High Performance Parallel Programming (CS61064)

Week – 5
Part 1

Pralay Mitra

MPI APIs

- Emphasis on messages
- MPI is huge (about 400 functions in MPI-2)
- MPI is small
  - 9 concepts:
    - init, finalize
    - size, rank, communicator
    - send, receive, broadcast, reduce
  - 6 variations:
    - standard, synchronous, ready, buffered
    - blocking, non-blocking

MPI functions

- Function format
  - int retval = MPI_Xxxxx(param1, param2, …);
  - MPI_Xxxxx(param1, param2, …);
- Initialization MPI
  - int MPI_Init(int *argc, char ***argv);
- Exiting MPI
  - int MPI_Finalize();
- Note

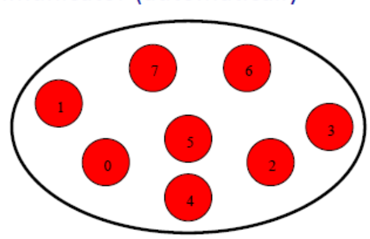
  No MPI calls are allowed before MPI_Init and after MPI_Finalize. However, the program can go on as a serial program.

MPI Communicator

- In MPI it is possible to divide the total number of processes into groups, called *communicators*.
- The Communicator is a variable identifying a group of processes that are allowed to communicate with each other.
- The communicator that includes all processes is called MPI_COMM_WORLD
- MPI_COMM_WORLD is the default communicator (automatically defined):

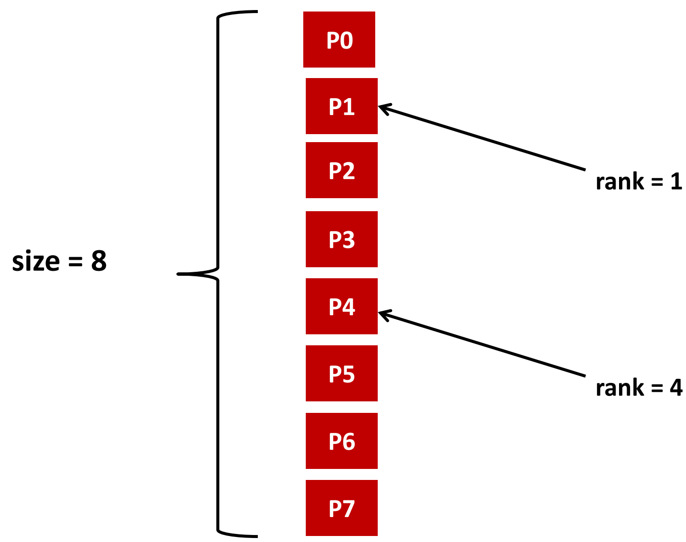
All MPI communication subroutines have a communicator argument.

The Programmer can define many communicators at the same time



MPI_COMM_WORLD

MPI Communicator



MPI Communicator

- Communicator size
  - The number of processors associated with a communicator.
    - MPI_Comm_size(MPI_Comm comm, int *size);
- Process rank
  - Identify different processes through the ID of a processor in a group
    - MPI_Comm_rank(MPI_Comm comm, int *rank);

MPI_ABORT

- Purpose
  - To terminate all MPI processes associated with the communicator comm; in most systems (all to date), terminate all processes.

- Function
  - int MPI_Abort(MPI_Comm comm, int errorcode);

Your second MPI program

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

int main(int argc, char *argv[])
{
    int ret, nproc, myid;

    ret=MPI_Init(&argc,&argv);
    ret=MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    ret=MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    printf("Number of processors %d ... my id is %d\n",nproc,myid);

    ret=MPI_Finalize();

    return 0;
}
```

Your second MPI program

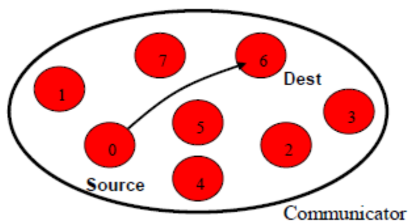
```
$ mpicc template.c
$ mpirun -np 8 ./a.out
Number of processors 8 ... my id is 4
Number of processors 8 ... my id is 3
Number of processors 8 ... my id is 6
Number of processors 8 ... my id is 0
Number of processors 8 ... my id is 2
Number of processors 8 ... my id is 1
Number of processors 8 ... my id is 7
Number of processors 8 ... my id is 5
$
```

MPI

Point-to-Point Communications

Point-to-Point Communication

- It is the basic communication method provided by MPI library. Communication between 2 processes
- It is conceptually simple: source process A sends a message to destination process B, B receive the message from A.
- Communication take places within a communicator
- Source and Destination are identified by their rank in the communicator



Point-to-Point Communication

if rank == i
    send information
else if rank ==j
    receive information

if rank == i
    send information
else
    receive information

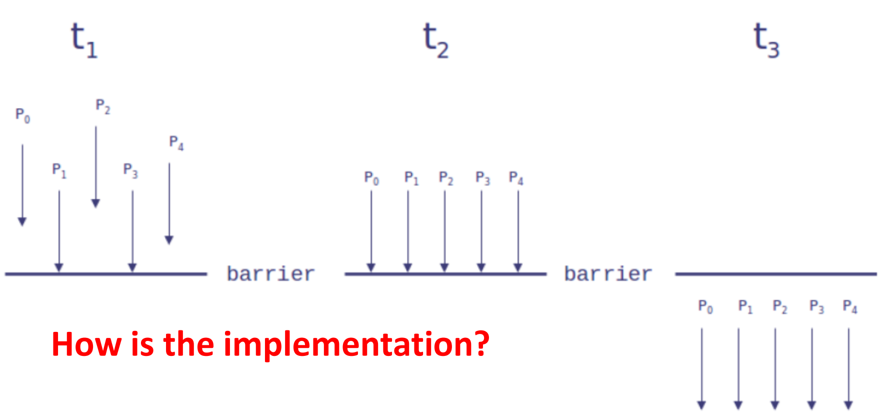
Communication type

Communications involving a group of processes. They are called by all the ranks involved in a communicator (or a group) and are of three types:

- Synchronization (e.g. Barrier)
- Data Movement (e.g. Broadcast or Gather/scatter)
- Global Computation (e.g. reductions)

MPI Barrier

- It stops all processes within a communicator until they are synchronized
  - *int MPI_Barrier(MPI_Comm comm);*



How is the implementation?

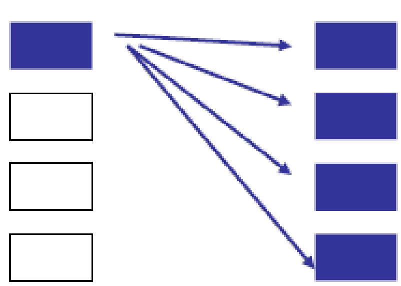
Send and receive

