</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>
</tbody>
</table>
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, 5, MPI_INTEGER,
                     MPI_ANY_SOURCE,
                     MPI_ANY_TAG,
                     MPI_COMM_WORLD,&status);
```

• MPI_ANY_SOURCE and MPI_ANY_TAG are wild cards

• The status structure can be used to clarify wildcard options
The status parameter

- The status parameter returns additional information
- Parameter of 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)
Broadcast

- One node sends a message (root)
- All others receive the message in the same memory space.
- Execution blocked until all processors arrive to broadcast call.
- Automatically acts as a synchronizing point.

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

- Fortran
  
  ```fortran
  call MPI_Bcast(buffer, COUNT,
                 datatype,
                 root,
                 communicator, ierr)
  ```
Parallel Reductions

Definitions

• Used to combine partial results from all processors
• Called a parallel prefix or parallel reduction
• Processor i has an array Yi(K)
• Corresponding values on processors combined
• Works on 1 or 2d arrays
• Result returned to 1 or all processors
MPI Reduction Subroutines

• 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)

• Like MPI_Bcast, a root is specified.

• Results are only sent back to the root node.

• Also available: MPI_Allreduce().
Types of Global Operations

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

• MPI_Op_create can be used to bind a user-defined global operation to an op handle.
Example of Computing a Global Sum with MPI_Allreduce

- Each processor has variables sum and sum_global
- Value of sum_global updated on all processors

- C
  ```c
double sum_partial, sum_global;
  sum_partial = ...;
  ierr = MPI_Allreduce(&sum_partial, &sum_global,
                       1, MPI_DOUBLE_PRECISION,
                       MPI_SUM,
                       MPI_COMM_WORLD);
  ```

- Fortran
  ```fortran
double precision sum_partial, sum_global
  sum_partial = ...
  call MPI_Allreduce(sum_partial, sum_global,
                     1, MPI_DOUBLE_PRECISION,
                     MPI_SUM,
                     MPI_COMM_WORLD, ierr)
  ```
Sum on 2d array using MPI_Allreduce

<table>
<thead>
<tr>
<th>NODE 0</th>
<th>X(0)</th>
<th>X(1)</th>
<th>X(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE 1</td>
<td>A0</td>
<td>B0</td>
<td>C0</td>
</tr>
<tr>
<td>NODE 2</td>
<td>A1</td>
<td>B1</td>
<td>C1</td>
</tr>
<tr>
<td>NODE 3</td>
<td>A2</td>
<td>B2</td>
<td>C2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NODE 0</th>
<th>X(0)</th>
<th>X(1)</th>
<th>X(2)</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>NODE 3</td>
<td>A0+A1+A2</td>
<td>B0+B1+B2</td>
<td>C0+C1+C2</td>
</tr>
</tbody>
</table>
Sum on 2d array using MPI_Reduce

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

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

- MPI_BARRIER blocks the caller until all members in the communicator have called it.
- Used as a synchronization tool.
- C
  
  ```c
  int MPI_BARRIER(MPI_Comm comm )
  ```

- Fortran
  
  ```fortran
  INTEGER COMM, IERROR
  MPI_BARRIER(COMM, IERROR)
  ```

- Parameters

  [IN comm] communicator (handle)
Asynchronous Communication

MPI_Isend

• Non Blocking send

• C

```c
int MPI_Isend(void* buf, int count,
               MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm,
               MPI_Request *request)
```

• Fortran

```fortran
MPI_ISEND(BUF, COUNT, DATATYPE,
          DEST, TAG, COMM, REQUEST,IERROR)
```

• Parameters

  [ IN buf] initial address of send buffer (choice)
  [ IN count] # of elements in send buffer (integer)
  [ IN datatype] datatype of send buffer (handle)
  [ IN dest] rank of destination (integer)
  [ IN tag] message tag (integer)
  [ IN comm] communicator (handle)
  [ OUT request] communication request (handle)
Receive

- Non Blocking receive
- C

```c
int MPI_Irecv(void* buf, int count,
               MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm, MPI_Request *request)
```

- Fortran

