# MPI
# More of the Story

Timothy H. Kaiser, PH.D.
tkaiser2@nrel.gov

**Session 1: Introduction to MPI (this invitation)**
Monday, January 24th 11:00 AM - 12:00 PM Mountain

The first session will introduce MPI. We will give a background, show some sources of Documentation. We will show the classic "Hello world" program in MPI running on multiple processors. We will discuss Basic communications and show a simple send and receive program where messages are passed between processors. We will be running examples in the official languages supported by MPI, C and Fortran. Also, we will briefly discuss support for Python, R, and Java. Source code and scripts will be provided that can be run on Eagle.

**Session 2: Expansion to Higher-level MPI Calls**
Monday, January 31st 11:00 AM - 12:00 PM Mountain

The second session will expand on the first, showing many of the higher-level MPI calls commonly used to write parallel programs. Examples will be provided that can run on Eagle. We will look at: using the various predefined data types, broadcast, wildcards, asynchronous communications, and using probes and status information to control flow.

**Session 3: Additional Collective Operations**
Monday, February 7th 11:00 AM - 12:00 PM Mountain

In the third MPI session, we will look at additional collective operations including scatter, gather, and reductions. We'll also show examples of the "variable" versions of these calls where the amount of information shared is processor-dependent. We'll look at creating derived data types and managing subsets of processes using communicators.

**Session 4: Finite Difference Model**
Monday, February 14th 11:00 AM - 12:00 PM Mountain

In the fourth session, we will introduce a finite difference model that will demonstrate what a computational scientist needs to do to take advantage of computers using MPI. The model we are using is a two-dimensional finite-difference code. After discussing the serial code, we will show the modifications necessary to turn it into a parallel program using MPI. We will look at domain decomposition, initialization, data distribution, message passing, reduction operations, and multiple methods for data output. We will also look at the performance of the application on various numbers of processors to illustrate Amdahl's parallel program scaling law.

# Examples and Slides

Examples:

```
git clone https://github.com/timkphd/examples
cd examples/mpi
```

Slides:

https://github.com/timkphd/slides

# Outline

- Review
- Types
- Broadcast
- Wildcards
- Using Status and Probing
- Asynchronous Communication, first cut
- More Global communications
- Advanced topics
  - "V" operations
  - Derived types
  - Communicators

# 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

```
#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

| C MPI Types | |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | - |
| MPI_PACKED | - |

# Predefined types in Fortran

| Fortran MPI Types | |
|---|---|
| MPI_INTEGER | INTEGER |
| MPI_REAL | REAL |
| MPI_DOUBLE PRECISION | DOUBLE PRECISION |
| MPI_COMPLEX | COMPLEX |
| MPI_LOGICAL | LOGICAL |
| MPI_CHARACTER | CHARACTER(1) |
| MPI_BYTE | - |
| MPI_PACKED | - |

# 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

| c_ex04.c.c | f_ex04.f90 |
| --- | --- |

# Wildcards

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

```
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

| c_ex10.c.c | f_ex10.f90 |
|---|---|

# 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)`

| c_ex10.c.c | f_ex10.f90 |
|---|---|

# 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

| c_ex02.c.c | f_ex02.f90 |
|---|---|

# 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_COM_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
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)

