Message Passing with MPI

Graham E Fagg
CS 594 Spring 2006

Notes

- This talk is a combination of lots of different material from a host of sources including:
  - David Cronk & David Walker
  - EPCC
  - NCSA
  - LAM and MPICH teams

Introduction to MPI

- What is MPI?
  - MPI stands for “Message Passing Interface”
  - In ancient times (late 1980’s early 1990’s) each vender had its own message passing library
    - Non-portable code
    - Not enough people doing parallel computing due to lack of standards

What is MPI?

- April 1992 was the beginning of the MPI forum
  - Organized at SC92
  - Consisted of hardware vendors, software vendors, academicians, and end users
  - Held 2 day meetings every 6 weeks
  - Created drafts of the MPI standard
  - This standard was to include all the functionality believed to be needed to make the message passing model a success
  - Final version released may, 1994

What is MPI?

- A standard library specification!
  - Defines syntax and semantics of an extended message passing model
  - It is not a language or compiler specification
  - It is not a specific implementation
  - It does not give implementation specifics
    - Hints are offered, but implementers are free to do things however they want
    - Different implementations may do the same thing in a very different manner
  - http://www.mpi-forum.org

What is MPI

- A library specification designed to support parallel computing in a distributed memory environment
  - Routines for cooperative message passing
    - There is a sender and a receiver
    - Point-to-point communication
    - Collective communication
  - Routines for synchronization
  - Derived data types for non-contiguous data access patterns
  - Ability to create sub-sets of processors
  - Ability to create process topologies

What is MPI?

- Continuing to grow!
  - New routines have been added to replace old routines
  - New functionality has been added
    - Dynamic process management
    - One sided communication
    - Parallel I/O

Getting Started with MPI

- Outline
  - Introduction
  - 6 basic functions
  - Basic program structure
  - Groups and communicators
  - A very simple program
  - Message passing
  - A simple message passing example
  - Types of programs
    - Traditional
    - Master/Slave
    - Examples
  - Unsafe communication

Getting Started with MPI

- MPI contains 128 routines (more with the extensions)!
- Many programs can be written with just 6 MPI routines!
- Upon startup, all processes can be identified by their ***rank***, which goes from 0 to N-1 where there are N processes

6 Basic Functions

- MPI_INIT: Initialize MPI
- MPI_Finalize: Finalize MPI
- MPI_COMM_SIZE: How many processes are running?
- MPI_COMM_RANK: What is my process number?
- MPI_SEND: Send a message
- MPI_RECV: Receive a message

MPI_INIT (ierr)

- ierr: Integer error return value. 0 on success, non-zero on failure.
- This **MUST** be the first MPI routine called in any program.
  - Except for MPI_Initialized ( ) can be called to check if MPI_Init has been called!!
- Can only be called once
- Sets up the environment to enable message passing

MPI_FINALIZE (ierr)

- ierr: Integer error return value. 0 on success, non-zero on failure.
- This routine must be called by each process before it exits
- This call cleans up all MPI state
- No other MPI routines may be called after MPI_FINALIZE
- All pending communication must be completed (locally) before a call to MPI_FINALIZE

Basic Program Structure

```
program main                    #include “mpi.h”
include ‘mpi.h’
integer ierr                    int main ()
                                {
call MPI_INIT (ierr)            MPI_Init ()
……                              ……
Do some work                    Do some work
……                              ……
call MPI_FINALIZE (ierr)        MPI_Finalize ()
Maybe do some additional        Maybe do some additional
Local computation               Local computation
…….                             …….
end                             }
```

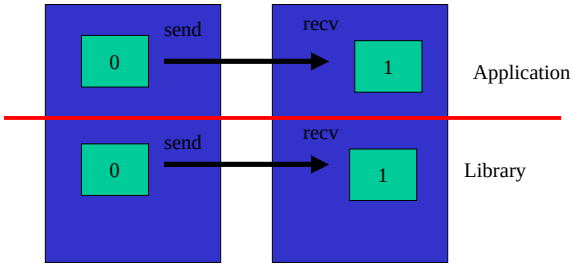
Groups and communicators

- Communicators are containers that hold messages and groups of processes together with additional meta-data
- All messages are passed only within communicators
- Upon startup, there is a single set of processes associated with the communicator MPI_COMM_WORLD
- Groups can be created which are sub-sets of this original group, also associated with communicators

Groups and communicators

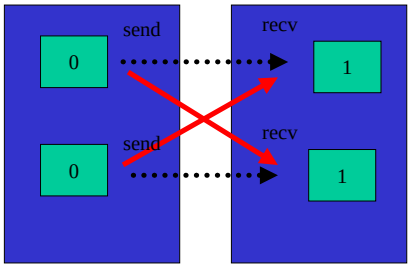
- Why do communicators exist
  - To keep different message passing libraries from interfering with each other
  - Allows the building of multiple layers of message passing code

Groups and communicators



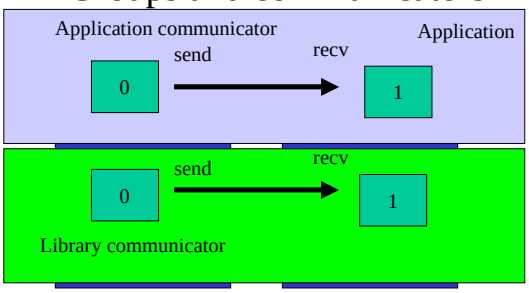
Library used by application that uses Message passing

Groups and communicators



Nothing to stop message passing to the wrong layer…..

Groups and communicators



Library used by application that uses Message passing

MPI_COMM_RANK (comm, rank, ierr)

- comm: Integer communicator.
- rank: Returned rank of calling process
- ierr: Integer error return code

- This routine returns the relative rank of the calling process, within the group associated with comm.

MPI_COMM_SIZE (comm, size, ierr)

- Comm: Integer communicator identifier
- Size: Upon return, the number of processes in the group associated with comm. For our purposes, always the total number of processes

- This routine returns the number of processes in the group associated with comm

A Very Simple Program Hello World

```
program main
include 'mpi.h'
integer ierr, size, rank

call MPI_INIT (ierr)
call MPI_COMM_RANK (MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, size, ierr)
print *, 'Hello World from process', rank, 'of', size
call MPI_FINALIZE (ierr)
end
```

Hello World

```
> mpirun –np 4 a.out
>
> Hello World from 2 of 4
> Hello World from 0 of 4
> Hello World from 3 of 4
> Hello World from 1 of 4
```

```
> mpirun –np 4 a.out
>
> Hello World from 3 of 4
> Hello World from 1 of 4
> Hello World from 2 of 4
> Hello World from 0 of 4
```

Message Passing

- Message passing is the transfer of data from one process to another
  - This transfer requires cooperation of the sender and the receiver, but is initiated by the sender
  - There must be a way to "describe" the data
  - There must be a way to identify specific processes
  - There must be a way to identify messages

Message Passing

- Data is described by a triple
  1. Address: Where is the data stored
  2. Count: How many elements make up the message
  3. Datatype: What is the type of the data
     - Basic types (integers, reals, etc)
     - Derived types (good for non-contiguous data access)

Message Passing

- Processes are specified by a double
  1. Communicator: safe space to pass message
  2. Rank: The relative rank of the specified process within the group associated with the communicator
- Messages are identified by a single tag
  - This can be used to differentiate between different types of messages
    - Max tag can be looked up but must be atleast 32k

MPI_SEND(buf, cnt, dtype, dest, tag, comm, ierr)

- buf: The address of the beginning of the data to be sent
- cnt: The number of elements to be sent
- dtype: datatype of each element
- dest: The rank of the destination
- tag: The message tag
- comm: The communicator

MPI_SEND

- Once this routine returns, the message has been copied out of the user buffer and the buffer can be reused
- This may require the use of system buffers. If there are insufficient system buffers, this routine will block until a corresponding receive call has been posted
- Completion of this routine indicates nothing about the designated receiver

MPI_RECV (buf, cnt, dtype, source, tag, comm, status, ierr)

- buf: Starting address of receive buffer
- cnt: Max number of elements to receive
- dtype: Datatype of each element
- source: Rank of sender (may use MPI_ANY_SOURCE)
- tag: The message tag (may use MPI_ANY_TAG)
- comm: Communicator
- status: Status information on the received message

MPI_RECV

- When this call returns, the data has been copied into the user buffer
- Receiving fewer than *cnt* elements is ok, but receiving more is an error
- Status is a structure in C (MPI_Status) and an array in Fortran (integer status(MPI_STATUS_SIZE))

MPI_STATUS

- The status parameter is used to retrieve information about a completed receive
- In C, status is a structure consisting of at least 3 fields: MPI_SOURCE, MPI_TAG, MPI_ERROR
- status.MPI_SOURCE, status.MPI_TAG, and status.MPI_ERROR contain the source, tag, and error code, respectively
- In Fortran, status must be an integer array of size MPI_STATUS_SIZE
- status(MPI_SOURCE), status(MPI_TAG), and status(MPI_ERROR) contain the source, tag, and error code