```fortran
MPI_IRecv(BUF, COUNT, DATATYPE, SOURCE, TAG, 
           COMM, REQUEST, IERROR)
```

- Parameters

  - [OUT buf] initial address of receive buffer (choice)
  - [IN count] # of elements in receive buffer (integer)
  - [IN datatype] datatype of receive buffer (handle)
  - [IN source] rank of source (integer)
  - [IN tag] message tag (integer)
  - [IN comm] communicator (handle)
  - [OUT request] communication request (handle)
MPI_Wait

- Used to complete a nonblocking communication
- The completion of a send operation indicates that the sender is now free to update the locations in the send buffer
- The completion of a receive operation indicates that the receive buffer contains the received message

- C
  
  ```
  int MPI_Wait(MPI_Request *request,
               MPI_Status *status)
  ```

- Fortran
  
  ```
  INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
  MPI_WAIT(REQUEST, STATUS, IERROR)
  ```

- Parameters
  
  [ INOUT request] request (handle)
  [ OUT status] status object (Status)
MPI_Test

MPI_Test

• Similar to MPI_Wait, but does not block.

• Value of flags signifies whether a message has been delivered

• C

  int MPI_Test(MPI_Request *request,
              int *flag, MPI_Status *status)

• Fortran

  LOGICAL FLAG
  INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
  MPI_TEST(REQUEST, FLAG, STATUS, IERROR)

• Parameters

  [ INOUT request] communication request (handle)
  [ OUT flag] true if operation completed (logical)
  [ OUT status] status object (Status)
Non blocking send example

• This example acts like a MPI_Wait

```fortran
    call MPI_Isend (buffer, count, datatype, dest, tag,
                    comm, request, ierr)

    <Do other work>

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

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.

• C
  
  int MPI_Probe( int source, int tag, MPI_Comm comm,
                 MPI_Status *status)

• Fortran
  
  INTEGER SOURCE,TAG,COMM,STATUS(MPI_STATUS_SIZE),
      IERROR
  MPI_PROBE(SOURCE, TAG, COMM, STATUS, IERROR)

• Parameters
  [IN source] source rank, or MPI_ANY_SOURCE (integer)
  [IN tag] tag value, or MPI_ANY_TAG (integer)
  [IN comm] communicator (handle)
  [OUT status] status object (Status)
Using MPI_Probe

! How to use probe and get_count to find the size of a message
program probe_it
include 'mpif.h'
integer myid,numprocs
integer status(MPI_STATUS_SIZE)
integer mytag,i,icount,ierr
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
mytag=123
i=0
icount=0
if(myid .eq. 0)then
  i=100
  icount=1
  call MPI_SEND(i,icount,MPI_INTEGER,1,mytag,MPI_COMM_WORLD,ierr)
endif
if(myid .eq. 1)then
  call mpi_probe(0,mytag,MPI_COMM_WORLD,status,ierr)
  call mpi_get_count(status,MPI_INTEGER,icount,ierr)
  write(*,*)"getting ", icount
  call mpi_recv &
  (i,icount,MPI_INTEGER,0,mytag,MPI_COMM_WORLD,status,ierr)
  write(*,*)"i=",i
endif
  call mpi_finalize(ierr)
stop
end
Communicators