```
int MPI_Send(void *buf, int count, MPI_Datatype
   type, int dest, int tag, MPI_Comm comm);

int MPI_Recv (void *buf, int count, MPI_Datatype
   type, int source, int tag, MPI_Comm comm,
   MPI_Status *status);
```

MPI Broadcast

*int MPI_Bcast (void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)*

- Note that all processes must specify the same root and same comm.



Question?

- Check the time efficiency of *MPI_Broadcast* instead of implementing using *MPI_Send* and *MPI_Receive*.

MPI

Data Type

Datatypes

- **MPI Datatypes**
  - Basic
  - Derived (MPI_Type_xxx functions)
- Derived type
- User defined type
  - Allows MPI to automatically scatter and gather data from non-contiguous buffers.

- C/C++ MPI handles to refer to datatypes and structures are macro to structs (#define MPI_INT …)

MPI Intrinsic Datatypes

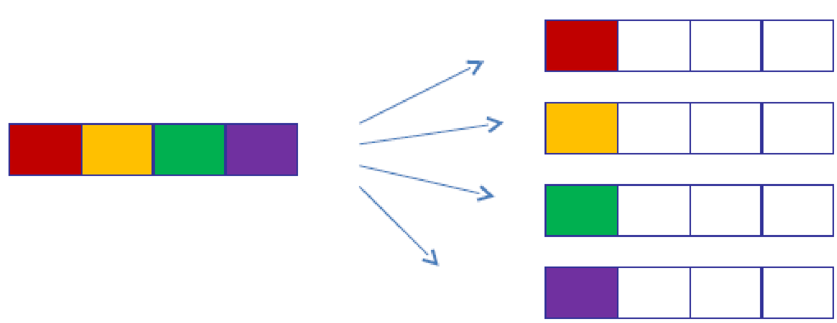
MPI Datatype	C Datatype
MPI_CHAR	singed char
MPI_SHORT	singed short int
MPI_INT	singed int
MPI_LONG	singed long int
MPI_UNSIGNED_CHAR	unsinged char
MPI_UNSIGNED_SHORT	unsinged short int
MPI_UNSIGNED	unsinged int
MPI_UNSIGNED_LONG	unsinged long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED	

MPI Scatter

- The root sends a message. The message is split into n equal segments, the *i*-th segment is sent to the *i*-th process in the group and each process receives this message.

MPI Scatter

*int MPI_Scatter(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)*

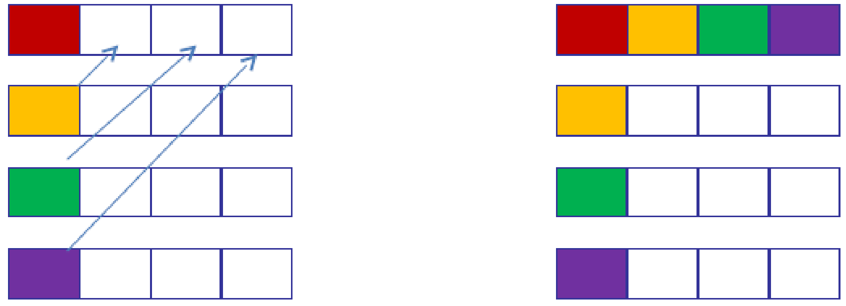


MPI Gather

- Each process, root included, sends the content of its send buffer to the root process. The root process receives the messages and stores them in the rank order.
- *recvcnt* parameter is the count of elements received per process, not the total summation of counts from all processes.

MPI Gather

*int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)*

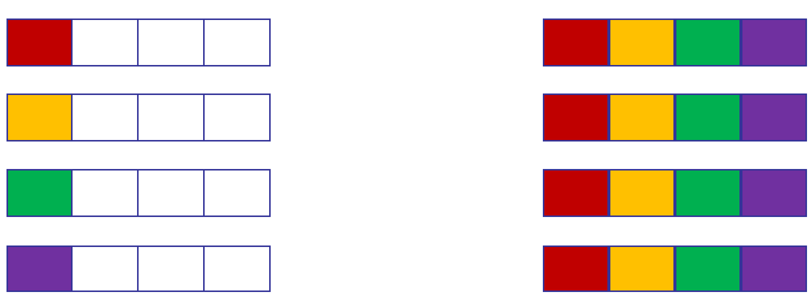


Workout

- Compute the histogram of an image using *MPI_Scatter* and *MPI_Gather*

MPI Allgather

- There are possible combinations of collective functions - a combination of a gather + a broadcast.
- *int MPI_Allgather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)*



Workout

- Using *MPI_Allgather* Normalize an image so that its intensity varies from *0* to *1.0* after the normalization.

MPI Reduce

*int MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)*

- Input Parameters
  - sendbuf: address of send buffer (choice)
  - Count: number of elements in send buffer (integer)
  - datatype: data type of elements of send buffer (handle)
  - op: reduce operation (handle)
  - root: rank of root process (integer)
  - comm: communicator (handle)

- Output Parameters
  - recvbuf: address of receive buffer (choice, significant only at root)

MPI reduction operations

- *MPI_MAX* - Returns the maximum element.
- *MPI_MIN* - Returns the minimum element.
- *MPI_SUM* - Sums the elements.
- *MPI_PROD* - Multiplies all elements.
- *MPI_LAND* - Performs a logical *and* across the elements.
- *MPI_LOR* - Performs a logical *or* across the elements.
- *MPI_BAND* - Performs a bitwise *and* across the bits of the elements.
- *MPI_BOR* - Performs a bitwise *or* across the bits of the elements.
- *MPI_MAXLOC* - Returns the maximum value and the rank of the process that owns it.
- *MPI_MINLOC* - Returns the minimum value and the rank of the process that owns it.

Workout

- Compute the average of a list of numbers using *MPI_Reduce.*

High Performance Parallel Programming (CS61064)

Week – 5
Part 2

Pralay Mitra

Calculation of PI using MPI