| c_ex10.c.c | f_ex10.f90 |
|---|---|

# 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 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_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
```

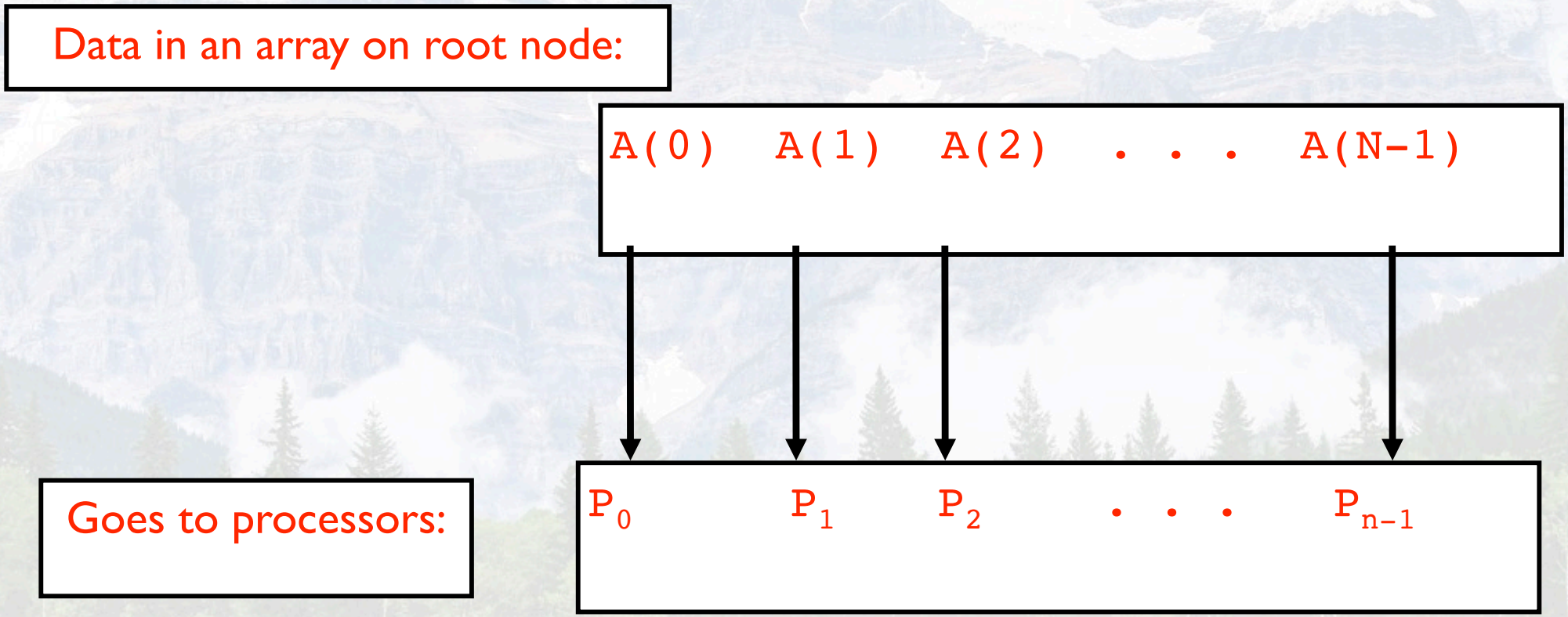
# Exercise 3 : Asynchronous Send and Receive

- 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

| c_ex03.c.c | f_ex03.f90 |
|---|---|

# Scatter Operation using MPI_Scatter

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