**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(MPI_Comm comm, MPI_Group group, MPI_Comm*newcomm)
  ```

- Fortran
  ```fortran
  MPI_COMM_CREATE(COMM, GROUP, NEWCOMM, IERROR)
  INTEGER COMM, GROUP, NEWCOMM, IERROR
  ```

- Parameters
  - [ IN comm] communicator (handle)
  - [ IN group] Group, which is a subset of the group of comm (handle)
  - [ OUT newcomm] new communicator (handle)

- So How do you define a group?
MPI_Comm_group

- MPI_Comm_group returns in group a handle to the group of comm.

- C
  ```c
  int MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
  ```

- Fortran
  ```fortran
  MPI_COMM_GROUP(COMM, GROUP, IERROR)
  INTEGER COMM, GROUP, IERROR
  ```

- Parameters
  - [ IN comm] communicator (handle)
  - [ OUT group] group corresponding to comm (handle)
MPI_Comm_incl

• MPI provides several functions to manipulate existing groups.
• The function MPI_GROUP_INCL creates a group newgroup that consists of the n processes in group with ranks rank[0], ..., rank[n-1];

• C
  
  int MPI_Group_incl(MPI_Group group, int n, int *ranks, MPI_Group *newgroup)

• Fortran
  
  MPI_GROUP_INCL(GROUP, N, RANKS, NEWGROUP, IERROR)
  INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR

• Parameters
  [ IN group] group (handle)
  [ IN n] number of elements in array ranks (and size of newgroup) (integer)
  [ IN ranks] ranks of processes in group to appear in newgroup (array of integers)
  [ OUT newgroup] new group derived from above, in the order defined by ranks (handle)
MPI_Comm_excl

- The function MPI_GROUP_EXCL creates a group of processes newgroup that is obtained by deleting from group those processes with ranks ranks[0], ..., ranks[n-1]

- C

  ```c
  int MPI_Group_excl(MPI_Group group, int n,
                     int *ranks, MPI_Group *newgroup)
  ```

- Fortran

  ```fortran
  MPI_GROUP_EXCL(GROUP, N, RANKS, NEWGROUP, IERROR)
  INTEGER GROUP, N, RANKS(*), NEWGROUP, IERROR
  ```

- Parameters

  - [ IN group] group (handle)
  - [ IN n] number of elements in array ranks (integer)
  - [ IN ranks] array of integer ranks in group not to appear in newgroup
  - [ OUT newgroup] new group derived from above, preserving the order defined by group (handle)

- See source code 1 http://www.sdsc.edu/~tkaiser/source/f_ex1.f & c_ex1.c
Derived Data types

- C and Fortran 90 have the ability to define arbitrary data types that encapsulate reals, integers, and characters.
- MPI has the functionality to also define arbitrary data types and to pass them between processors.
- Use the two functions
  
  ```c
  MPI_TYPE_STRUCT
  MPI_TYPE_COMMIT
  
  C
  ```
  ```c
  int MPI_TYPE_STRUCT(int count,
                     int *array_of_blocklengths,
                     MPIAint *array_of_displacement,
                     MPI_Datatype *array_of_types,
                     MPI_Datatype *newtype);
  
  int MPI_TYPE_COMMIT(MPI_Datatype *datatype);
  ```
Derived Data types (continued)

- Fortran

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

  MPI_TYPE_COMMIT(newtype, ierror)

- 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)
**Derived Data type Example**

- 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

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

- Call MPI_TYPE_STRUCT to create the MPI data structure

- Commit the new data type using MPI_TYPE_COMMIT
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)
    call MPI_TYPE_COMMIT(MPI_CHARLES,mpi_err)
The dreaded “V” or variable operators

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_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

• MPI_Alltoallv
  Send and receive different amounts of data form all processor
MPI_Gatherv

- C
  int MPI_Gatherv ( void *sendbuf, int *sendcmts,
                  MPI_Datatype sendtype,
                  void *recvbuf, int *recvcnts,
                  int *rdispl,
                  MPI_Datatype recvtype,
                  MPI_Comm comm );

- Fortran
  MPI_Gatherv ( sendbuf, sendcmts, sendtype,
               recvbuf, recvcnts, rdispl,
               recvtype, comm, ierror );
MPI_Gatherv (continued)

• PARAMETERS
  [IN sendbuf] starting address of send buffer (choice)
  [IN sendcounts] integer array equal to the group size specifying the number of elements to send to each processor (integer)
  [IN sendtype] data type of send buffer elements (handle)
  [OUT recvbuf] address of receive buffer (choice)
  [IN recvcounts] array equal to the group size specifying the maximum number of elements that can be received from each processor (integer)
  [IN rdispls] array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i (integer)
  [IN recvtype] data type of receive buffer elements (handle)
  [IN comm ] communicator (handle)

• See source code 2 http://www.sdsc.edu/~tkaiser/source/f_ex2.f & c_ex2.c
MPI_Scatterv

- C
  ```c
  int MPI_Scatterv ( void *sendbuf, int *sendcnts,
                   MPI_Datatype sendtype,
                   void *recvbuf, int *recvcnts,
                   MPI_Datatype recvtype,
                   MPI_Comm comm );
  ```

- Fortran
  ```fortran
  MPI_Scatterv ( sendbuf,sendcnts,sendtype,
                  recvbuf, recvcnts,recvtype,
                  comm,ierror);
  ```
MPI_Scatterv (continued)

- PARAMETERS

  [IN sendbuf] starting address of send buffer (choice)
  [IN sendcounts] integer array equal to the group size specifying the number of elements to send to each processor (integer)
  [IN sdispls] array (of length group size). Entry j specifies the displacement (relative to sendbuf from which to take the outgoing data destined for process j (integer)
  [IN sendtype] data type of send buffer elements (handle)
  [OUT recvbuf] address of receive buffer (choice)
  [IN recvcounts] array equal to the group size specifying the maximum number of elements that can be received from each processor (integer)
  [IN recvtype] data type of receive buffer elements (handle)
  [IN comm ) communicator (handle)

- See source code 3 http://www.sdsc.edu/~tkaiser/source/f_ex3.f & c_ex3.c
**MPI_Alltoallv**

- **C**

