MPI PROGRAMMING

compiled by
Dr. N. Gopalakrishna Kini
Prof., Dept of CSE School of Computer Engg.
MIT, Manipal

What is MPI?

- Processes coordinate and communicate via calls to message passing library routines.
- Programmers “parallelize” algorithm and add message calls.
- MPI addresses primarily the message-passing parallel programming model.

Key Concepts of MPI

- Used to create parallel programs
  - Normally the same program is running.
  - Processors communicate using message passing.
  - No process can be created or terminated in the middle of program execution.

Introduction:

- autonomous processes.
- MIMD style.
- processes communicate via calls to MPI communication primitives.
- The major goal of MPI, as with most standards, is a degree of portability across different machines.
- Another type of compatibility offered by MPI is the ability to run transparently on heterogeneous systems.
- MPI allows or supports scalability.

MPI Program Organization

- MIMD Multiple Instruction, Multiple Data
- SPMD Single Program, Multiple Data

MPI Progam Organization

- MIMD in a SPMD framework

The message-passing programming model →each processor has a local memory to which it has exclusive access.

The number of processes is fixed when starting the program.

Each of the processes could execute a different program (MPMD).

Message Passing Work Allocation

- Manager Process
- Worker Process

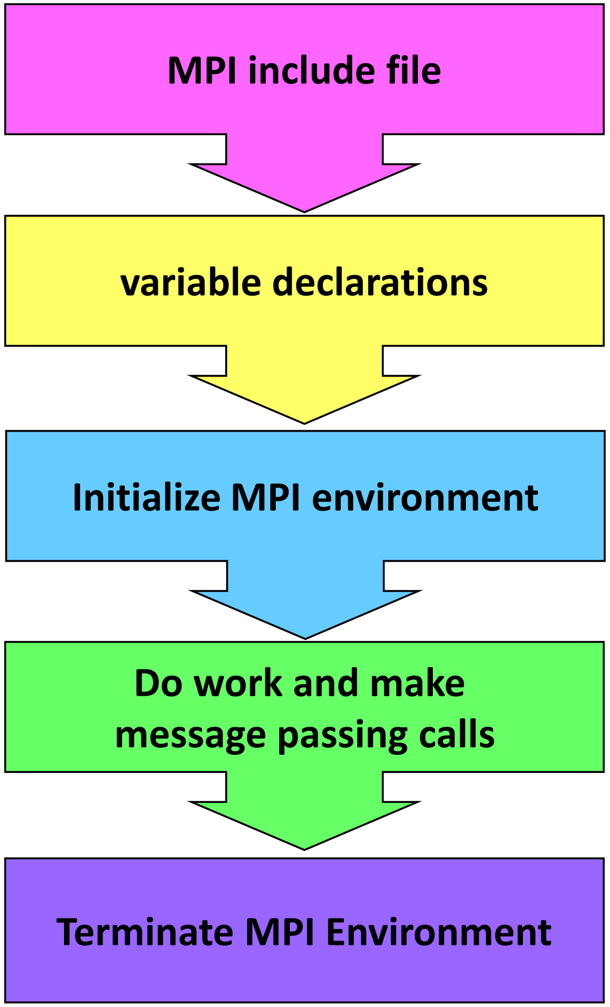
Message-passing program can exchange local data by using communication operations.

MPI consists of a collection of processes.

Processes in MPI are heavy-weighted and single threaded with separate address spaces.

On many parallel systems, an MPI program can be started from the command line.

General MPI Program Structure


MPI include file
variable declarations
Initialize MPI environment
Do work and make message passing calls
Terminate MPI Environment

```
#include <mpi.h>
void main (int argc, char *argv[])
{
int np, rank, ierr;
ierr = MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&np);
/*      Do Some Works            */
ierr = MPI_Finalize();
}
```

MPI Naming Conventions

- The names of all MPI entities (routines, constants, types, etc.) begin with **MPI_** to avoid conflicts.
- **C function names** have a mixed case:

  MPI_Xxxxx(parameter, ...)

  Example: MPI_Init(&argc, &argv).
- The names of **MPI constants** are all upper case in both C and Fortran, for example,

  MPI_COMM_WORLD, MPI_REAL, ...
- In C, specially defined types correspond to many MPI entities. Type names follow the C function naming convention above; for example,

  MPI_Comm

  is the type corresponding to an MPI "communicator".