```
#include "mpi.h"
#include <math.h>
#include <stdio.h>
int main(int argc,char *argv[])
{
        int done = 0, n, myid, numprocs, i;
        double PI25DT = 3.141592653589793238462643;
        double mypi, pi, h, sum, x;
        MPI_Init(&argc,&argv);
        MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank(MPI_COMM_WORLD,&myid);
        while (!done) {
                if (myid == 0) {
                        printf("Enter the number of intervals: (0 quits) ");
                        scanf("%d",&n);
                }
                MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
                if (n == 0) break;
                h = 1.0 / (double) n;
                sum = 0.0;
                for (i = myid + 1; i <= n; i += numprocs) {
                        x = h * ((double)i - 0.5);
                        sum += 4.0 / (1.0 + x*x);
                }
                mypi = h * sum;
                MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
                if (myid == 0) printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
        }
        MPI_Finalize();
        return 0;
}
```

Using Bcast and Reduce

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);

MPI Allreduce

*int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)*

- Input Parameters
  - sendbuf: address of send buffer (choice)
  - count: number of elements in send buffer (integer)
  - datatype: data type of elements of send buffer (handle)
  - op: reduce operation (handle)
  - comm: communicator (handle)
- Output Parameters
  - recvbuf: address of receive buffer (choice)

Calculation of PI using MPI

```
int main(int argc, char* argv[]){
    int i,id, np,N;
    double x, y;
    int lhit;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &np);
    sscanf(argv[1], "%d", &N);
    MPI_Barrier(MPI_COMM_WORLD);
    lhit = 0;
    srand((unsigned)(time(0)));
    int lN = N/np;
    for(i = 0; i<lN;i++){
        x = ((double)rand())/((double)RAND_MAX);
        y = ((double)rand())/((double)RAND_MAX);
        if (((x*x) + (y*y)) <= 1) lhit++;
    }
    int hit=0;
    MPI_Allreduce(&lhit,&hit,1,MPI_INT,MPI_SUM,MPI_COMM_WORLD);
    double est;
    est = (hit*4)/((double)N);
    MPI_Barrier(MPI_COMM_WORLD);
    if (id == 0) {
        printf("Estimate of Pi:      %24.16f\n",est);
    }
    MPI_Finalize();
    return 0;
}
```

Using Allreduce and Barrier

MPI_Allreduce(&lhit,&hit,
1,MPI_INT,MPI_SUM,MPI_
COMM_WORLD);

MPI_Barrier(MPI_COMM_
WORLD);

Check the correctness.

MPI Library

- *MPI_Abort*
- *MPI_Broadcast*
- *MPI_Scatter*
- *MPI_Gather*
- *MPI_Allgather*
- *MPI_Reduce*
- *MPI_Allreduce*

MPI

The message

The message

Message Structure

envelope				body		
source	destination	communicator	tag	buffer	count	datatype

- Messages are identified by their envelopes.
- Data is exchanged in the buffer, an array of count elements of some particular MPI data type.
- Type of data is a must. C types are different from Fortran types.
- MPI programs run in heterogeneous environments.

For successful communication

- The communicator must be the same for sender and receiver.
- Sender must specify a valid destination rank.
- Receiver must specify a valid source rank.
- Tags must match.
- Buffers must be large enough to accommodate all data.

Message Structure

envelope				body		
source	destination	communicator	tag	buffer	count	datatype

Completion

- In a perfect world, every send operation would be perfectly synchronized with its matching receive. This is rarely the case. The MPI implementation is able to deal with storing data when the two tasks are out of sync.
- **Completion** of the communication means that memory locations used in the message transfer can be safely accessed
  - Send: variable sent can be reused after completion
  - Receive: variable received can be used after completion

Send and receive