  ```c
  int MPI_Alltoallv ( void *sendbuf, int *sendcnts,
                      int *sdispls,
                      MPI_Datatype sendtype,
                      void *recvbuf, int *recvcnts,
                      int *rdispls,
                      MPI_Datatype recvtype,
                      MPI_Comm comm );
  ```

- **Fortran**

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

- PARAMETERS
  - [IN sendbuf] starting address of send buffer (choice)
  - [IN sendcounts] integer array equal to the group size specifying the number of elements to send to each processor (integer)
  - [IN sdispls] array (of length group size). Entry j specifies the displacement (relative to sendbuf from which to take the outgoing data destined for process j) (integer)
  - [IN sendtype] data type of send buffer elements (handle)
  - [OUT recvbuf] address of receive buffer (choice)
  - [IN recvcounts] array equal to the group size specifying the maximum number of elements that can be received from each processor (integer)
  - [IN rdispls] array (of length group size). Entry i specifies the displacement (relative to recvbuf) at which to place the incoming data from process i (integer)
  - [IN recvtype] data type of receive buffer (handle)
  - [IN comm] communicator (handle)

- See source code 4 http://www.sdsc.edu/~tkaiser/source/f_ex4.f & c_ex4.c
Some examples
A parallel merge sort.

Start with a sorted list on each node
active = 1
while (2*active < N)
  active = 2 * active
if(myid >= active) then
  send(data , myid-active)
  return
endif
if(myid + active < N) then
  recv(new_data, myid+active)
  data = merge(data , new_data)
endif
while(active > 1)
  active = active / 2
  if(myid >= active) then
    send(data, myid-active)
  else
    recv(new_data , myid+active)
    data = merge(data, new_data)
  endif
endwhile
Some examples
A parallel merge sort.

<table>
<thead>
<tr>
<th>node</th>
<th>stage1a</th>
<th>stage1b</th>
<th>stage2a</th>
<th>stage2b</th>
<th>stage3a</th>
<th>stage3b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0&lt;-4</td>
<td>merge</td>
<td>0&lt;-2</td>
<td>merge</td>
<td>0&lt;-1</td>
<td>merge</td>
</tr>
<tr>
<td>1</td>
<td>1&lt;-5</td>
<td>merge</td>
<td>1&lt;-3</td>
<td>merge</td>
<td>1-&gt;0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2&lt;-6</td>
<td>merge</td>
<td>2-&gt;0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3&lt;-7</td>
<td>merge</td>
<td>3-&gt;1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4-&gt;0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5-&gt;1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6-&gt;2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7-&gt;3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• See source code 5 http://www.sdsc.edu/~tkaiser/source/f_ex5.f & c_ex5.c
Some examples
myalltoallv

- Algorithm is based on a hypercube algorithm