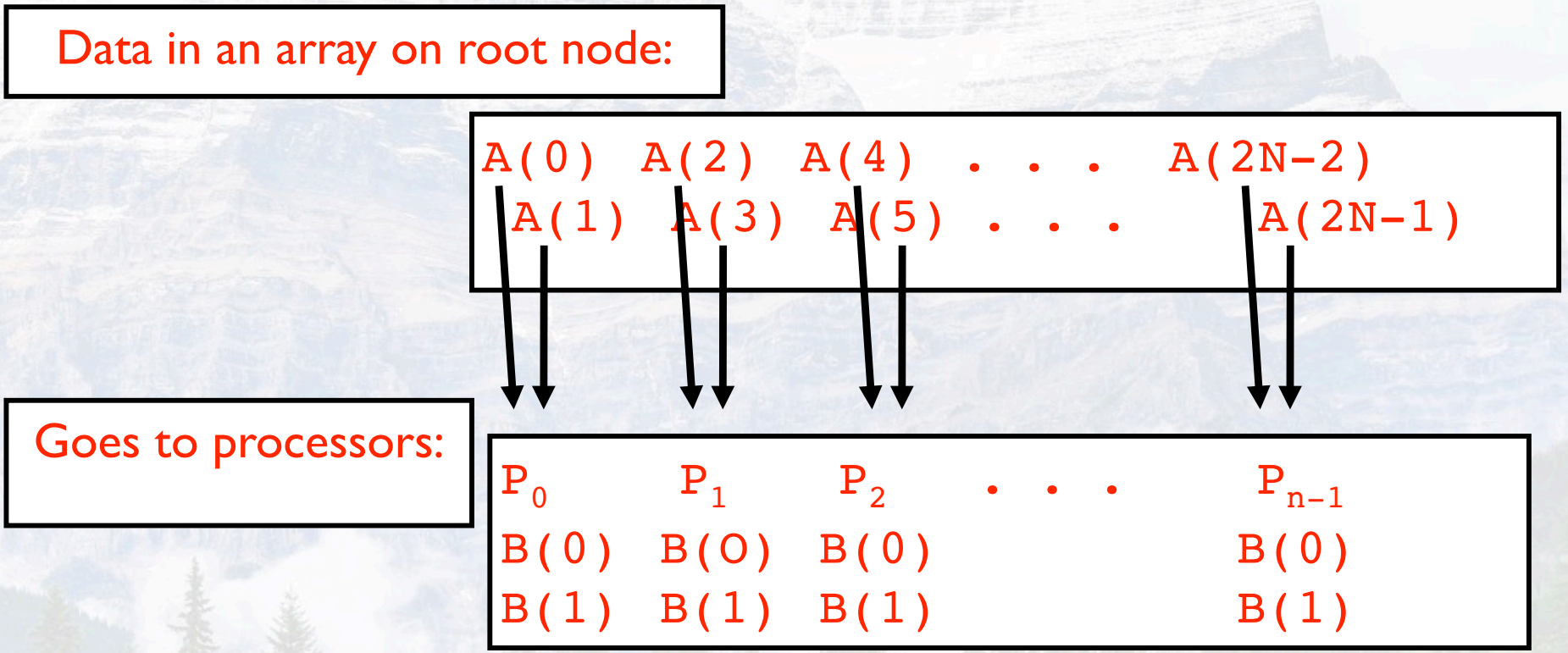


# MPI_Scatter

- 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

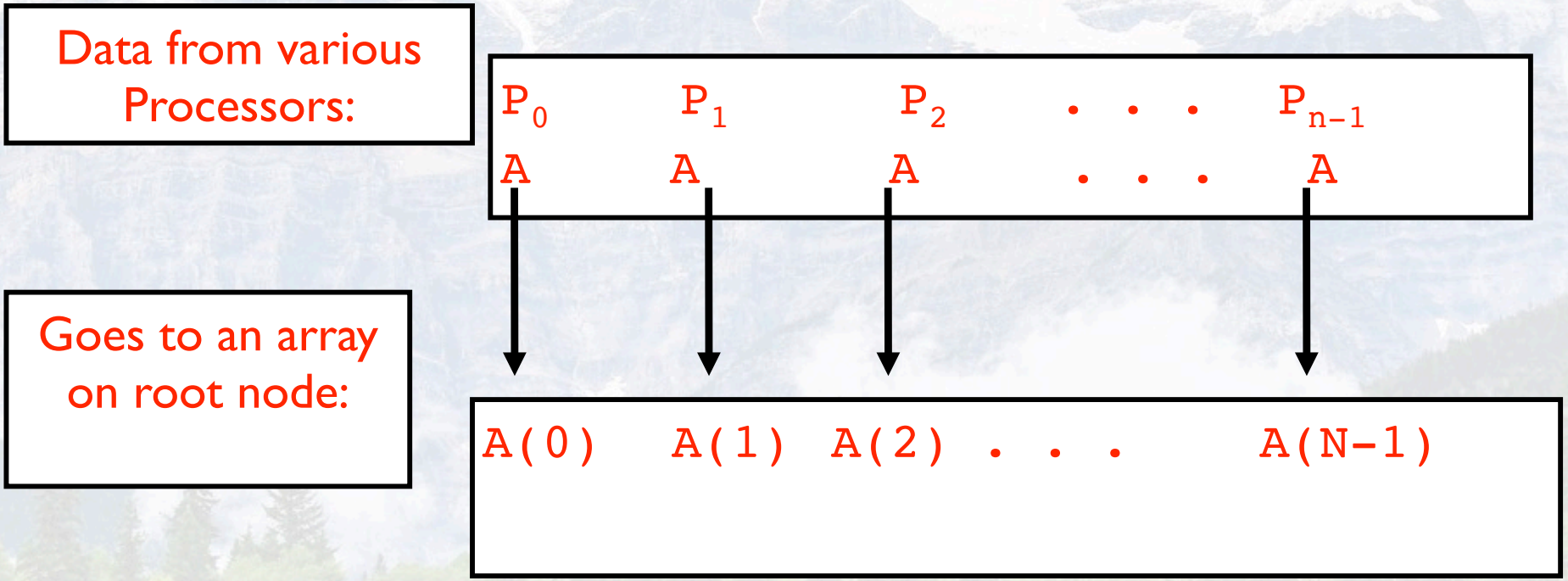
# Scatter Operation using MPI_Scatter

- Scatter with Sendcnts = 2



# 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



# 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

| c_ex05.c.c | f_ex05.f90 |
| --- | --- |

# 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)
```