Send/Recv Example

```
program main
include 'mpi.h'
CHARACTER*20 msg
integer ierr, rank, tag, status (MPI_STATUS_SIZE)

tag = 99
call MPI_INIT (ierr)
call MPI_COMM_RANK (MPI_COMM_WORLD, rank, ierr)
if (myrank .eq. 0) then
  msg = "Hello there"
  call MPI_SEND (msg, 11, MPI_CHARACTER, 1, tag, &
                 MPI_COMM_WORLD, ierr)
else if (myrank .eq. 1) then
  call MPI_RECV(msg, 20, MPI_CHARACTER, 0, tag, &
                 MPI_COMM_WORLD, status, ierr)
endif
call MPI_FINALIZE (ierr)
end
```

Types of MPI Programs

- Traditional
  - Break the problem up into about even sized parts and distribute across all processors
  - What if problem is such that you cannot tell how much work must be done on each part?
- Master/Slave
  - Break the problem up into many more parts than there are processors
  - Master sends work to slaves
  - Parts may be all the same size or the size may vary

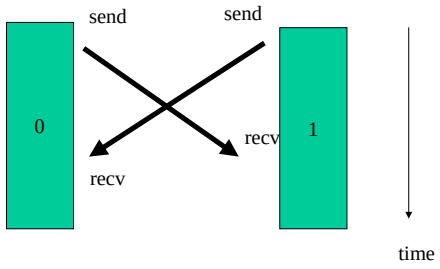
Traditional Example

Compute the sum of a large array of N integers

```
Comm = MPI_COMM_WORLD
Call MPI_COMM_RANK (comm, rank)
Call MPI_COMM_SIZE (comm, npes)
Stride = N/npes
Start = (stride * rank) + 1
Sum = 0
DO (I = start, start+stride)
  sum = sum + array(I)
ENDDO
```

```
If (rank .eq. 0) then
 DO (I = 1, npes-1)
   call MPI_RECV(tmp, 1, MPI_INTEGER,
&          I, 2, comm, status)
   sum = sum + tmp
 ENDDO
ELSE
 MPI_SEND (sum, 1, MPI_INTEGER, &
&    0, 2 comm)
ENDIF
```

Unsafe Communication Patterns



Unsafe Communication Patterns

- Process 0 and process 1 must exchange data
- Process 0 sends data to process 1 and then receives data from process 1
- Process 1 sends data to process 0 and then receives data from process 0
- If there is not enough system buffer space for either message, this will deadlock
- Any communication pattern that relies on system buffers is unsafe
- Any pattern that includes a cycle of blocking sends is unsafe

Communication Modes

- Outline
  - Standard mode
    - Blocking
    - Non-blocking
  - Non-standard mode
    - Buffered
    - Synchronous
    - Ready
  - Performance issues

Point-to-Point Communication Modes

- Standard Mode:
  - blocking:
    - MPI_SEND (buf, count, datatype, dest, tag, comm)
    - MPI_RECV (buf, count, datatype, source, tag, comm, status)
      - Generally **ONLY** use if you cannot call earlier **AND** there is no other work that can be done!
      - Standard **ONLY** states that buffers can be used once calls return. It is implementation dependant on when blocking calls return.
      - Blocking sends **MAY** block until a matching receive is posted. This is not required behavior, but the standard does not prohibit this behavior either. Further, a blocking send may have to wait for system resources such as system managed message buffers.
  - Be VERY careful of deadlock when using blocking calls!

Point-to-Point Communication Modes (cont)

- Standard Mode:
  - Non-blocking (immediate) sends/receives:
    - MPI_ISEND (buf, count, datatype, dest, tag, comm, request)
    - MPI_IRECV (buf, count, datatype, source, tag, comm, request)
    - MPI_WAIT (request, status)
    - MPI_TEST (request, flag, status)
      - Allows communication calls to be posted early, which may improve performance.
        - » Overlap computation and communication
        - » Latency tolerance
        - » Less (or no) buffering
    - * MUST either complete these calls (with wait or test) or call MPI_REQUEST_FREE

MPI_ISEND (buf, cnt, dtype, dest, tag, comm, request)

- Same syntax as MPI_SEND with the addition of a request handle
- Request is a handle (int in Fortran) used to check for completeness of the send
- This call returns immediately
- Data in buf may not be accessed until the user has completed the send operation
- The send is completed by a successful call to MPI_TEST or a call to MPI_WAIT

MPI_IRECV(buf, cnt, dtype, source, tag, comm, request)

- Same syntax as MPI_RECV except status is replaced with a request handle
- Request is a handle (int in Fortran) used to check for completeness of the recv
- This call returns immediately
- Data in buf may not be accessed until the user has completed the receive operation
- The receive is completed by a successful call to MPI_TEST or a call to MPI_WAIT

MPI_WAIT (request, status)

- Request is the handle returned by the non-blocking send or receive call
- Upon return, status holds source, tag, and error code information
- This call does not return until the non-blocking call referenced by *request* has completed
- Upon return, the request handle is freed
- If *request* was returned by a call to MPI_ISEND, return of this call indicates nothing about the destination process

MPI_TEST (request, flag, status)

- *Request* is a handle returned by a non-blocking send or receive call
- Upon return, *flag* will have been set to true if the associated non-blocking call has completed. Otherwise it is set to false
- If *flag* returns true, the request handle is freed and *status* contains source, tag, and error code information
- If *request* was returned by a call to MPI_ISEND, return with flag set to true indicates nothing about the destination process
  - Why?

Non-blocking Communication

- Simple example
  - Process wants to exchange data will all others
  - (later on we will show how to use a collective to do this)

Non-blocking Communication

```
for (I=0;I<size;I++) {
  if (I!=myrank)
        ierr = MPI_Send (data, cnt, dtype,
          I, tag, comm)
}

for (I=0;I<size;I++)  {
  if (I!=myrank)
        ierr = MPI_RECV (data, cnt, dtype,
          I, tag, comm, &status)
}
```

Clearly unsafe,
Why ?

```
for (I=0;I<size;I++) {
  if (myrank!=I)
        ierr = MPI_Isend (data, cnt, dtype,
          I, tag, comm, &request)
}

for (I=0;I<size;I++) {
  if (myrank!=I)
        ierr = MPI_Recv (data, cnt, dtype,
          I, tag, comm, &status)
}
```

May run out of handles
Why ?

Non-blocking Communication

```
for (I=0;I<size;I++) {
  if (myrank!=I)
     ierr = MPI_Isend (data, cnt, dtype,
          I, tag, comm, &request(I))
}

for (I=0;I<size;I++) {
  if (myrank!=I)
     ierr = MPI_Wait (request(I), &status)
}

      ……..
      receive data
      ……..
```

Unsafe
why?

```
for (I=0;I<size;I++) {
  if (myrank!=I)
     ierr = MPI_Isend (data, cnt, dtype,
          I, tag, comm, &request(I))
}

      ……..
      receive data
      ……..


for (I=0;I<size;I++) {
  if (myrank!=I)
     ierr = MPI_Wait (request(I), &status)
}
```

Safe…….
But how could you do it better and why ?

Non-blocking communication

```
for (I=0;I<size;I++) {
  if (myrank!=I)
     ierr = MPI_Isend (data, cnt, dtype,
          I, tag, comm, &srequest(I))
}

for (I=0;I<size;I++) {
  if (myrank!=I)
     ierr = MPI_Irecv (data, cnt, dtype,
          I, tag, comm, &rrequest(I))
}

MPI_Waitall (…)
```

Safe,
And [maybe] faster? Why?
What does it have to do with the MPI implementation or how many processes are involved and the network???