MPI Routines and Return Values

- MPI routines are implemented as **functions** in C. In either case generally an error code is returned, enabling you to test for the successful operation of the routine.
- In C, MPI functions return an int, which indicates the exit status of the call.

```
int ierr;
…
ierr = MPI_Init(&argc, &argv);
…
```

MPI Routines and Return Values

- The error code returned is MPI_SUCCESS if the routine ran successfully (that is, the integer returned is equal to the pre-defined integer constant MPI_SUCCESS). Thus, you can test for successful operation with
- If an error occurred, then the integer returned has an implementation-dependent value indicating the specific error.

```
int rank;
MPI_Init((void *), (void *));
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0)
  printf("Starting program\n");
MPI_Finalize();
```

Special MPI Datatypes (C)

- In C, MPI provides several special datatypes (structures). Examples include
  - MPI_Comm - a communicator
  - MPI_Status - a structure containing several pieces of status information for MPI calls
  - MPI_Datatype
- These are used in variable declarations, for example,

```
MPI_Comm some_comm;
```

declares a variable called some_comm, which is of type MPI_Comm (i.e. a communicator).

Include files

- The MPI include file
  - C: mpi.h

Initializing MPI

- The first MPI routine called in any MPI program must be the initialization routine MPI_Init. This routine establishes the MPI environment, returning an error code if there is a problem.
- Note that the arguments to MPI_Init are the addresses of argc and argv.

Communicators

- A **communicator** is a handle representing a group of processors that can communicate with one another.
- The **communicator name** is required as an argument.

Communicators

- There can be many communicators, and a given processor can be a member of a number of different communicators.

Getting Communicator Information: Rank

- A processor can determine its rank in a communicator with a call to MPI_Comm_rank.
  - The argument comm is a variable of type MPI_Comm, a communicator.
  - Note that the second argument is the address of the integer variable rank.

Getting Communicator Information: Size

- A Communicator can also determine the size, or number of processors, of any communicator to which it belongs with a call to MPI_Comm_size.
  - The argument comm is of type MPI_Comm, a communicator.
  - Note that the second argument is the address of the integer variable size.

Terminating MPI

- The last MPI routine called should be MPI_Finalize which
  - cleans up all MPI data structures, cancels operations that never completed, etc.
  - must be called by all processes; if any one process does not reach this statement, the program will appear to hang.
- Once MPI_Finalize has been called, no other MPI routines (including MPI_Init) may be called.

Sample Program: Hello World!

- In this "Hello World" program, each processor prints its rank as well as the total number of processors in the communicator MPI_COMM_WORLD.
- Notes:
  - Makes use of the pre-defined communicator MPI_COMM_WORLD.
  - Not testing for error status of routines!

Sample Program: Hello World!

```
#include <stdio.h>
#include <mpi.h>
void main (int argc, char *argv[]) {
   int myrank, size;

   /* Initialize MPI */
   MPI_Init(&argc, &argv);

   /* Get my rank */
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

   /* Get the total number of processors */
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   printf("Processor %d of %d: Hello World!\n",
     myrank, size);

   MPI_Finalize(); /* Terminate MPI */
}
```

Sample Program: Output

- Running this code on four processors will produce a result like:

```
Processor 2 of 4: Hello World!
Processor 1 of 4: Hello World!
Processor 3 of 4: Hello World!
Processor 0 of 4: Hello World!
```

- Each processor executes the same code, including probing for its rank and size and printing the string.

Summary

- MPI is used to create parallel programs based on message passing
- Usually the same program is run on multiple processors
- The 6 basic calls in MPI are:
  - `MPI_INIT( ierr )`
  - `MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )`
  - `MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )`
  - `MPI_Send(buffer, count,MPI_INTEGER,destination, tag, MPI_COMM_WORLD, ierr)`
  - `MPI_Recv(buffer, count, MPI_INTEGER,source,tag, MPI_COMM_WORLD, status,ierr)`
  - `call MPI_FINALIZE(ierr)`

Point-to-Point Communication

A communicator → communication domain → a set of processes that exchange messages between each other.

MPI default communicator MPI_COMM_WORLD is used.