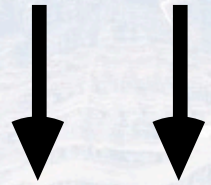
# Exercise 6 : Global Sum with MPI_Reduce

- Write a program to sum data from all processors

# Global Sum with MPI_Reduce

2d array spread across processors

| | X(0) | X(1) | X(2) |
|---|---|---|---|
| NODE 0 | A0 | B0 | C0 |
| NODE 1 | A1 | B1 | C1 |
| NODE 2 | A2 | B2 | C2 |



| | X(0) | X(1) | X(2) |
|---|---|---|---|
| NODE 0 | A0+A1+A2 | B0+B1+B2 | C0+C1+C2 |
| NODE 1 | | | |
| NODE 2 | | | |

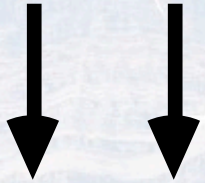
# All Gather and All Reduce

- 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

| | X(0) | X(1) | X(2) |
|---|---|---|---|
| NODE 0 | A0 | B0 | C0 |
| NODE 1 | A1 | B1 | C1 |
| NODE 2 | A2 | B2 | C2 |



| | Y(0) | Y(1) | Y(2) |
|---|---|---|---|
| NODE 0 | A0+A1+A2 | B0+B1+B2 | C0+C1+C2 |
| NODE 1 | A0+A1+A2 | B0+B1+B2 | C0+C1+C2 |
| NODE 2 | A0+A1+A2 | B0+B1+B2 | C0+C1+C2 |

| c_ex06.c.c | f_ex06.f90 |
|---|---|

# 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 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

| c_ex07.c.c | f_ex07.f90 |
|---|---|

# Things Left

- "V" operations
- Communicators
- Derived typed
- Some nontrivial examples
  - Pass a token with a new communicator
  - Implementation of all to all communication
  - Bag of Tasks

# The dreaded "V" or variable or 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_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 and MPI_Alltoallv

# MPI_Gatherv

- C
  - `int MPI_Gatherv (&sendbuf, sendcnts, sendtype, &recvbuf, &recvcnts, &rdispls,recvtype,root,comm);`
- Fortran
  - `MPI_Gatherv (sendbuf, sendcnts, sendtype, recvbuf, recvcnts, 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 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.

# 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=…
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_Alltoallv example

- Each processor  send/rec a different and random amount of data to/from otherprocessors.
- We use MPI_Alltoall first to tell how much data is going to be sent.

| c_ex08.c.c | f_ex08.f90 |
| --- | --- |

# 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)`

# 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)`**

| c_ex11.c.c | f_ex11.f90 |
|---|---|

# 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

- 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

# 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_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
```

| c_ex12.c.c | f_ex12.f90 |
|---|---|

# MPI_Comm_split example output

- 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
```