Point-to-Point Communication Modes (cont)

- Non-standard mode communication
  - Only used by the sender! (MPI uses the push communication model)
  - Buffered mode - A buffer must be provided by the application
  - Synchronous mode - Completes only after a matching receive has been posted
  - Ready mode - May only be called when a matching receive has already been posted

Point-to-Point Communication Modes: Buffered

- MPI_BSEND (buf, count, datatype, dest, tag, comm)
- MPI_IBSEND (buf, count, dtype, dest, tag, comm, req)
- MPI_BUFFER_ATTACH (buff, size)
- MPI_BUFFER_DETACH (buff, size)
  - Buffered sends do not rely on system buffers
  - The user supplies a buffer that **MUST** be large enough for all messages
  - User need not worry about calls blocking, waiting for system buffer space
  - The buffer is managed by MPI
  - The user **MUST** ensure there is no buffer overflow

Buffered Sends

```
#define BUFFSIZE 1000

char *buff;
char b1[500], b2[500]

MPI_Buffer_attach (buff, BUFFSIZE);
```

Seg violation

```
#define BUFFSIZE 1000

char *buff;
char b1[500], b2[500];

buff = (char*) malloc(BUFFSIZE);
MPI_Buffer_attach(buff, BUFFSIZE);
MPI_Bsend(b1, 500, MPI_CHAR, 1, 1,
      MPI_COMM_WORLD);
MPI_Bsend(b2, 500, MPI_CHAR, 2, 1,
      MPI_COMM_WORLD);
```

Buffer overflow

```
int size;
char *buff;
char b1[500], b2[500];

MPI_Pack_size (500, MPI_CHAR,
   MPI_COMM_WORLD, &size);
size += MPI_BSEND_OVERHEAD;
size = size * 2;
buff = (char*) malloc(size);
MPI_Buffer_attach(buff, size);
MPI_Bsend(b1, 500, MPI_CHAR, 1, 1,
      MPI_COMM_WORLD);
MPI_Bsend(b2, 500, MPI_CHAR, 2, 1,
      MPI_COMM_WORLD);
MPI_Buffer_detach (&buff, &size);
```

Safe

Point-to-Point Communication Modes: Synchronous

- MPI_SSEND (buf, count, datatype, dest, tag, comm)
- MPI_ISSEND (buf, count, dtype, dest, tag, comm, req)
  - Can be started (called) at any time.
  - Does not complete until a matching receive has been posted and the receive operation has been started
    - * Does NOT mean the matching receive has completed
  - Can be used in place of sending and receiving acknowledgements
  - Can be more efficient when used appropriately
    - buffering may be avoided

Point-to-Point Communication Modes: Ready Mode

- MPI_RSEND (buf, count, datatype, dest, tag, comm)
- MPI_IRSEND (buf, count, dtype, dest, tag, comm, req)
  - May ONLY be started (called) if a matching receive has already been posted.
  - If a matching receive has not been posted, the results are undefined
  - May be most efficient when appropriate
    - Removal of handshake operation
- Should only be used with **extreme** caution
- Only really faster on an Intel Paragon or a system that RDMA (pinned memory). Why ?

Ready Mode

MASTER

```
while (!done) {
  MPI_Recv (NULL, 0, MPI_INT, MPI_ANYSOURCE,
     1, MPI_COMM_WORLD, &status);
  source = status.MPI_SOURCE;
  get_work (…..);
  MPI_Rsend (buff, count, datatype, source, 2,
    MPI_COMM_WORLD);
  if (no more work) done = TRUE;
}
```

SLAVE

```
while (!done) {
  MPI_Send (NULL, 0, MPI_INT, MASTER,
    1, MPI_COMM_WORLD);
  MPI_Recv (buff, count, datatype, MASTER,
    2, MPI_COMM_WORLD, &status);
  …
}
```

UNSAFE

```
while (!done) {
  MPI_Irecv (buff, count, datatype, MASTER,
    2, MPI_COMM_WORLD, &request);
  MPI_Ssend (NULL, 0, MPI_INT, MASTER,
    1, MPI_COMM_WORLD);
  MPI_Wait (&request, &status);
…
}
```

SAFE

Point-to-Point Communication Modes: Performance Issues

- Non-blocking calls are almost always the way to go
  - Communication can be carried out during blocking system calls
  - Computation and communication can be overlapped if there is special purpose communication hardware
  - Less likely to have errors that lead to deadlock
  - Standard mode is usually sufficient - but buffered mode can offer advantages
    - Particularly if there are frequent, large messages being sent
    - If the user is unsure the system provides sufficient buffer space
  - Synchronous mode can be more efficient if acks are needed
    - Also tells the system that buffering is not required
- But, if no overlapping then non blocking is Slower due to extra data structures and book keeping!
  - Only way to know.. Benchmark it!

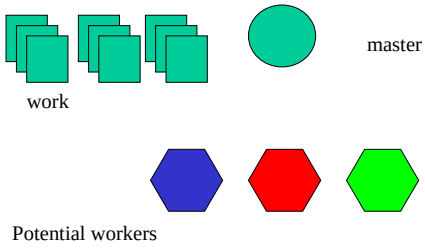
Point to Point summary

- Covered all basic communications
  - For here we can build all other communication patterns
    - Manually
    - May be slower than ‘collectives’ that can use special features of some MPP/SMPs.

Point 2 point case study

- Master has a large number of ‘tests’ that need to ran and some average result needs to be calculated.
- We will consider four things
  - Overall execution structure
  - What this means for message passing
  - Performance issue
  - Improving the structure for better performance

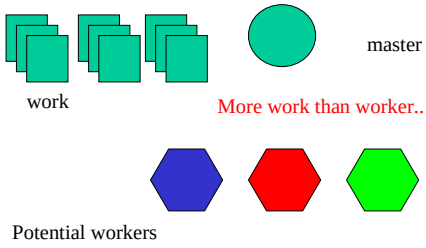
P2p example



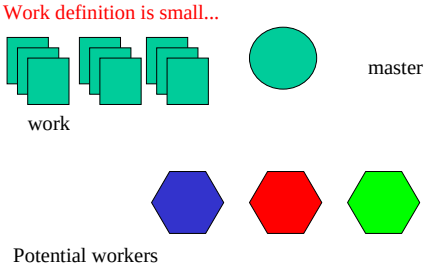
P2p case study

- What to consider
  - Do we have more work than workers?
  - How big is the work?
  - Is the work independent from each other
  - Are their intermediate results?
    - Do they need to get shared? Stored on disk?
  - Does the master need to do some work as well?

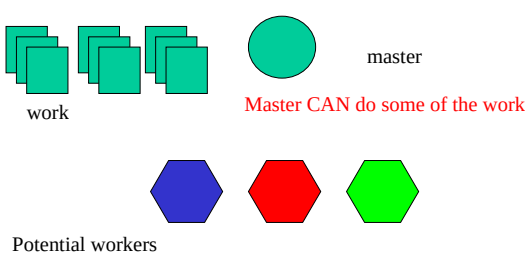
P2p case study



P2p case study

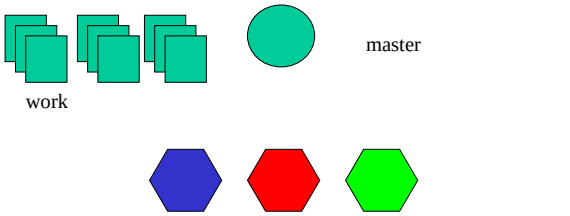


P2p case study



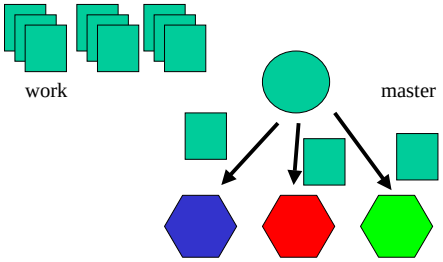
P2p case study

Work result needs to be at the master



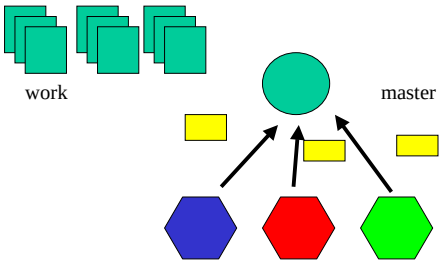
Potential workers

P2p case study



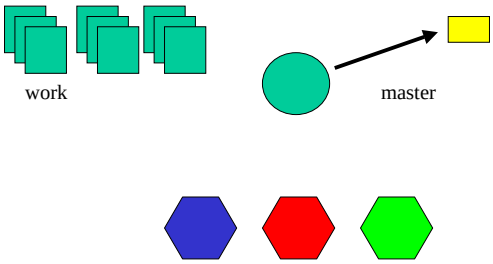
Master sends data to the workers : Slaves receive work