```
int MPI_Send(void *buf, int count, MPI_Datatype
   type, int dest, int tag, MPI_Comm comm);

int MPI_Recv (void *buf, int count, MPI_Datatype
   type, int source, int tag, MPI_Comm comm,
   MPI_Status *status);
```

MPI_Recv accepts wild cards: MPI_ANY_SOURCE, MPI_ANY_TAG

Actual source and tag is returned in the receiver's status parameter.

Example

```
int size, rank, n;
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
    n = -1;
    MPI_Send(&n, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank == 1) {
    MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
           number);
}
```

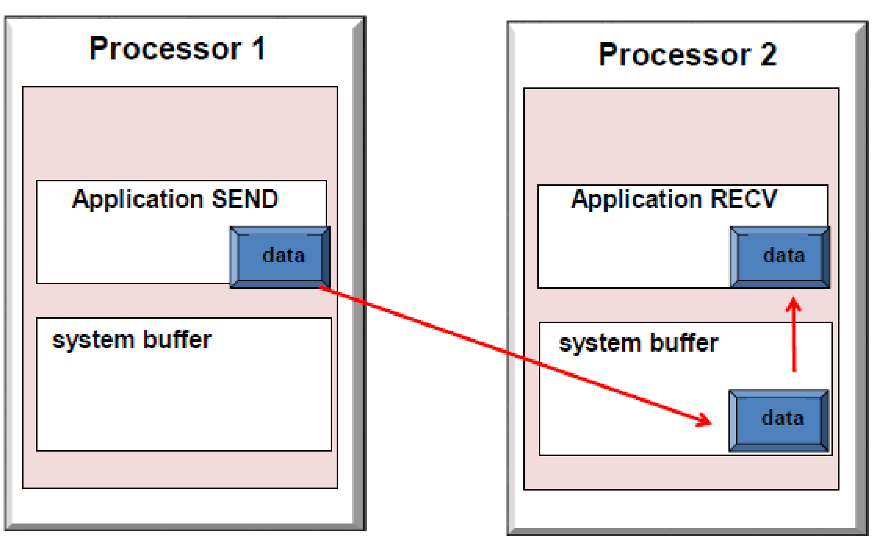
Work out

1. Modify the **`pi`** program to use *send* and *receive* instead of bcast/reduce.

2. Write a program that sends a message around a ring. That is, process 0 reads a line from the terminal and sends it to process 1, who sends it to process 2, etc. The last process sends it back to process 0, who prints it.

46

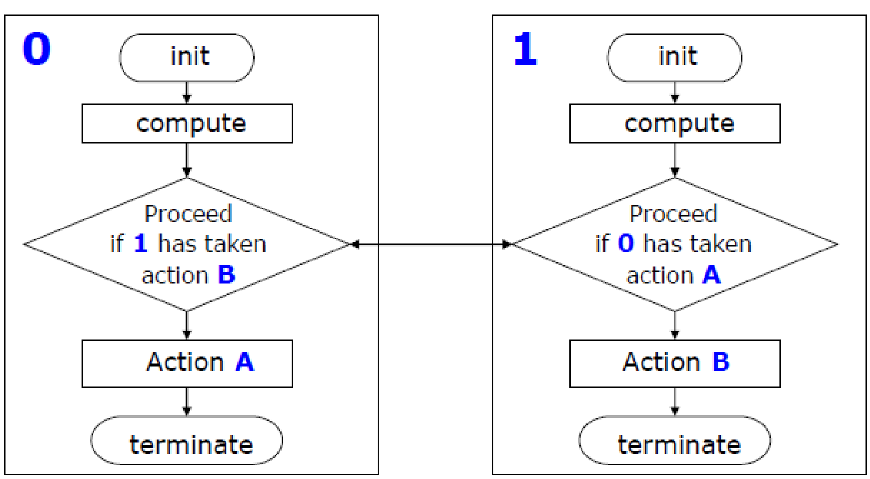
Blocking communications



Blocking communications

- A blocking send returns after it is safe to modify the application buffer (your send data) for reuse. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
- A blocking send can be synchronous
- A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
- A blocking receive only "returns" after the data has arrived and is ready for use by the program.

Deadlock in MPI



Send and Receive

- Blocking
  - MPI_Send does not complete until buffer is empty (available for reuse)
  - MPI_Recv does not complete until buffer is full (available for use)

```
int MPI_Send(void *buf, int count, MPI_Datatype
  type, int dest, int tag, MPI_Comm comm);

int MPI_Recv (void *buf, int count, MPI_Datatype
  type, int source, int tag, MPI_Comm comm,
  MPI_Status *status);
```

  - Simple, but can be “unsafe”

Send and Receive

(Solutions to the “Unsafe”)

- Order the operations more carefully
- Supply receive buffer at same time as send, with MPI_Sendrecv
- Non Blocking