# 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(old_comm, group, &newcomm)`
- Fortran
  - `Call 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
  - `int MPI_Group_incl(group, n,&ranks,&new_group)`
- 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_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)**

| c_ex10.c.c | f_ex10.f90 |
|---|---|

# Pass Token

- c_ex10.c & f_ex10.f90
- Set up a new communicator
  - All but one task is in the new communicator
  - These tasks pass a token
- Remaining task reads data and
- Injects it into the "token" communicator via MPI_COM_WORLD

| c_ex11.c.c | f_ex11.f90 |
| --- | --- |

# My All to All Personalized Communication

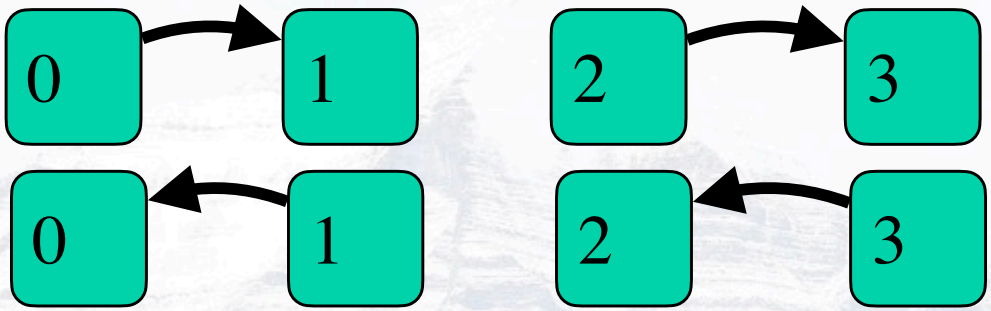
- testall2all.f90
- My algorithm is based on a Hypercube algorithm
  - Does not require power of 2 processors
    - Base algorithm assumes power of 2 processors
    - Check to see if you are sending to a valid processor
  - Uses simple trick to avoid nonblocking send/receive
    - Lower processor sends first
      - If Myid < partner send first
      - If Myid > partner recv first