P2p case study



Works send partial results to the master : Master receiver partial results

P2p case study

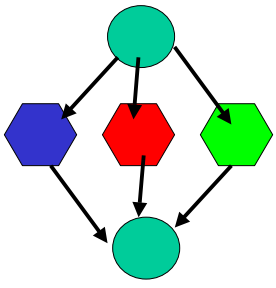


Master calculates the final answer

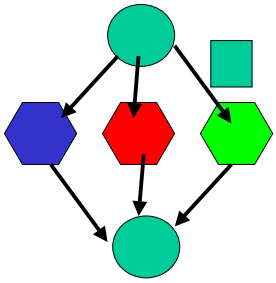
Point 2 point case study

- To consider how this looks as a parallel algorithm we need to draw it as a DAG

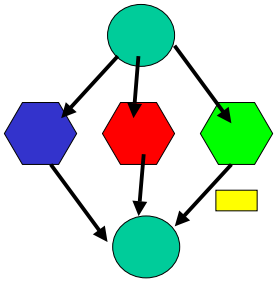
P2p case study



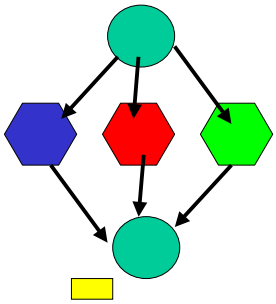
P2p case study



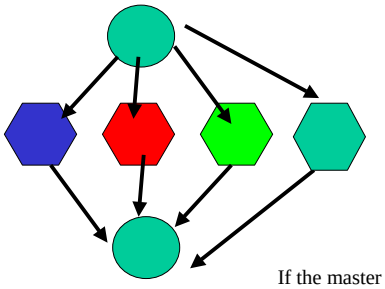
P2p case study



P2p case study

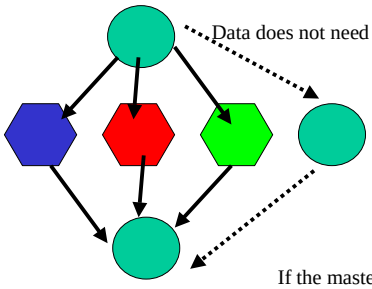


P2p case study



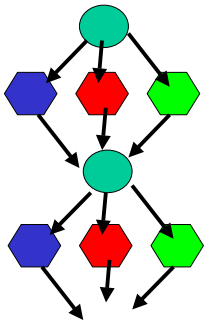
If the master
does work as well

P2p case study



If the master
does work as well

P2p case study



If you do not
distribute or
send out all
work in one go
we need to
loop until all the
work is done

P2p case study

- What does this look like in terms of code?
- The job of each process is defined by who they are (master or slave)
- Arcs in the graphs are data in the form of messages

P2p case study

- What does this look like in terms of code?
- The job of each process is defined by who they are (master or slave)
  - In MPI we can use RANK to define a master
  - RANK can also identify who the slaves are
- Arcs in the graphs are data in the form of messages
  - Depending on if your master or worker and which arc, we know if we are Sending to Receiving data

P2p case study

- Master

```
MPI_Init (..)

MPI_Comm_rank
    (MPI_COMM_WORLD, &rank)

If (rank==0) {
      /* I AM MASTER */
      Do_master ( )
      }
MPI_Finalize ( )
```

- Worker

```
MPI_Init (..)

MPI_Comm_rank
(MPI_COMM_WORLD, &rank);

If (rank!=0) {
     /* I am Worker */
     Do_worker ( rank )
}
MPI_Finalize ( )
```

P2p case study

- Do_master ( )

```
/* find out how many workers */
MPI_Comm_size (MCW, &size);
Workers=size-1;
Loop for each worker I
      Work [I] = dividework (I)

loop {
      MPI_Send (work[I],1, worker)
}
Loop
      MPI_Recv (result[I],1, worker..)
}
Do_calculate_result ()
Display_result ()
}
```

- Do_Worker (my id)
- {
- /* get work */
- MPI_Recv (work, 1, 0, status)
- PResult = Do_work (work)
- MPI_Send (work, 1, 0..)
- }

P2p case study

- To make the previous code work if the work does not divide up into the workers correctly you need to change the data being sent:
  - Special value for no-more work
  - you need to tell workers how much work they have
  - they can ask for work

P2p case study

- Special value for no-more work

```
Loop {
    MPI_Recv (work, 1, 0, .. Status)
    If (work==NOWORKLEFT) return ();
    Else
        Presult = Do_work ( ) …
        MPI_Send (Presult…)
    }
```

P2p case study

- you need to tell workers how much work they have

Master:

work has 3 pieces of work…

MPI_Send (howmuch, work..)

loop {

MPI_Send (work[I]…)

}

P2p case study

- you need to tell workers how much work they have

Slave:

do_work

MPI_recv (howmuch,…)

loop (1..howmuch)

MPI_Recv (work, 1, 0, .. Status)

Presult = Do_work ( ) …

MPI_Send (Presult…)

P2p case study

– they can ask for work

– Master

- If work-left or workers-still-working{

MPI_Recv (what&who..)

If what=result add it to partial result

If work-left MPI_Send (nextwork, who..)

Else MPI_Send (NOMOREWORK, who…)

}

P2p case study

– they can ask for work

– Worker

```
MPI_Send (Iwantsomework, 1, 0…)
Loop {
    MPI_Recv (work, 1, 0, .. Status)
    If (work==NOWORKLEFT) return ();
    Else
        Presult = Do_work ( ) …
        MPI_Send (Presult…)
    }
```

Collective Communication

- Outline
  - Introduction
  - Barriers
  - Broadcasts
  - Gather
  - Scatter
  - All gather
  - Alltoall
  - Reduction
  - Performance issues

Collective Communication

- Total amount of data sent must exactly match the total amount of data received
- Collective routines are collective across an entire communicator and must be called in the same order from all processors within the communicator
- Collective routines are all blocking
  - This simply means buffers can be re-used upon return
- Collective routines return as soon as the calling process' participation is complete
  - Does not say anything about the other processors
  - Collective routines may or may not be synchronizing
- No mixing of collective and point-to-point communication

Collective Communication

- Barrier: MPI_BARRIER (comm)
  - Only collective routine which provides explicit synchronization
  - Returns at any processor only after all processes have entered the call

Collective Communication

- Collective Communication Routines:
  - Except broadcast, each routine has 2 variants:
    - Standard variant: All messages are the same size
    - Vector Variant: Each item is a vector of possibly varying length
  - If there is a single origin or destination, it is referred to as the *root*
  - Each routine (except broadcast) has distinct send and receive arguments
  - Send and receive buffers must be disjoint
  - Each can use MPI_IN_PLACE, which allows the user to specify that data contributed by the caller is already in its final location.

Collective Communication: Bcast

- MPI_BCAST (buffer, count, datatype, root, comm)
  - Strictly in place
  - MPI-1 insists on using an intra-communicator
  - MPI-2 allows use of an inter-communicator
  - REMEMBER: A broadcast need not be synchronizing. Returning from a broadcast tells you nothing about the status of the other processes involved in a broadcast. Furthermore, though MPI does not require MPI_BCAST to be synchronizing, it neither prohibits synchronous behavior.

BCAST

```
If (myrank == root) {
  fp = fopen (filename, 'r');
  fscanf (fp, '%d', &iters);
  fclose (&fp);
  MPI_Bcast (&iters, 1, MPI_INT,
    root, MPI_COMM_WORLD);
}
else {
  MPI_Recv (&iters, 1, MPI_INT,
    root, tag, MPI_COMM_WORLD,
    &status);
}
```

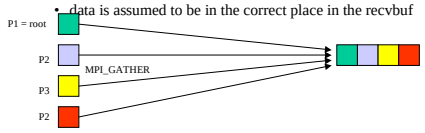
OOPS!

```
If (myrank == root) {
  fp = fopen (filename, 'r');
  fscanf (fp, '%d', &iters);
  fclose (&fp);
}
MPI_Bcast (&iters, 1, MPI_INT,
    root, MPI_COMM_WORLD);
cont
```

THAT'S BETTER

Collective Communication: Gather

- MPI_GATHER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
  - Receive arguments are only meaningful at the root
  - Each processor must send the same *amount* of data
  - Root can use MPI_IN_PLACE for sendbuf:
    - data is assumed to be in the correct place in the recvbuf



MPI_Gather

```
int tmp[20];
int res[320];
```

```
for (i = 0; i < 20; i++) {
  do some computation
  tmp[i] = some value;
}
MPI_Gather (tmp, 20, MPI_INT, res,
  20, MPI_INT, 0, MPI_COMM_WORLD);
if (myrank == 0)
  write out results
```

WORKS

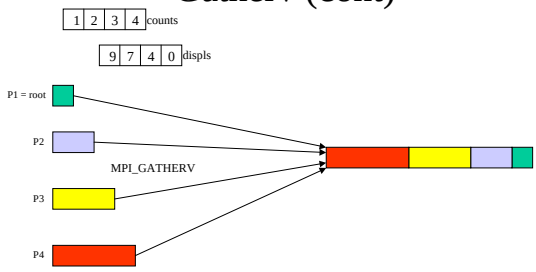
```
for (i = 0; i < 20; i++) {
  do some computation
  if (myrank == 0)
    res[i] = some value
  else tmp[i] = some value
}
if (myrank == 0)
  MPI_Gather (MPI_IN_PLACE,
    20, MPI_INT, RES, 20, MPI_INT,
    0, MPI_COMM_WORLD);
  write out results
else
  MPI_Gather (tmp, 20, MPI_INT,
    tmp, 320, MPI_REAL,
    MPI_COMM_WORLD);
```

A OK

Collective Communication: Gatherv

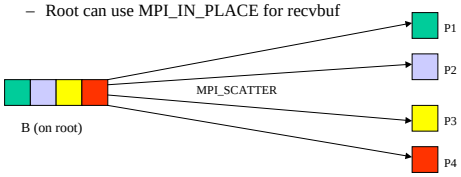
- MPI_GATHERV (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, root, comm)
  - Vector variant of MPI_GATHER
  - Allows a varying amount of data from each proc
  - allows root to specify where data from each proc goes
  - No portion of the receive buffer may be written more than once
  - MPI_IN_PLACE may be used by root.