  - Does not require power of 2 processors
  - Iterate up to power of 2 -1 processors
    - check to see if you are sending to a valid processor
- Uses simple trick to avoid nonblocking send/receive
  - if Myid < partner send first
  - if Myid > partner recv first
Some examples
myalltoally

!find n2, the power of two >= numnodes
  do i=1,n2-1
    !do xor to find the processor xchng
    xchng=xor(i,myid)
    if(xchng <= (numnodes-1))then
      if(myid < xchng)then
        send from myid to xchng
        recv from xchng to myid
      else
        recv from xchng to myid
        send from myid to xchng
      endif
    else
      skip this stage
    endif
  enddo
Some examples
myalltoallv

• Algorithm with 5 nodes

<table>
<thead>
<tr>
<th>stage</th>
<th>node 0</th>
<th>node 1</th>
<th>node 2</th>
<th>node 3</th>
<th>node 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>0 to 1</td>
<td>0 to 1</td>
<td>2 to 3</td>
<td>2 to 3</td>
<td>skip</td>
</tr>
<tr>
<td>1b</td>
<td>1 to 0</td>
<td>1 to 0</td>
<td>3 to 2</td>
<td>3 to 2</td>
<td>skip</td>
</tr>
<tr>
<td>2a</td>
<td>0 to 2</td>
<td>1 to 3</td>
<td>0 to 2</td>
<td>1 to 3</td>
<td>skip</td>
</tr>
<tr>
<td>2b</td>
<td>2 to 0</td>
<td>3 to 1</td>
<td>2 to 0</td>
<td>3 to 1</td>
<td>skip</td>
</tr>
<tr>
<td>3a</td>
<td>0 to 3</td>
<td>1 to 2</td>
<td>1 to 2</td>
<td>0 to 3</td>
<td>skip</td>
</tr>
<tr>
<td>3b</td>
<td>3 to 4</td>
<td>2 to 1</td>
<td>2 to 1</td>
<td>3 to 0</td>
<td>skip</td>
</tr>
<tr>
<td>4a</td>
<td>0 to 4</td>
<td>skip</td>
<td>skip</td>
<td>skip</td>
<td>0 to 4</td>
</tr>
<tr>
<td>4b</td>
<td>4 to 0</td>
<td>skip</td>
<td>skip</td>
<td>skip</td>
<td>4 to 0</td>
</tr>
<tr>
<td>5a</td>
<td>skip</td>
<td>1 to 4</td>
<td>skip</td>
<td>skip</td>
<td>1 to 4</td>
</tr>
<tr>
<td>5b</td>
<td>skip</td>
<td>4 to 1</td>
<td>skip</td>
<td>skip</td>
<td>4 to 1</td>
</tr>
<tr>
<td>6a</td>
<td>skip</td>
<td>skip</td>
<td>2 to 4</td>
<td>skip</td>
<td>2 to 4</td>
</tr>
<tr>
<td>6b</td>
<td>skip</td>
<td>skip</td>
<td>4 to 2</td>
<td>skip</td>
<td>4 to 2</td>
</tr>
<tr>
<td>7a</td>
<td>skip</td>
<td>skip</td>
<td>skip</td>
<td>3 to 4</td>
<td>3 to 4</td>
</tr>
<tr>
<td>7b</td>
<td>skip</td>
<td>skip</td>
<td>skip</td>
<td>4 to 3</td>
<td>4 to 3</td>
</tr>
</tbody>
</table>

• See source code 6 http://www.sdsc.edu/~tkaiser/source/f_ex6.f & c_ex6.c
Notes for examples

• Compiling and running on the SP
  
  mpxlf program.f
  mpxlf90 program.f
  mpcc program.c
  poe a.out -procs 3 -rmpool 1

• Compiling and running on the T3e
  
  f90 program.f
  cc program.c
  mpprun -n 3 a.out

• Suggested Exercise 1 : Hello World discussed above

• Suggested Exercise 2 : Write a program to send a token around a ring

• Suggested Exercise 3 : Write a program to send a derived data type

• Suggested Exercise 4 : Modify the myalltoall routine or sort routine to use probe