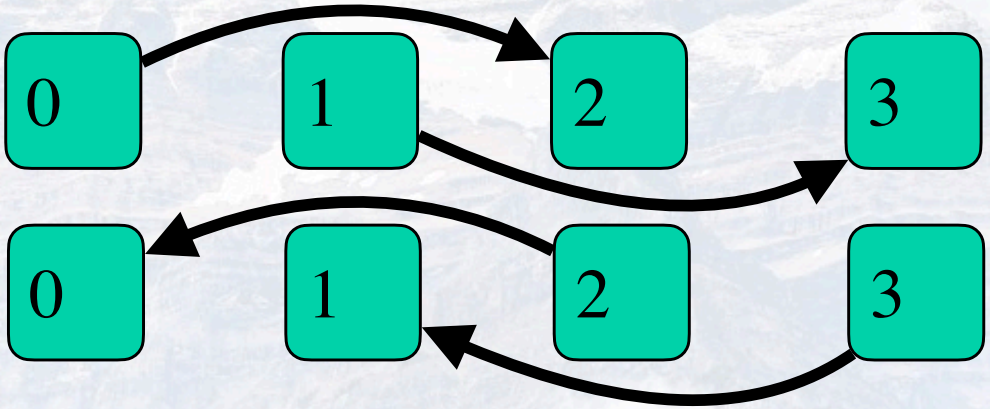
`testall2all.f90`

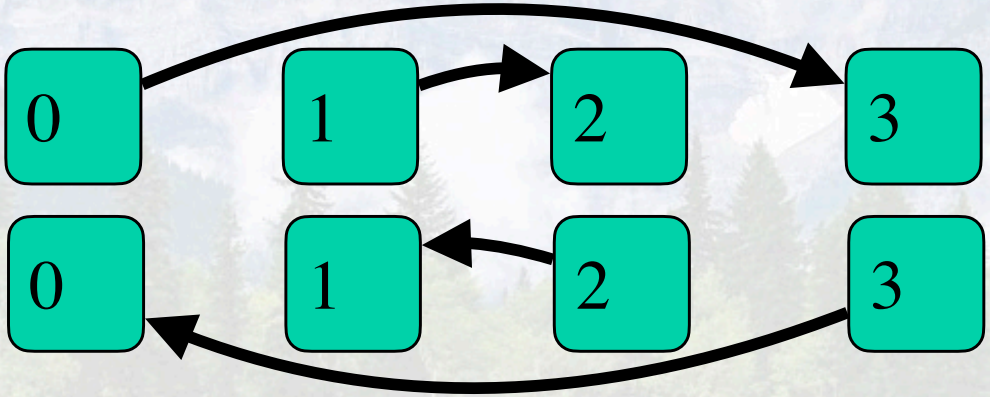
# My ALLtoALLv Algorithm

```
### find n2, the power of two >= numnodes
do i=1,n2-1
   ### do xor to find the processor xchng
   xchng=xor(i,myid)
   ### are we sending to a valid processor, else skip
    if(xchng <= (numnodes-1))then
       ### lower processor sends first
       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
```