Collective Communication: Gatherv (cont)



Collective Communication: Scatter

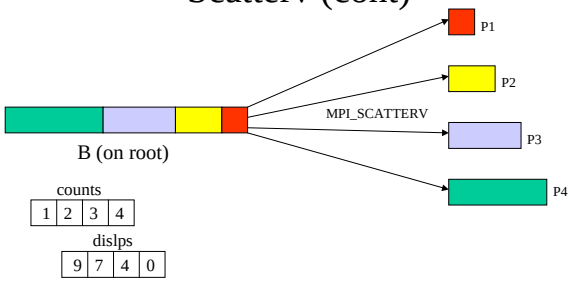
- MPI_SCATTER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)
  - Opposite of MPI_GATHER
  - Send arguments only meaningful at root
  - Root can use MPI_IN_PLACE for recvbuf



Collective Communication: Scatterv

- MPI_SCATTERV (sendbuf, scounts, displs, sendtype, recvbuf, recvcount, recvtype)
  - Opposite of MPI_GATHERV
  - Send arguments only meaningful at root
  - Root can use MPI_IN_PLACE for recvbuf
  - No location of the sendbuf can be read more than once

Collective Communication: Scatterv (cont)



Collective Communication: Allgather

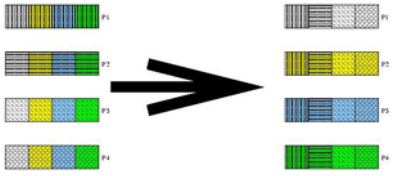
- MPI_ALLGATHER (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)
  - Same as MPI_GATHER, except all processors get the result
  - MPI_IN_PLACE may be used for sendbuf of all processors
  - Equivalent to a gather followed by a bcast

Collective Communication: Allgatherv

- MPI_ALLGATHERV (sendbuf, sendcount, sendtype, recvbuf, recvcounts, displs, recvtype, comm)
  - Same as MPI_GATHERV, except all processors get the result
  - MPI_IN_PLACE may be used for sendbuf of all processors
  - Equivalent to a gatherv followed by a bcast

Collective Communication: Alltoall (scatter/gather)

- MPI_ALLTOALL (sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, comm)



Collective Communication: Alltoallv

- MPI_ALLTOALLV (sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm)
  - Same as MPI_ALLTOALL, but the vector variant
    - Can specify how many blocks to send to each processor, location of blocks to send, how many blocks to receive from each processor, and where to place the received blocks

Collective Communication: Alltoallw

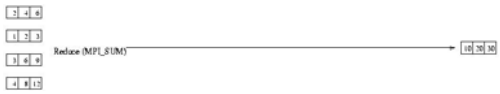
- MPI_ALLTOALLW (sendbuf, sendcounts, sdispls, sendtypes, recvbuf, recvcounts, rdispls, recvtypes, comm)
  - Same as MPI_ALLTOALLV, except different datatypes can be specified for data scattered as well as data gathered
    - Can specify how many blocks to send to each processor, location of blocks to send, how many blocks to receive from each processor, and where to place the received blocks
    - Displacements are now in terms of bytes rather that types

Collective Communication: Reduction

- Global reduction across all members of a group
- Can us predefined operations or user defined operations
- Can be used on single elements or arrays of elements
- Counts and types must be the same on all processors
- Operations are assumed to be associative
- User defined operations can be different on each processor, but not recommended

Collective Communication: Reduction (reduce)

- MPI_REDUCE (sendbuf, recvbuf, count, datatype, op, root, comm)
  - recvbuf only meaningful on root
  - Combines elements (on an element by element basis) in sendbuf according to *op*
  - Results of the reduction are returned to root in recvbuf
  - MPI_IN_PLACE can be used for sendbuf on root



Collective Communication: Reduction (cont)

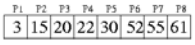
- MPI_ALLREDUCE (sendbuf, recvbuf, count, datatype, op, comm)
  - Same as MPI_REDUCE, except all processors get the result
- MPI_REDUCE_SCATTER (sendbuf, recv_buff, recvcounts, datatype, op, comm)
  - Acts like it does a reduce followed by a scatterv

Collective Communication: Prefix Reduction

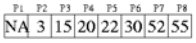
- MPI_SCAN (sendbuf, recvbuf, count, datatype, op, comm)
  - Performs an *inclusive* element-wise prefix reduction
- MPI_EXSCAN (sendbuf, recvbuf, count, datatype, op, comm)
  - Performs an *exclusive* prefix reduction
  - Results are undefined at process 0

MPI_SCAN

MPI_SCAN (sbuf, rbuf, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD)



MPI_EXSCAN (sbuf, rbuf, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD)



Collective Communication: Reduction - user defined ops

- MPI_OP_CREATE (function, commute, op)
  - if *commute* is true, operation is assumed to be commutative
  - Function is a user defined function with 4 arguments
    - invec: input vector
    - inoutvec: input and output value
    - len: number of elements
    - datatype: MPI_DATATYPE
    - Returns invec[i] *op* inoutvec[i], i = 0..len-1
- MPI_OP_FREE (op)

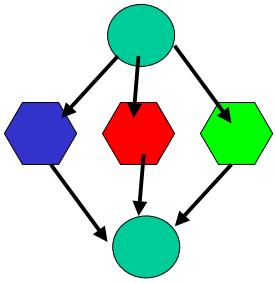
Collective Communication: Performance Issues

- Collective operations should have much better performance than simply sending messages directly
  - Broadcast may make use of a broadcast tree (or other mechanism)
  - All collective operations can potentially make use of a tree (or other) mechanism to improve performance
- Important to use the simplest collective operations which still achieve the needed results
- Use MPI_IN_PLACE whenever appropriate
  - Reduces unnecessary memory usage and redundant data movement

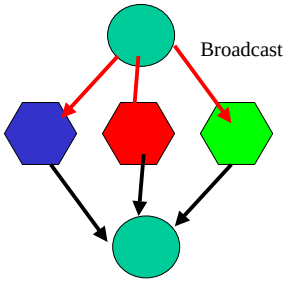
Case study again

- In the previous example we sent all the work out using point to point calls
- Received all the results using point to pint calls.
- Could use collectives

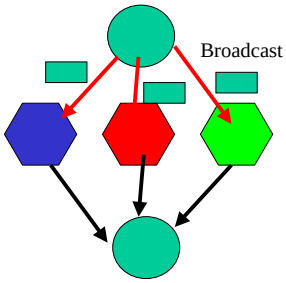
Case study



Case study

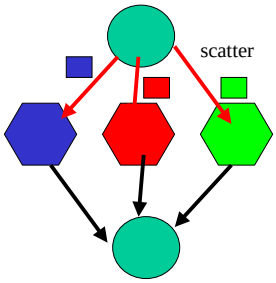


Case study



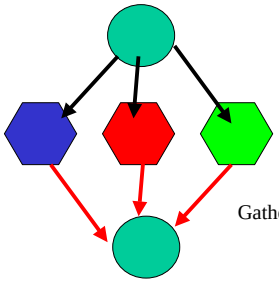
If broadcast all nodes get the same set of work
the workers have to understand what work they are doing

Case study



If scatter then custom work per worker can be sent

Case study



What Else is There

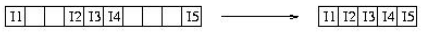
- Lots of other routines
  - Derived datatypes
  - Process groups and communicators
  - Process topologies
  - Profiling
- MPI-2
  - Parallel I/O
  - Dynamic process management
  - One sided communication

But we stop here
The rest is useful
as a reference

Derived Datatypes

- A derived datatype is a sequence of primitive datatypes and displacements
- Derived datatypes are created by building on primitive datatypes
- A derived datatype's *typemap* is the sequence of (primitive type, disp) pairs that defines the derived datatype
  - These displacements need not be positive, unique, or increasing.
- A datatype's type signature is just the sequence of primitive datatypes
- A messages type signature is the type signature of the datatype being sent, repeated *count* times

Derived Datatypes (cont)



Typemap =
(MPI_INT, 0) (MPI_INT, 12) (MPI_INT, 16) (MPI_INT, 20) (MPI_INT, 36)

Type Signature =
{MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT}

Type Signature =
{MPI_INT, MPI_INT, MPI_INT, MPI_INT, MPI_INT}

In collective communication, the type signature of data sent must match the type signature of data received!

