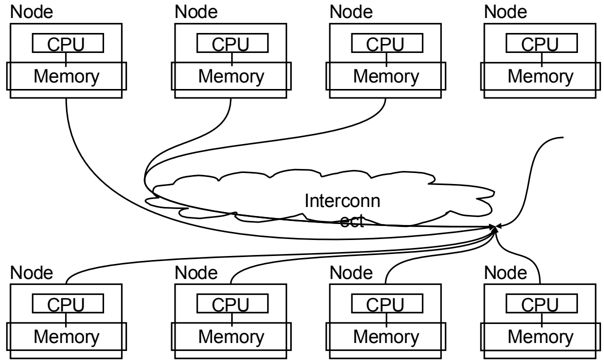
Slide 1

Programming with Message Passing using MPI

Slide 2

Message passing model

- Assumes a set of *independent* processes
  - Each with its own address space
- All communication requires explicit interaction between two procs
  - *Message passing*



Slide 3

MPI

- Very generalized library of routines.
- Abstracts over architecture/operating system/language
  - This gives portability.
- Basically asynchronous
  - Blocking
  - Non-blocking
- Basic comms primitives do not inherently give reliable delivery
  - But protocols can be defined to cope with unreliable delivery
    - » eg *transfer-acknowledge* and *request-transfer-acknowledge*
- The aim is to provide generalized communications on top of a reasonable virtual architecture.

Slide 4

Properties of MPI

- SPMD model
  - all tasks run the same program.
- Communications organized according to *contexts*
  - a context is required for each message (so we know who should listen).
- At the start we have one user context called:
  `MPI_COMM_WORLD`
- All users tasks belong to this context.
- Each task has a number within the contexts it belongs to.
  - We can extract this task number by calling:
    `MPI_Comm_rank(context,&id);`
  - where `id` is an integer variable

Slide 5

Outline of simple MPI program.

```
main(int argc, char *argv[]){
   MPI_Init(&argc, &argv);

   … initialization stuff here …

   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   if (myrank == 0)
     master();
   else
     slave();

   … finalization stuff here ..
   MPI_finalize();
}
```

Slide 6

Communications Groups in MPI

- We start with one global communications context.
  - this context is associated with a pre-existing group.
- What if we want multiple contexts?
  - We make more groups out of the pre-existing group.
  - We attach contexts to these new groups.
  - More than one context can be attached to one group.
- How do we access the pre-existing group?

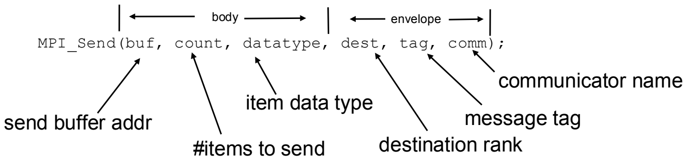
```
MPI_Comm world_group; /*declare a communicator ptr*/
MPI_Comm_group(MPI_COMM_WORLD,&world_group);
```

- After this code the group can be accessed via `&world_group`
- Built-in function available to create new groups and communications contexts *(communicators).*
- Built-in functions also available to emulate *topologies*.

Slide 7

Communications Primitives

- Communications primitives are either *point-to-point* or *collective* operations.
- Both types use a communications context.
- Point-to-point send:

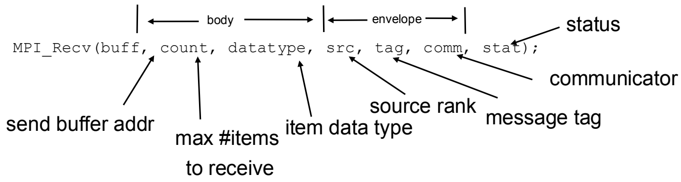


- Source rank is implicitly defined in envelope
- 'tag' is defined by the programmer to distinguish between different types of messages. For example: data and status.

Slide 8

Receive

- Point to point receive:



- For a message to be received, its *envelope* must match the one specified in MPI_Recv.
- In general, many messages may be pending: envelope is used to select only those of immediate interest.
- 'tag' and 'src' can also take 'wildcard' values. ie. accept messages of any tag or any src respectively.