```
int MPI_Isend(void *buf, int count,
  MPI_Datatype type, int dest, int tag,
  MPI_Comm comm, MPI_Request *req);

int MPI_Irecv (void *buf, int count,
  MPI_Datatype type, int source, int tag,
  MPI_Comm comm, MPI_Request *req);
```

MPI_Sendrecv

- **Synopsis**
  - int MPI_Sendrecv (const void *sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, int recvtag, MPI_Comm comm, MPI_Status *status)

- **Input Parameters**
  - **sendbuf** initial address of send buffer (choice)
  - **sendcount** number of elements in send buffer (integer)
  - **sendtype** type of elements in send buffer (handle)
  - **dest** rank of destination (integer)
  - **sendtag** send tag (integer)
  - **recvcount** number of elements in receive buffer (integer)
  - **recvtype** type of elements in receive buffer (handle)
  - **source** rank of source (integer)
  - **recvtag** receive tag (integer)
  - **comm** communicator (handle)

- **Output Parameters**
  - **recvbuf** initial address of receive buffer (choice)
  - **status** status object (Status). This refers to the receive operation.

Non Blocking communications

- Non-blocking send and receive routines will return almost immediately. They do not wait for any communication events to complete
- Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
- It is unsafe to modify the application buffer until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
- Non-blocking communications are primarily used to overlap computation with communication.

Non Blocking

Non-blocking operations return (immediately) “request handles” that can be waited on and queried

Waiting for completion

```
int MPI_Wait (MPI_Request *req, MPI_Status *status);
int MPI_Waitall (count, &array_of_requests, &array_of_statuses);
```

Testing for completion

```
int MPI_Test (&request, &flag, &status);
int MPI_Testall (count, &array_of_requests, &flag, &array_of_statuses);
```

Variations of Send and Receive

Mode	Completion Condition	Blocking subroutine	Non-blocking subroutine
Standard send	Message sent (receive state unknown)	MPI_SEND	MPI_ISEND
receive	Completes when a matching message has arrived	MPI_RECV	MPI_IRECV
Synchronous send	Only completes after a matching recv() is posted and the receive operation is started.	MPI_SSEND	MPI_ISSEND
Buffered send	Always completes, irrespective of receiver Guarantees the message being buffered	MPI_BSEND	MPI_IBSEND
Ready send	Always completes, irrespective of whether the receive has completed	MPI_RSEND	MPI_IRSEND

MPI: All to All

- This function makes a redistribution of the content of each process in a way that each process know the buffer of all others. It is a way to implement the matrix data transposition.

- *int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)*

a1	a2	a3	a4
b1	b2	b3	b4
c1	c2	c3	c4
d1	d2	d3	d4

a1	b1	c1	d1
a2	b2	c2	d2
a3	b3	c3	d3
a4	b4	c4	d4

Matrix Transposition

MPI_Alltoall(&send, 4, MPI_INT, &recv, 4, MPI_INT, MPI_COMM_WORLD);

```
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv )
{
   int send[4], recv[4];
   int rank, size, k;
   MPI_Init( &argc, &argv );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   MPI_Comm_size( MPI_COMM_WORLD, &size );
   if (size != 4) {
     printf("#processors must be equal to 4 programm aborting....");
     MPI_Abort(MPI_COMM_WORLD, 1);
   }
   for (k=0;k<size;k++) send[k] = (k+1) + rank*size;
   printf("%d : send = %d %d %d %d", rank, send[0], send[1], send[2], send[3]);
   MPI_Alltoall(&send, 4, MPI_INT, &recv, 4, MPI_INT, MPI_COMM_WORLD);
   printf("%d : recv = %d %d %d %d", rank, recv[0], recv[1], recv[2], recv[3]);
   MPI_Finalize();
   return 0;
}
```

MPI Reduction

Reduction operations permits us to

- Collect data from each process
- Reduce the data to a single value
- Store the result on the root process (MPI_Reduce) or
- Store the result on all processes (MPI_Allreduce)

MPI Reduction

MPI op	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location
MPI_MINLOC	Minimum and location

Communicator revisited

A **communicator** can be thought as a handle to a **group**.

- a group is a ordered set of processes
- each process is associated with a rank
- ranks are contiguous and start from zero

Groups allow collective operations to be operated on a subset of processes

The group routines are primarily used to specify which processes should be used to construct a communicator.

Communicator revisited

- **Intracommunicators :** These are used for communications within a single group
- **Intercommunicators:** These are used for communications between two disjoint groups

Communicator revisited

Group management:

All group operations are local (no communication is needed)

- Groups are not initially associated with communicators
- Groups can only be used for message passing within a communicator
- We can access groups, construct groups, destroy groups, i.e. groups/communicators are dynamic - they can be created and destroyed during program execution.

Communicator revisited

Using MPI Groups

1. Extract handle of global group from MPI_COMM_WORLD using MPI_Comm_group

2. Form new group as a subset of global group using MPI_Group_incl

3. Create new communicator for new group using MPI_Comm_create

4. Determine new rank in new communicator using MPI_Comm_rank

5. When finished, free up new communicator and group (optional) using MPI_Comm_free and MPI_Group_free

Communicator revisited

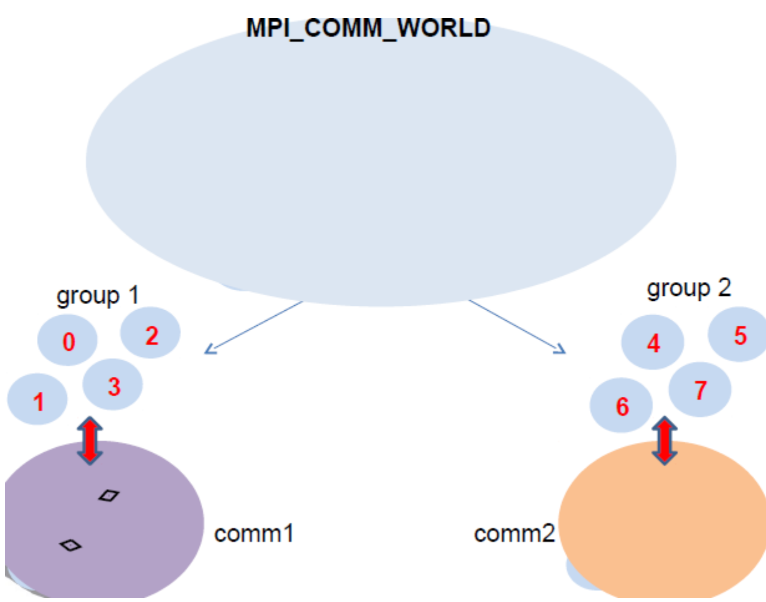
Group constructors

Group constructors are used to create new groups from existing ones (initially from the group associated with MPI_COMM_WORLD; you can use MPI_Comm_Group to get this).

Group creation is a local operation: no communication is needed.

After the creation of a group, no communicator has been associated to this group, and hence no communication is possible within the new group

Group Creation



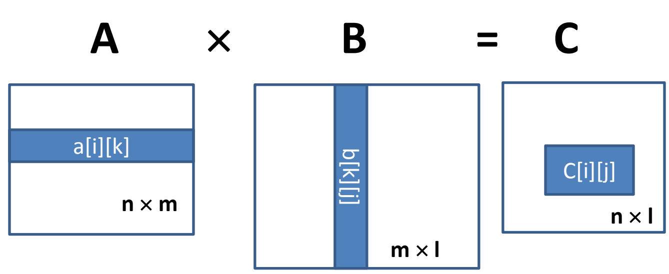
Matrix Multiplication

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}, 0 \le i < m, 0 \le j < l$$

$$\begin{pmatrix} 3 & 2 & 0 & -1 \\ 5 & -2 & 1 & 1 \\ 1 & 0 & -1 & -1 \end{pmatrix} \times \begin{pmatrix} 1 & -1 \\ 2 & 5 \\ -3 & 2 \\ 7 & 4 \end{pmatrix} = \begin{pmatrix} 0 & 3 \\ 5 & -9 \\ -3 & -7 \end{pmatrix}$$