Derived Datatypes (cont)

- Lower Bound: The lowest displacement of an entry of this datatype
- Upper Bound: Relative address of the last byte occupied by entries of this datatype, rounded up to satisfy alignment requirements
- Extent: The span from lower to upper bound
- MPI_GET_EXTENT (datatype, lb, extent)
- MPI_TYPE_SIZE (datatype, size)
- MPI_GET_ADDRESS (location, address)

Datatype Constructors

- MPI_TYPE_DUP (oldtype, newtype)
  - Simply duplicates an existing type
  - Not useful to regular users
- MPI_TYPE_CONTIGUOUS (count, oldtype, newtype)
  - Creates a new type representing *count* contiguous occurrences of *oldtype*
  - ex: MPI_TYPE_CONTIGUOUS (2, MPI_INT, 2INT)
    - creates a new datatype *2INT* which represents an array of 2 integers

Datatype 2INT

CONTIGUOUS DATATYPE

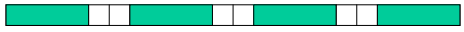
P1 sends 100 integers to P2

```
P1
int buff[100];
MPI_Datatype dtype;
...
...
MPI_Type_contiguous (100,
   MPI_INT, &dtype);
MPI_Type_commit (&dtype);

MPI_Send (buff, 1, dtype, 2, tag,
   MPI_COMM_WORLD)
```

```
P2
int buff[100]

MPI_Recv (buff, 100, MPI_INT, 1, tag,
   MPI_COMM_WORLD, &status)
```

Datatype Constructors (cont)

- MPI_TYPE_VECTOR (count, blocklength, stride, oldtype, newtype)
  - Creates a datatype representing *count* regularly spaced occurrences of *blocklength* contiguous *oldtype*s
  - *stride* is in terms of elements of *oldtype*
  - ex: MPI_TYPE_VECTOR (4, 2, 3, 2INT, AINT)



AINT

Datatype Constructors (cont)

- MPI_TYPE_HVECTOR (count, blocklength, stride, oldtype, newtype)
  - Identical to MPI_TYPE_VECTOR, except stride is given in bytes rather than elements.
  - ex: MPI_TYPE_HVECTOR (4, 2, 20, 2INT, BINT)

EXAMPLE

- REAL a(100,100), B(100,100)

- CALL MPI_COMM_RANK (MPI_COMM_WORLD, myrank, ierr)
- CALL MPI_TYPE_SIZE (MPI_REAL, sizeofreal, ierr)
- CALL MPI_TYPE_VECTOR (100, 1, 100,MPI_REAL, rowtype, ierr)
- CALL MPI_TYPE_CREATE_HVECTOR (100, 1, sizeofreal, rowtype, xpose, ierr)
- CALL MPI_TYPE_COMMIT (xpose, ierr)
- CALL MPI_SENDRECV (a, 1, xpose, myrank, 0, b, 100*100, MPI_REAL, myrank, 0, MPI_COMM_WORLD, status, ierr)

Datatype Constructors (cont)

MPI_TYPE_INDEXED (count, blocklengths, displs, oldtype, newtype)

- Allows specification of non-contiguous data layout
- Good for irregular problems
- ex: MPI_TYPE_INDEXED (3, lengths, displs, 2INT, CINT)
  - lengths = (2, 4, 3) displs = (0,3,8)

- Most often, block sizes are all the same (typically 1)
- MPI-2 introduced a new constructor

Datatype Constructors (cont)

- MPI_TYPE_CREATE_INDEXED_BLOCK (count, blocklength, displs, oldtype, newtype)
  - Same as MPI_TYPE_INDEXED, except all blocks are the same length (*blocklength)*
  - ex: MPI_TYPE_INDEXED_BLOCK (7, 1, displs, MPI_INT, DINT)
    - displs = (1, 3, 4, 6, 9, 13, 14)

Datatype Constructors (cont)

- MPI_TYPE_CREATE_HINDEXED (count, blocklengths, displs, oldtype, newtype)
  - Identical to MPI_TYPE_INDEXED except displacements are in bytes rather then elements
- MPI_TYPE_CREATE_STRUCT (count, lengths, displs, types, newtype)
  - Used mainly for sending arrays of structures
  - count is number of fields in the structure
  - lengths is number of elements in each field
  - displs should be calculated (portability)

MPI_TYPE_CREATE_STRUCT

```
MPI_Datatype stype;
MPI_Datatype types[3] =
 {MPI_CHAR, MPI_DOUBLE,
   MPI_CHAR};
int lens[3] = {1, 6, 7};
MPI_Aint displs[3] = {0,
   sizeof(double), 7*sizeof(double)};

MPI_Type_create_struct (3, lens,
   displs, types, &stype);
```

Non-portable

```
struct s1 {
  char class;
  double d[6];
  char b[7];
};

struct s1 sarray[100];
```

```
MPI_Datatype stype;
MPI_Datatype types[3] =
 {MPI_CHAR, MPI_DOUBLE,
   MPI_CHAR};
int lens[3] = {1, 6, 7};
MPI_Aint displs[3];

MPI_Get_address (&sarray[0].class,
   &displs[0]);
MPI_Get_address (&sarray[0].d,
   &displs[1]);
MPI_Get_address (&sarray[0].b,
   &displs[2]);
for (i=2;i>=0;i--) displs[i] -=displs[0]
MPI_Type_create_struct (3, lens,
   displs, types, &stype);
```

Semi-portable

MPI_TYPE_CREATE_STRUCT

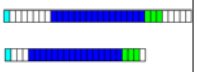
```
int i;
char c[100];
float f[3];
int a;
MPI_Aint disp[4];
int lens[4] = {1, 100, 3, 1};
MPI_Datatype types[4] = {MPI_INT, MPI_CHAR, MPI_FLOAT, MPI_INT};
MPI_Datatype stype;

MPI_Get_address(&i, &disp[0]);
MPI_Get_address(c, &disp[1]);
MPI_Get_address(f, &disp[2]);
MPI_Get_address(&a, &disp[3]);

MPI_Type_create_struct(4, lens, disp, types, &stype);
MPI_Type_commit (&stype);
MPI_Send (MPI_BOTTOM, 1, stype, ……..);
```

Derived Datatypes (cont)

- MPI_TYPE_CREATE_RESIZED (oldtype, lb, extent, newtype)
  - sets a new lower bound and extent for oldtype
  - Does NOT change amount of data sent in a message
    - only changes data access pattern

MPI_TYPE_CREATE_RESIZED

```
Struct s1 {
  char class;
  double d[2];
  char b[3];
};

struct s1 sarray[100];
```



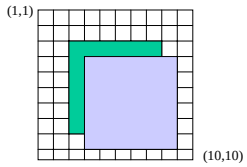
```
MPI_Datatype stype, ntype;
MPI_Datatype types[3] =
  {MPI_CHAR, MPI_DOUBLE,
    MPI_CHAR};
int lens[3] = {1, 6, 7};
MPI_Aint displs[3];

MPI_Get_address (&sarray[0].class,
   &displs[0];
MPI_Get_address (&sarray[0].d,
   &displs[1];
MPI_Get_address (&sarray[0].b,
   &displs[2];
for (i=2;i>=0;i--) displs[i] -=displs[0]
MPI_Type_create_struct (3, lens,
   displs, types, &stype);
MPI_Type_create_resized (stype, 0,
   sizeof(struct s1), &ntype);
```

Really Portable

Datatype Constructors (cont)

- MPI_TYPE_CREATE_SUBARRAY (ndims, sizes, subsizes, starts, order, oldtype, newtype)
  - Creates a *newtype* which represents a contiguous subsection of an array with *ndims* dimensions
    - This sub-array is only contiguous conceptually, it may not be stored contiguously in memory!
  - Arrays are assumed to be indexed starting a zero!!!
  - *Order must be MPI_ORDER_C or MPI_ORDER_FORTRAN*
    - C programs may specify Fortran ordering, and vice-versa

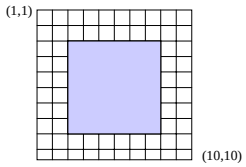
Datatype Constructors: Subarrays



MPI_TYPE_CREATE_SUBARRAY (2, sizes, subsizes, starts, MPI_ORDER_FORTRAN, MPI_INT, sarray)

sizes = (10, 10)
subsizes = (6,6)
starts = (3, 3)

Datatype Constructors: Subarrays



MPI_TYPE_CREATE_SUBARRAY (2, sizes, subsizes, starts, MPI_ORDER_FORTRAN, MPI_INT, sarray)

sizes = (10, 10)
subsizes = (6,6)
starts = (2,2)

Datatype Constructors: Darrays

- MPI_TYPE_CREATE_DARRAY (size, rank, dims, gsizes, distribs, dargs, psizes, order, oldt, newtype)
  - Used with arrays that are distributed in HPF-like fashion on Cartesian process grids
  - Generates datatypes corresponding to the sub-arrays stored on each processor
  - Returns in *newtype* a datatype specific to the sub-array stored on process *rank*

Datatype Constructors (cont)