Slide 9

Using Point to Point primitives

- Code to send an integer from task 0 to task 1

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0){
   int x;
   MPI_Send(&x,1,MPI_INT,1,msgtag,MPI_COMM_WORLD);
}else if(myrank == 1){
   int x;
   MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD,
            &status);
}
```

Slide 10

Blocking vs Non-Blocking

- These are *blocking* sends and receives.
  - They wait until the operation has completed before proceeding.
  - 'Operation Completion' is different for Send and Receive:
    - » Send is *complete* when the message buffer has been fully transferred to the MPI system. That is, when it is safe for the program to modify/reuse the buffer. (ie. `x` in the previous example)
    - » Receive is *complete* when the message data has arrived at the destination and is available for use.
  - Note that this is different from the normal use of the term 'blocking send', which normally means "wait until the message has arrived safely at its destination"
- MPI also provides *non-blocking* send and receive.
  - These just continue with no regard for completion status.
  - Can be useful to help avoid *deadlock*.
  - sending / receiving processes can use *polling* to check status of non-blocking operations.
- Note in a "normal" context both MPI blocking and non-blocking comms primitives would be called non-blocking .

Slide 11

Collective Communication.

- Some MPI collective communications primitives
- `MPI_Bcast()     -Broadcast from root to all tasks`
- `MPI_Gather()    -Gather values for a task group.`
- `MPI_Scatter()   -Scatters parts of buffer to group.`
- `MPI_AlltoAll()  -Sends from all tasks to all tasks`
- `MPI_Reduce()    -Combine values from group into one.`
- `MPI_Reduce_scatter() - Comb vals and scatter results.`
- `MPI_Scan()      -Parallel prefix.`
- Barrier Synchronization:
  - The tasks in a communications context can meet at a barrier by calling:

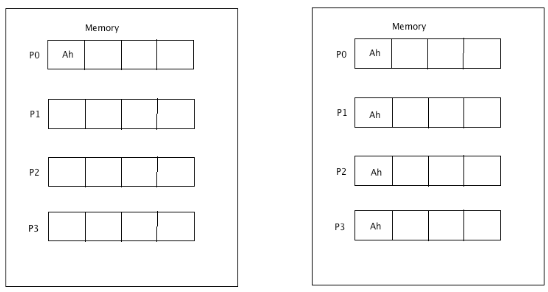
```
MPI_Barrier(communicator);
```

where `communicator` is a communications context.

Slide 12

Collective Communications - BCast

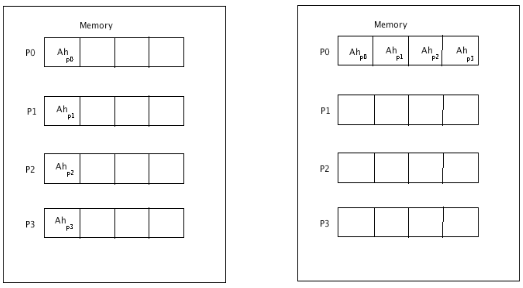
- Copies data from the root node to the same memory location in every other node.



Slide 13

Collective Communications - Gather

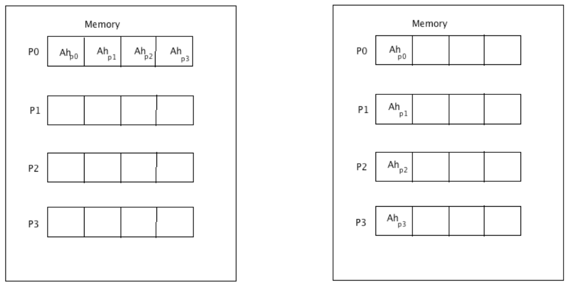
- Each node sends the contents of the send buffer to the root node
- Root node stores them in *rank* order



Slide 14

Collective Communications - Scatter

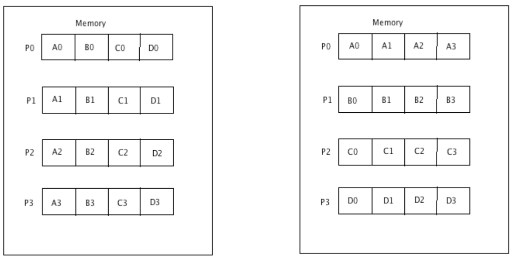
- Root process splits buffer into equal chunks and sends one chunk to each processor



Slide 15

Collective Communications - AlltoAll

- Each node performs a *Scatter* operation on its own data.
- Thus every node receives some data from every other node.



Slide 16

Sample MPI program

- Declarations.

```
#include “mpi.h”
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000

void main(int argc, char *argv[]){
   int myid,numprocs;
   int data[MAXSIZE], i, x, low, high, myresult,
       result;
   char fn[255];
   char *fp;
```

Slide 17

Example continued.

- Init code and code for supervisor process

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
if (my_id == 0){
   strcpy(fn,getenv(“HOME”));
   strcat(fn,”/MPI/rand_data.txt”);
   if((fp = fopen(fn,”r”)) == NULL){
      printf(“Can’t open the input file”);
      exit(1);
   }
   for (i=0; i < MAXSIZE; i++)
     fscanf(fp,”%d”,&data[i]);
}
```

Slide 18

Example cont’d

- Code for everyone..

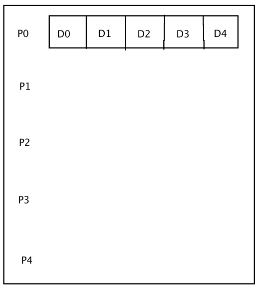
```
MPI_Bcast(data,MAXSIZE,MPI_INT,0,MPI_COMM_WORLD);
x = MAXSIZE / numprocs;
low = myid * x;
high = low + x;
for(i = low; i < high; i++)
   myresult += data[i];
printf(“I got %d from %d\n”,myresult, myid);

MPI_Reduce(&myresult,&result,1,MPI_INT,MPI_SUM,0,
           MPI_COMM_WORLD);
if (myid == 0) printf(“the sum is %d.\n”,result);
MPI_Finalize();
}
```

root task, MPI_Bcast must be called by all tasks in the communicator

Slide 19

Example cont’d

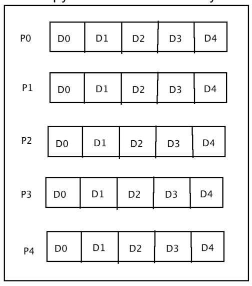
- Assume we have 5 processors
- Data array is created on the root node: 0



Slide 20

Example cont’d

- `MPI_Bcast(data,MAXSIZE,MPI_INT,0,MPI_COMM_WORLD);`
- Bcast sends a copy of the *entire* data array to each node.

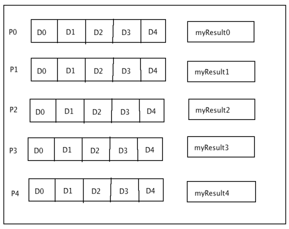


Slide 21

Example cont’d

- Each process sums its respective partition and puts the answer in myResult

```
x = MAXSIZE / numprocs;
low = myid * x;
high = low + x;
for(i = low; i < high; i++)
   myresult += data[i];
```



Slide 22

Example cont’d

- Finally, a parallel reduction is performed summing the values of myResult.

```
MPI_Reduce(&myresult,&result,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
if (myid == 0) printf(“the sum is %d.\n”,result);
```