```
for(i = 0; i<n; i++)
        for(j=0; k<l; j++)
                c[i][j]=0.0;
                for(k=0;k<m;k++)
                        c[i][j] = c[i][j] + a[i][k]*b[k][j];
                }
        }
}
```

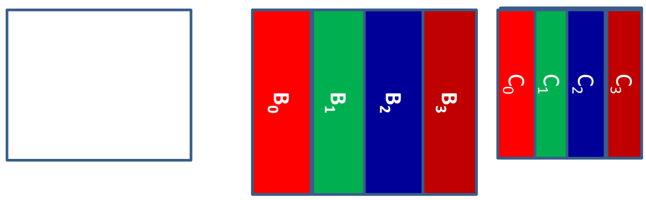
Matrix Multiplication




```
for(i = 0; i<n; i++)
        for(j=0; k<l; j++)
                c[i][j]=0.0;
                for(k=0;k<m;k++)
                        c[i][j] = c[i][j] + a[i][k]*b[k][j];
                }
        }
}
```

Matrix Multiplication

$$A \times (B_0|B_1|B_2|B_3) = (C_0|C_1|C_2|C_3)$$



Matrix Multiplication-1

```
int main (int argc, char *argv[])
{
      int numtasks,taskid,numworkers,source,dest,mtype,rows,averow, extra, offset;
      int i, j, k, rc;
      double a[NRA][NCA],b[NCA][NCB],c[NRA][NCB];
      MPI_Status status;

      MPI_Init(&argc,&argv);
      MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
      MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
      if (numtasks < 2 ) {
        printf("Need at least two MPI tasks. Quitting...\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
        exit(1);
      }
      numworkers = numtasks-1;
```

Matrix Multiplication-2

```
if (taskid == MASTER) {
    printf("Started with %d tasks.\n",numtasks);
    for (i=0; i<NRA; i++)
      for (j=0; j<NCA; j++)
        a[i][j]= i+j;
    for (i=0; i<NCA; i++)
      for (j=0; j<NCB; j++)
        b[i][j]= i*j;

    /* Send matrix data to the worker tasks */
    averow = NRA/numworkers;
    extra = NRA%numworkers;
    offset = 0;
    mtype = FROM_MASTER;
    for (dest=1; dest<=numworkers; dest++) {
      rows = (dest <= extra) ? averow+1 : averow;
      printf("Sending %d rows to task %d offset=%d\n",rows,dest,offset);
      MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
      MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
      MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,MPI_COMM_WORLD);
      MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
      offset = offset + rows;
    }
```

Matrix Multiplication-3