- Derived datatypes must be committed before they can be used
  - MPI_TYPE_COMMIT (datatype)
  - Performs a "compilation" of the datatype description into an efficient representation
- Derived datatypes should be freed when they are no longer needed
  - MPI_TYPE_FREE (datatype)
  - Does not effect datatypes derived from the freed datatype or current communication

Pack and Unpack

- MPI_PACK (inbuf, incount, datatype, outbuf, outsize, position, comm)
- MPI_UNPACK (inbuf, insize, position, outbuf, outcount, datatype, comm)
- MPI_PACK_SIZE (incount, datatype, comm, size)
  - Packed messages must be sent with the type MPI_PACKED
  - Packed messages can be received with any matching datatype
  - Unpacked messages can be received with the type MPI_PACKED
  - Receives must use type MPI_PACKED if the messages are to be unpacked

Pack and Unpack

```
int i;
char  c[100];
MPI_Aint disp[2];
int lens[2] = {1, 100};
MPI_Datatype types[2] = {MPI_INT, MPI_CHAR};
MPI_Datatype type1;

MPI_Get_address (&i, &(disp[0]);
MPI_Get_address(c, &(disp[1]);
MPI_Type_create_struct (2, lens, disp, types, &type1);
MPI_Type_commit (&type1);

MPI_Send (MPI_BOTTOM, 1, type1, 1, 0, MPI_COMM_WORLD)
```

```
Char c[100];
MPI_Status status;
int i, comm;
MPI_Aint disp[2];
int len[2] = {1, 100};
MPI_Datatype types[2] = {MPI_INT, MPI_CHAR};
MPI_Datatype type1;

MPI_Get_address (&i, &(disp[0]);
MPI_Get_address(c, &(disp[1]);
MPI_Type_create_struct (2, lens, disp, types, &type1);
MPI_Type_commit (&type1);

comm = MPI_COMM_WORLD;
MPI_Recv (MPI_BOTTOM, 1, type1, 0, 0, comm, &status);
```

```
int i;
char c[100];
char buf[110];
int pos = 0;

MPI_Pack(&i, 1, MPI_INT, buf, 110, &pos, MPI_COMM_WORLD);
MPI_Pack(c, 100, MPI_CHAR, buf, 110, &pos, MPI_COMM_WORLD);

MPI_Send(buf, pos, MPI_PACKED, 1, 0, MPI_COMM_WORLD);
```

```
int i, comm;
char c[100];
MPI_Status status;
char buf[110]
int pos = 0;

comm = MPI_COMM_WORLD;
MPI_Recv (buf, 110, MPI_PACKED, 1, 0, comm, &status);
MPI_Unpack (buf, &pos, &i, 1, MPI_INT, comm);
MPI_Unpack (buf, 110, &pos, c, 100, MPI_CHAR, comm);
```

Derived Datatypes: Performance Issues

- May allow the user to send fewer or smaller messages
  - System dependant on how well this works
- May be able to significantly reduce memory copies
- can make I/O much more efficient
- Data packing may be more efficient if it reduces the number of send operations by packing meta-data at the front of the message
  - This is often possible (and advantageous) for data layouts that are runtime dependant

Communicators and Groups

- If you need to handle lots of processes in a simple way by breaking them into relative groups that have a certain relationship
  - Column communicator
  - Row communicator
  - Simplifying communication
    - A group just for summing a residue value

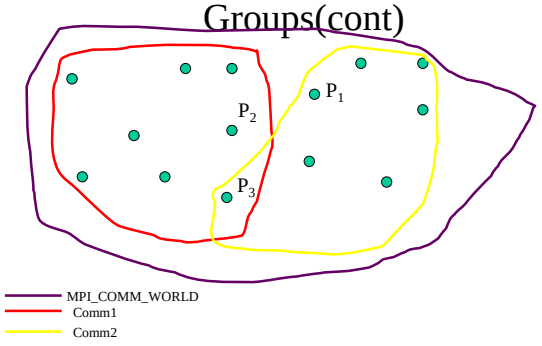
Communicators and Groups

- Many MPI users are only familiar with MPI_COMM_WORLD
- A communicator can be thought of a handle to a group
- A group is an ordered set of processes
  - Each process is associated with a rank
  - Ranks are contiguous and start from zero
- For many applications (dual level parallelism) maintaining different groups is appropriate
- Groups allow collective operations to work on a subset of processes
- Information can be added onto communicators to be passed into routines

Communicators and Groups(cont)

- While we think of a communicator as spanning processes, it is actually unique to a process
- A communicator can be thought of as a handle to an object (group attribute) that describes a group of processes
- An intracommunicator is used for communication within a single group
- An intercommunicator is used for communication between 2 disjoint groups

Communicators and Groups(cont)



Communicators and Groups(cont)

- Refer to previous slide
  - There are 3 distinct groups
  - These are associated with MPI_COMM_WORLD, comm1, and comm2
  - P_3 is a member of all 3 groups and may have different ranks in each group(say 0, 3 , & 4)
  - If P_2 wants to send a message to P_1 it can use MPI_COMM_WORLD (intracommunicator) or an intercommunicator (covered later)
  - If P_2 wants to send a message to P_3 it can use MPI_COMM_WORLD (send to rank 0), comm1 (send to rank 3), or and intercommunicator

Group Management

- All group operations are local
- As will be clear, groups are initially not associated with communicators
- Groups can only be used for message passing within a communicator
- We can access groups, construct groups, and destroy groups

Group Accessors

- MPI_GROUP_SIZE(group, size)
  - MPI_Group group
  - int size
  - This routine returns the number of processes in the group
- MPI_GROUP_RANK(group, rank)
  - MPI_Group group
  - int rank
  - This routine returns the rank of the calling process

Group Accessors (cont)

- MPI_GROUP_TRANSLATE_RANKS (group1, n, ranks1, group2, ranks2)
  - MPI_Group group1, group2
  - int n, *ranks1, *ranks2
  - This routine takes an array of n ranks (ranks1) which are ranks of processes in group1. It returns in ranks2 the corresponding ranks of the processes as they are in group2
  - MPI_UNDEFINED is returned for processes not in group2

Groups Accessors (cont)

- MPI_GROUP_COMPARE (group1, group2 result)
  - MPI_Group group1, group2
  - int result
  - This routine returns the relationship between group1 and group2
  - If group1 and group2 contain the same processes, ranked the same way, this routine returns MPI_IDENT
  - If group1 and group2 contain the same processes, but ranked differently, this routine returns MPI_SIMILAR
  - Otherwise this routine returns MPI_UNEQUAL

Group Constructors

- Group constructors are used to create new groups from existing groups
- Base group is the group associated with MPI_COMM_WORLD
- Group creation is a local operation
  - No communication needed
- Following group creation, no communicator is associated with the group
  - No communication possible with new group

Group Constructors (cont)

- MPI_COMM_GROUP (comm, group)
  - MPI_Comm comm
  - MPI_Group group
  - This routine returns in group the group associated with the communicator comm

Group Constructors (cont) Set Operations

- MPI_GROUP_UNION(group1, group2, newgroup)
- MPI_GROUP_INTERSECTION(group1, group2, newgroup)
- MPI_GROUP_DIFFERENCE(group1, group2, newgroup)
  - MPI_Group group1, group2, *newgroup

Group Constructors (cont) Set Operations

- Union: Returns in newgroup a group consisting of all processes in group1 followed by all processes in group2, with no duplication
- Intersection: Returns in newgroup all processes that are in both groups, ordered as in group1
- Difference: Returns in newgroup all processes in group1 that are not in group2, ordered as in group1

Group Constructors (cont) Set Operations

- Let group1 = {a,b,c,d,e,f,g} and group2 = {d,g,a,c,h,I}
- MPI_Group_union(group1,group2,newgroup)
  - Newgroup = {a,b,c,d,e,f,g,h,I}
- MPI_Group_intersection(group1,group2,newgroup)
  - Newgroup = {a,c,d,g}
- MPI_Group_difference(group1,group2,newgroup)
  - Newgroup = {b,e,f}

Group Constructors (cont) Set Operations

- Let group1 = {a,b,c,d,e,f,g} and group2 = {d,g,a,c,h,I}
- MPI_Group_union(group2,group1,newgroup)
  - Newgroup = {d,g,a,c,h,l,b,e,f}
- MPI_Group_intersection(group2,group1,newgroup)
  - Newgroup = {d,g,a,c}
- MPI_Group_difference(group1,group2,newgroup)
  - Newgroup = {h,i}

Group Constructors (cont)

- MPI_GROUP_INCL(group, n, ranks, newgroup)
  - MPI_Group group, *newgroup
  - int n, *ranks
  - This routine creates a new group that consists of all the n processes with ranks ranks[0]..ranks[n-1]
  - The process with rank i in newgroup has rank ranks[i] in group

Group Constructors (cont)