# Algorithm with 4 nodes



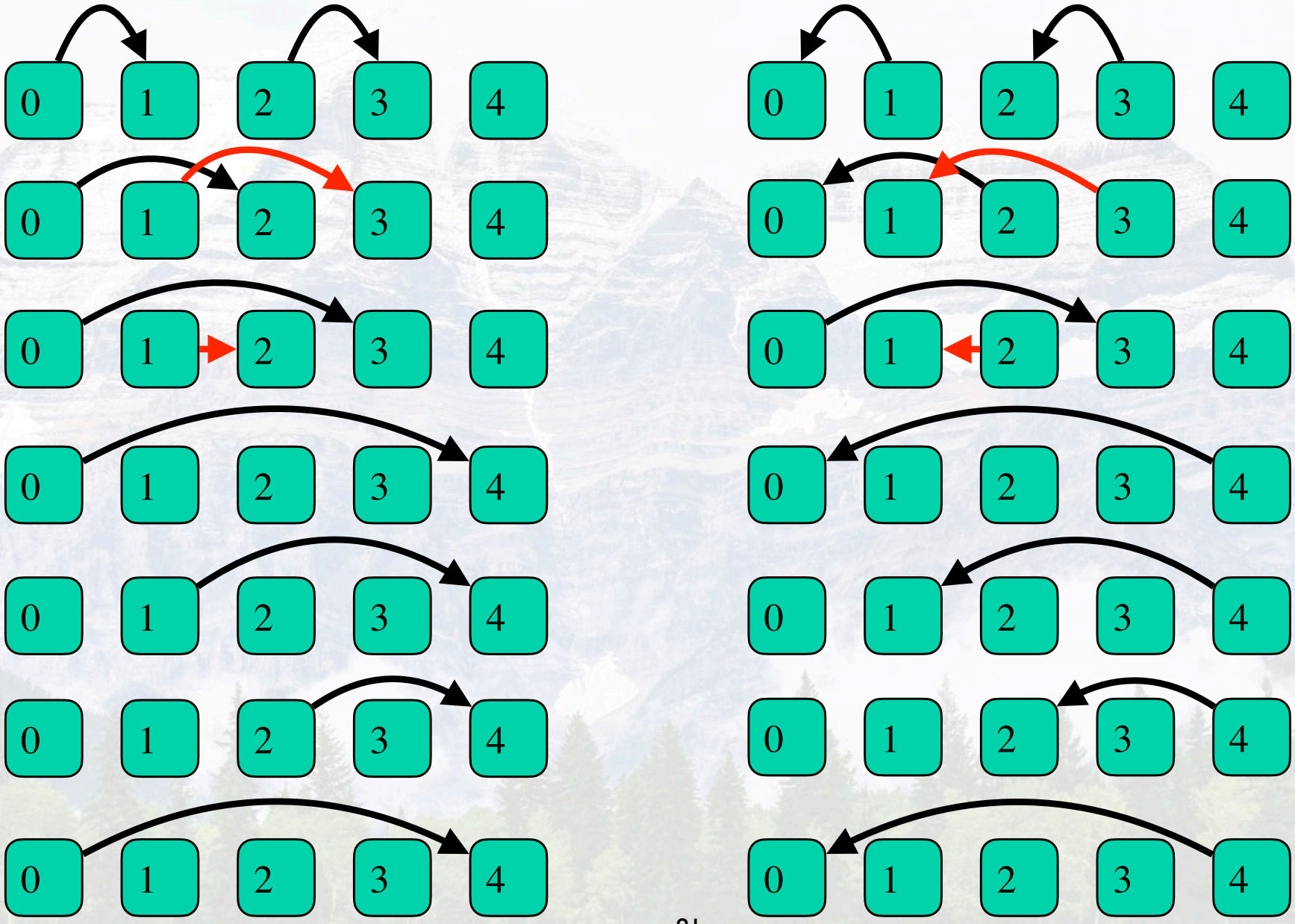



# Algorithm with 4 nodes

| Stage | Node 0 | Node 1 | Node 2 | Node 3 |
|---|---|---|---|---|
| 1a | 0 to 1 | 0 to 1 | 2 to 3 | 3 to 2 |
| 1b | 1 to 0 | 1 to 0 | 3 to 2 | 2 to 3 |
| 2a | 0 to 2 | 1 to 3 | 0 to 2 | 1 to 3 |
| 2b | 2 to 0 | 3 to 1 | 2 to 0 | 3 to 1 |
| 3a | 0 to 3 | 1 to 2 | 1 to 2 | 0 to 3 |
| 3b | 3 to 0 | 2 to 1 | 2 to 1 | 3 to 0 |

# Algorithm with 5 nodes



# Algorithm with 5 nodes

| Stage | Node 0 | Node 1 | Node 2 | Node 3 | Node 4 |
|---|---|---|---|---|---|
| 1a | 0 to 1 | 0 to 1 | 2 to 3 | 3 to 2 | skip |
| 1b | 1 to 0 | 1 to 0 | 3 to 2 | 2 to 3 | skip |
| 2a | 0 to 2 | 1 to 3 | 0 to 2 | 1 to 3 | skip |
| 2b | 2 to 0 | 3 to 1 | 2 to 0 | 3 to 1 | skip |
| 3a | 0 to 3 | 1 to 2 | 1 to 2 | 0 to 3 | skip |
| 3b | 3 to 0 | 2 to 1 | 2 to 1 | 3 to 0 | skip |
| 4a | 0 to 4 | skip | skip | skip | 0 to 4 |
| 4b | 4 to 0 | skip | skip | skip | 4 to 0 |
| 5a | skip | 1 to 4 | skip | skip | 1 to 4 |
| 5b | skip | 4 to 1 | skip | skip | 4 to 1 |
| 6a | skip | skip | 2 to 4 | skip | 2 to 4 |
| 6b | skip | skip | 4 to 2 | skip | 4 to 2 |
| 7a | skip | skip | skip | 3 to 4 | 3 to 4 |
| 7b | skip | skip | skip | 4 to 3 | 4 to 3 |

# Python code to create previous chart

```
from math import log2
numnodes=3
procs=list(range(0,numnodes))


#find n2 such that it is a power of 2**n2 is equal to or greater than numnodes
j=round(math.log2(numnodes))
n2=int(log2(numnodes))
if((2**n2) < numnodes) : n2=n2+1
n2=2**n2
#subtract 1 because nodes are numbered from 0 to n-1
n2=n2-1
print(n2)
```

```
3
```

```
for myid in procs:
    print("myid=",myid)
    for i in range(1,n2+1):
        xchng=i^myid
        if xchng <= numnodes-1:
            print("stage ",i," task ",myid," exchanges with ",xchng)
        else:
            print("stage ",i," task ",myid," skips ",xchng)
```

```
myid= 0
stage  1  task  0  exchanges with  1
stage  2  task  0  exchanges with  2
stage  3  task  0  skips  3
myid= 1
stage  1  task  1  exchanges with  0
stage  2  task  1  skips  3
stage  3  task  1  exchanges with  2
myid= 2
stage  1  task  2  skips  3
stage  2  task  2  exchanges with  0
stage  3  task  2  exchanges with  1
```