```
/* Receive results from worker tasks */
    mtype = FROM_WORKER;
    for (i=1; i<=numworkers; i++) {
      source = i;
      MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
      printf("Received results from task %d\n",source);
    }
    /* Print results */
    for (i=0; i<NRA; i++) {
      printf("\n");
      for (j=0; j<NCB; j++) printf("%6.2f   ", c[i][j]);
    }
    printf ("Done.\n");
```

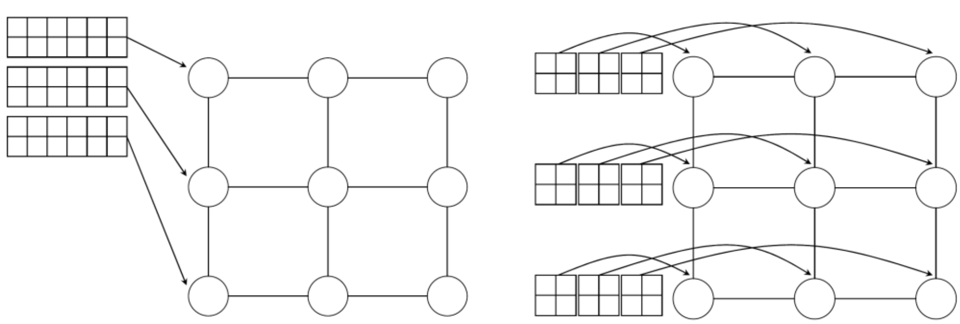
Matrix Multiplication-4

```
} else  {
/**************** worker task ************************************/
     mtype = FROM_MASTER;
     MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
     MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
     MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);
     MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &status);

     for (k=0; k<NCB; k++)
       for (i=0; i<rows; i++) {
         c[i][k] = 0.0;
         for (j=0; j<NCA; j++) c[i][k] = c[i][k] + a[i][j] * b[j][k];
       }
     mtype = FROM_WORKER;
     MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
     MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
     MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);
   }
   MPI_Finalize();
}
```

Combine

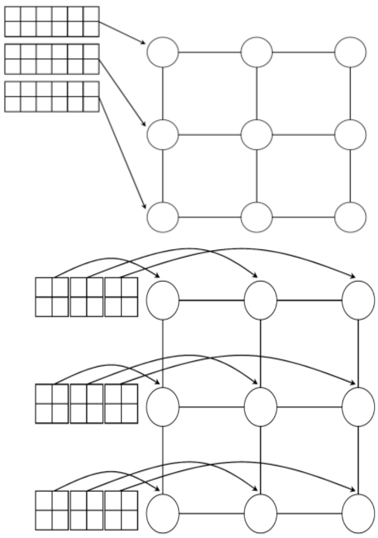
2D to 1D representation???

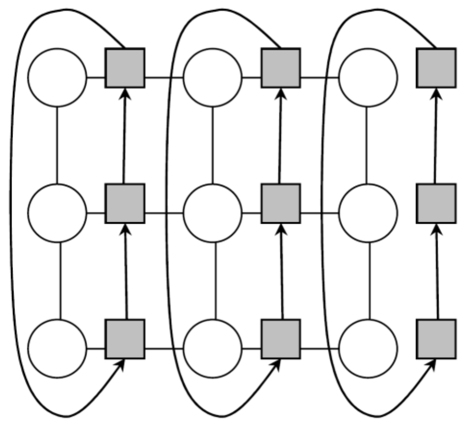
Matrix Multiplication



Step 1 **Step 2**

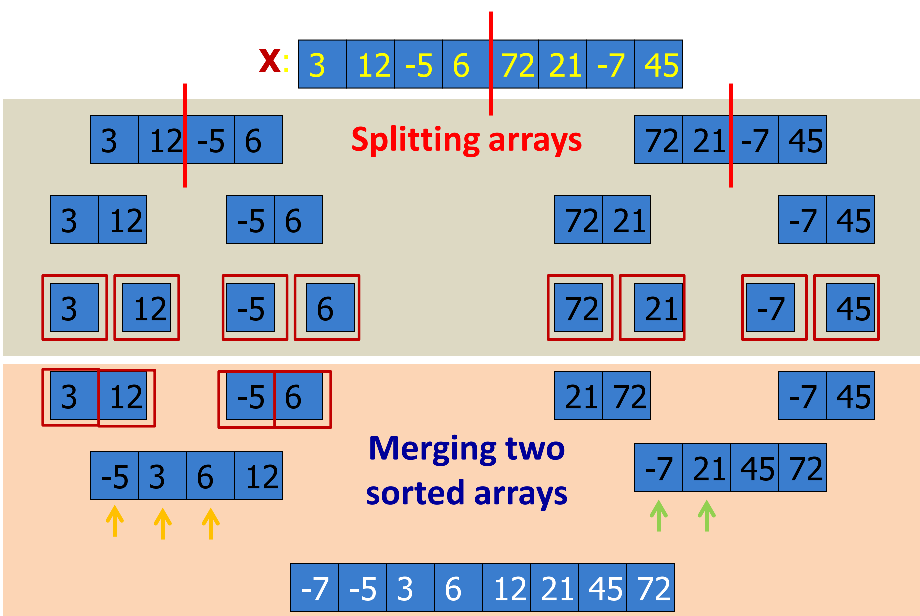
Matrix Multiplication





Workout: Implement

Merge Sort



Merge Sort C program