- MPI_GROUP_EXCL(group, n, ranks, newgroup)
  - MPI_Group group, *newgroup
  - int n, *ranks
  - This routine creates a new group that consists of all the processes in group after deleting processes with ranks ranks[0]..ranks[n-1]
  - The ordering in newgroup is identical to the ordering in group

Group Constructors (cont)

- MPI_GROUP_RANGE_INCL(group, n, ranges, newgroup)
  - MPI_Group group, *newgroup
  - int n, ranges[][3]
  - Ranges is an array of triplets consisting of start rank, end rank, and stride
  - Each triplet in ranges specifies a sequence of ranks to be included in newgroup
  - The ordering in newgroup is as specified by ranges

Group Constructors (cont)

- MPI_GROUP_RANGE_EXCL(group, n, ranges, newgroup)
  - MPI_Group group, *newgroup
  - int n, ranges[][3]
  - Ranges is an array of triplets consisting of start rank, end rank, and stride
  - Each triplet in ranges specifies a sequence of ranks to be excluded from newgroup
  - The ordering in newgroup is identical to that in group

Group Constructors (cont)

- Let group = {a,b,c,d,e,f,g,h,i,j}
- n=5, ranks = {0,3,8,6,2}
- ranges= {(4,9,2),(1,3,1),(0,9,5)}
- MPI_Group_incl(group,n,ranks,newgroup)
  - newgroup = {a,d,I,g,c}
- MPI_Group_excl(group,n,ranks,newgroup)
  - newgroup = {b,e,f,h,j}
- MPI_Group_range_incl(group,n,ranges,newgroup)
  - newgroup = {e,g,I,b,c,d,a,f}
- MPI_Group_range_excl(group,n,ranges,newgroup)
  - newgroup = {h}

Communicator Management

- Communicator access operations are local, thus requiring no interprocess communication
- Communicator constructors are collective and may require interprocess communication
- All the routines in this section are for intracommunicators, intercommunicators will be covered separately

Communicator Accessors

- MPI_COMM_SIZE (comm, size)
  - Returns the number of processes in the group associated with comm
- MPI_COMM_RANK (comm, rank)
  - Returns the rank of the calling process within the group associated with comm
- MPI_COMM_COMPARE (comm1, comm2, result) returns:
  - MPI_IDENT if comm1 and comm2 are handles for the same object
  - MPI_CONGRUENT if comm1 and comm2 have the same group attribute
  - MPI_SIMILAR if the groups associated with comm1 and comm2have the same members but in different rank order
  - MPI_UNEQUAL otherwise

Communicator Constructors

- MPI_COMM_DUP (comm, newcomm)
- This routine creates a duplicate of comm
- newcomm has the same fixed attributes as comm
- Defines a new communication domain
  - A call to MPI_Comm_compare (comm, newcomm, result) would return MPI_CONGRUENT
- Useful to library writers and library users

Communicator Constructors

- MPI_COMM_CREATE (comm,group,newcomm)
  - This is a collective routine, meaning it must be called by all processes in the group associated with comm
  - This routine creates a new communicator which is associated with group
  - MPI_COMM_NULL is returned to processes not in group
  - All group arguments must be the same on all calling processes
  - group must be a subset of the group associated with comm

Communicator Constructors

- MPI_COMM_SPLIT(comm,color,key,newcomm)
  - MPI_Comm comm, newcomm
  - int color, key
  - This routine creates as many new groups and communicators as there are distinct values of color
  - The rankings in the new groups are determined by the value of key, ties are broken according to the ranking in the group associated with comm
  - MPI_UNDEFINED is used as the color for processes to not be included in any of the new groups

Communication Constructors

Rank	0	1	2	3	4	5	6	7	8	9	10
Process	a	b	c	d	e	f	g	h	i	j	k
Color	U	3	1	1	3	7	3	3	1	U	3
Key	0	1	2	3	1	9	3	8	1	0	0

Both process a and j are returned MPI_COMM_NULL
3 new groups are created
{i, c, d}
{k, b, e, g, h}
{f}

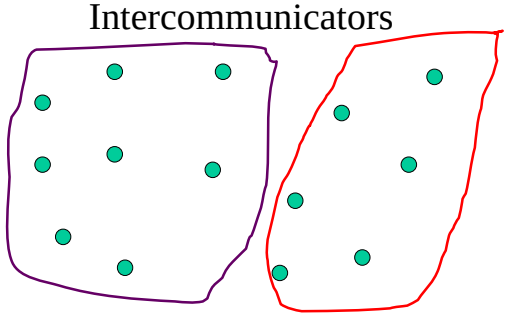
Destructors

- The communicators and groups from a process' viewpoint are merely handles
- Like all handles in MPI, there is a limited number available – YOU CAN RUN OUT
- MPI_GROUP_FREE (group)
- MPI_COMM_FREE (comm)

Intercommunicators

- Intercommunicators are associated with 2 groups of disjoint processes
- Intercommunicators are associated with a remote group and a local group
- A communicator is either intra or inter, never both

Intercommunicators



Intercommunicator

Intercommunicator Accessors

- MPI_COMM_TEST_INTER (comm, flag)
  - This routine returns true if comm is an intercommunicator, otherwise, false
- MPI_COMM_REMOTE_SIZE(comm, size)
  - This routine returns the size of the remote group associated with intercommunicator comm
- MPI_COMM_REMOTE_GROUP(comm, group)
  - This routine returns the remote group associated with intercommunicator comm

Intercommunicator Constructors

- The communicator constructors described previously will return an intercommunicator if the are passed intercommunicators as input
  - MPI_COMM_DUP: returns an intercommunicator with the same groups as the one passed in
  - MPI_COMM_CREATE: each process in group A must pass in group the same subset of group A (A1). Same for group B (B1). The new communicator has groups A1 and B1 and is only valid on processes in A1 and B1
  - MPI_COMM_SPLIT: As many new communicators as there are distinct pairs of colors are created

Communication Constructors

Rank	0	1	2	3	4	5	6	7	8	9	10
Process	a	b	c	d	e	f	g	h	i	j	k
Color	U	3	3	1	1	7	3	3	1	U	3
Key	0	1	2	3	1	9	3	8	1	0	0

A

Rank	0	1	2	3	4	5	6	7	8	9	10
Process	l	m	n	o	p	q	r	s	t	u	v
Color	5	3	1	U	3	3	3	7	1	U	3
Key	0	1	2	3	1	9	3	8	1	0	0

B

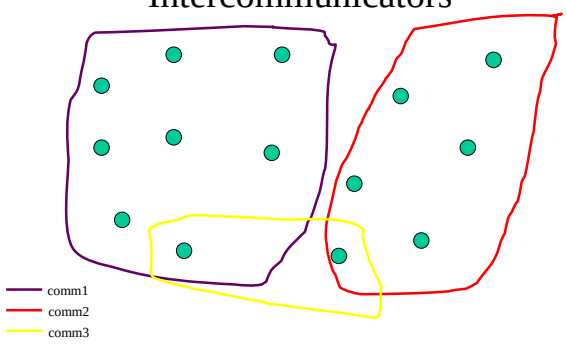
Intercommunicator Constructors

- Processes a, j, l, o, and u would all have MPI_COMM_NULL returned in newcomm
- newcomm1 would be associated with 2 groups: {e, i, d} and {t, n}
- newcomm2 would be associated with 2 groups: {k, b, c, g, h} and {v, m, p, r, q}
- newcomm3 would be associated with 2 groups: {f} and {s}

Intercommunicator Constructors

- MPI_INTERCOMM_CREATE (local_comm, local_leader, bridge_comm, remote_leader, tag, newintercomm)
- This routine is called collectively by all processes in 2 disjoint groups
- All processes in a particular group must provide matching local_comm and local_leader arguments
- The local leaders provide a matching bridge_comm (a communicator through which they can communicate), in remote_leader the rank of the other leader within bridge_comm, and the same tag
- The bridge_comm, remote_leader, and tag are significant only at the leaders
- There must be no pending communication across bridge_comm that may interfere with this call

Intercommunicators



Intercommunicators

- MPI_INTERCOMM_MERGE (intercomm, high, newintracomm)
  - This routine creates an intracommunicator from a union of the two groups associated with intercomm
  - High is used for ordering. All process within a particular group must pass the same value in for high (true or false)
  - The new intracommunicator is ordered with the high processes following the low processes
  - If both groups pass the same value for high, the ordering is arbitrary

Attribute Caching

- It is possible to *cache* attributes to be associated with a communicator
- This cached information is process specific.
- The same attribute can be cached with multiple communicators
- Many attributes can be cached with a single communicator
- This is most commonly used in libraries

Selected References

- MPI - The Complete Reference Volume 1, The MPI Core
- MPI - The Complete Reference Volume 2, The MPI Extensions
- USING MPI: Portable Parallel Programming with the Message-Passing Interface
- Using MPI-2: Advanced Features of the Message-Passing Interface