```
#include<stdio.h>
void mergesort(int a[],int i,int j);
void merge(int a[],int i1,int j1,int i2,int j2);

int main()
{
   int a[30],n,i;
   printf("Enter no of elements:");
   scanf("%d",&n);
   printf("Enter array elements:");

   for(i=0;i<n;i++)
      scanf("%d",&a[i]);

   mergesort(a,0,n-1);

   printf("\nSorted array is :");
   for(i=0;i<n;i++)
      printf("%d ",a[i]);

   return 0;
}
```

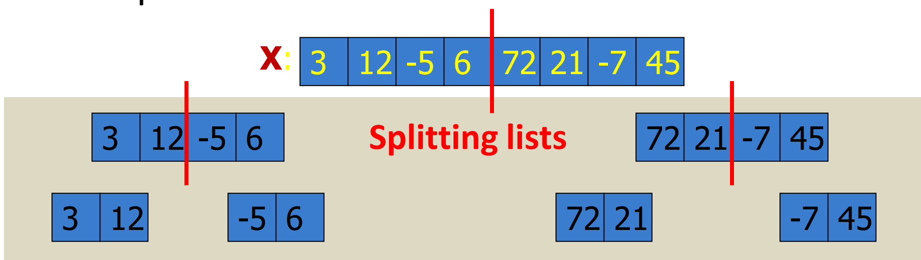
```
void mergesort(int a[],int i,int j)
{
   int mid;

   if(i<j)    {
      mid=(i+j)/2;
           /* left recursion  */
      mergesort(a,i,mid);
           /* right recursion */
      mergesort(a,mid+1,j);
/* merging of two sorted sub-arrays */
      merge(a,i,mid,mid+1,j);
    }
}
```

Merge Sort C program

```
void merge(int a[],int i1,int j1,int i2,int j2)
{
   int temp[50];    //array used for merging
   int i=i1,j=i2,k=0;

   while(i<=j1 && j<=j2)    //while elements in both lists
   {
      if(a[i]<a[j])
         temp[k++]=a[i++];
      else
         temp[k++]=a[j++];
   }

   while(i<=j1)    //copy remaining elements of the first list
      temp[k++]=a[i++];

   while(j<=j2)    //copy remaining elements of the second list
      temp[k++]=a[j++];

      for(i=i1,j=0;i<=j2;i++,j++)
      a[i]=temp[j];                //Transfer elements from temp[] back to a[]
}
```

Parallel Merge Sort

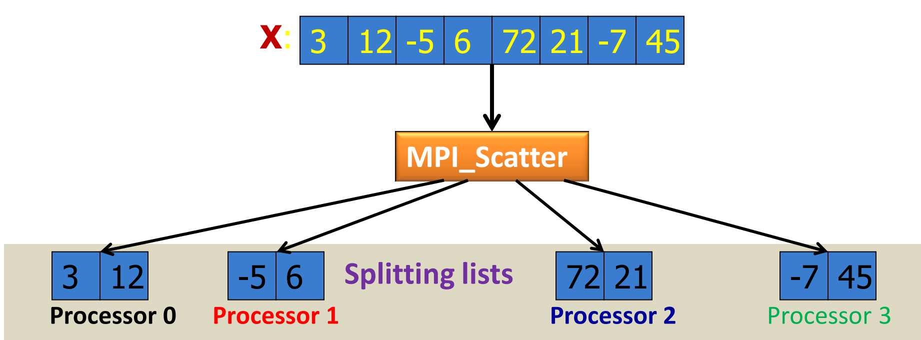
- Step 1: Divide list into unsorted sub-lists



- Step 2: Sort sub-lists
- Step 3: Merge sub-lists

Parallel Merge Sort

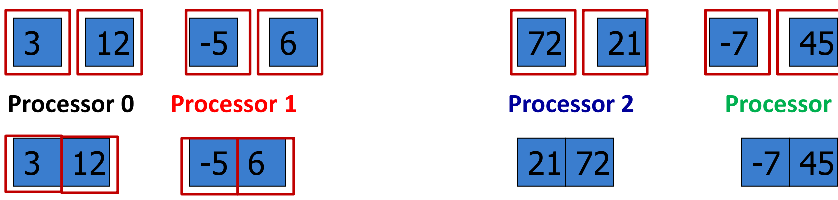
- Step 1: Divide list into unsorted sub-lists



- Step 2: Sort sub-lists
- Step 3: Merge sub-lists

Parallel Merge Sort

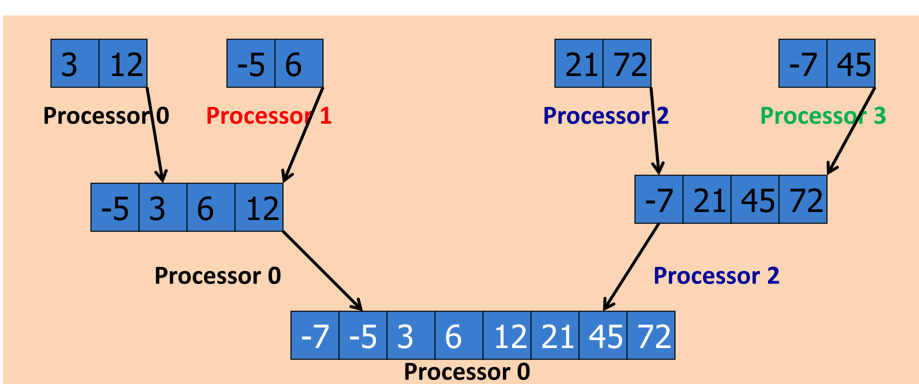
- Step 1: Divide list into unsorted sub-lists
- Step 2: Sort sub-lists



- Step 3: Merge sub-lists

Parallel Merge Sort

- Step 1: Divide list into unsorted sub-lists
- Step 2: Sort sub-lists
- Step 3: Merge sub-lists



MPI Sources

- The Standard:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Books:
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface* by Gropp, Lusk, and Skjellum.
  - *MPI: The Complete Reference* by Snir, Otto, Huss-Lederman, Walker, and Dongarra.
  - *Designing and Building Parallel Programs* by Ian Foster.
  - *Parallel Programming with MPI* by Peter Pacheco.