Basic form of data exchange between processes is provided by point-to-point communication.

```
int MPI_Send(void *smessage,
             int count,
             MPI_Datatype datatype,
             int dest,
             int tag,
             MPI_Comm comm)
```

To receive a message, a process executes the following operation:

```
int MPI_Recv(void *rmessage,
                  int count,
                  MPI_Datatype datatype,
                  int source,
                  int tag,
                 MPI_Comm comm,
                 MPI_Status *status)
```

Predefined data types for MPI

MPI_Datatype	**C-Data type**
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_WCHAR	wide char
MPI_PACKED	special data type for packing
MPI_BYTE	single byte value

Some semantic terms that are used for the description of MPI operations:

Blocking operation
Non-blocking operation

The terms *blocking* and *non-blocking* describe the behavior of operations from the *local* view of the executing process, without taking the effects on other processes into account.
Synchronous and *asynchronous c*ommunications:

BLOCKING OPERATION :

i) Standard Mode : MPI_Send and MPI_Recv

ii) Synchronous mode : MPI_Ssend and MPI_Recv

iii) Buffered Mode : MPI_Bsend and MPI_Recv

NON-BLOCKING OPERATION :

i) Standard Mode : MPI_ISend and MPI_Irecv

ii) Synchronous mode: MPI_Issend and MPI_Irecv

iii) Buffered Mode: MPI_Ibsend and MPI_Irecv

P2P: Blocking Send/Recv

- Waits to return until the message has been received by the destination process
- This synchronizes the sender with the receiver
- Perform a standard mode blocking send and standard mode blocking receive, resp.
- The receive can be started before the corresponding send is initiated.
- Receive will only return after message data is stored in receive buffer.
- The send can be started whether or not the corresponding receive has been posted.

Example

- Always succeeds, even if no buffering is done.

```
if(rank==0)
{
    MPI_Send(...);
    MPI_Recv(...);
}
else if(rank==1)
{
    MPI_Recv(...);
    MPI_Send(...);
}
```

Example

- Will always deadlock, no matter the buffering mode.

```
if(rank==0)
{
   MPI_Recv(...);
   MPI_Send(...);
}
else if(rank==1)
{
   MPI_Recv(...);
   MPI_Send(...);
}
```

Example

- Only succeeds if sufficient buffering is present -- strictly unsafe!

```
if(rank==0)
{
   MPI_Send(...);
   MPI_Recv(...);
}
else if(rank==1)
{
   MPI_Send(...);
   MPI_Recv(...);
}
```

Standard Send-Recv:

- Order of send from source process (Send-Send) to receive at target process (Recv-Recv) is maintained.

  NO DEADLOCK.

Order of receive at target process (Recv-Recv) is NOT in order of send (Send-Send). NO DEADLOCK.

Order at Process 0 (Send-Recv) and at Process 1 (Recv-Send) maintained.

NO DEADLOCK.

Order at Process 0 (Send-Recv) and at Process 1 (Send-Recv).

NO DEADLOCK.

Order at Process 0 (Recv-Send) and at Process 1 (Recv-Send).

DEADLOCK OCCURED

Some semantic terms that are used for the description of MPI operations:

Blocking operation
Non-blocking operation

The terms *blocking* and *non-blocking* describe the behavior of operations from the *local* view of the executing process, without taking the effects on other processes into account.
Synchronous and *asynchronous c*ommunications:

Buffering in MPI

- Implementation may buffer on sending process, receiving process, both, or none.
- In practice, tend to buffer "small" messages on receiving process.
- MPI has a buffered send-mode:
  - `MPI_Buffer_attach`
  - `MPI_Buffer_detach`
  - `MPI_Bsend`

Avoiding Deadlocks

Using non-blocking operations remove most deadlocks. Consider:

```
int a[10], b[10], myrank;
MPI_Status status;
...
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) {
   MPI_Send(a, 10, MPI_INT, 1, 1, MPI_COMM_WORLD);
   MPI_Send(b, 10, MPI_INT, 1, 2, MPI_COMM_WORLD);
}
else if (myrank == 1) {
   MPI_Recv(b, 10, MPI_INT, 0, 2, &status, MPI_COMM_WORLD);
   MPI_Recv(a, 10, MPI_INT, 0, 1, &status, MPI_COMM_WORLD);
}
...
```

Replacing either the send or the receive operations with non-blocking counterparts fixes this deadlock.

Collective Communication and Computation Operations

- MPI provides an extensive set of functions for performing common collective communication operations.
- Each of these operations is defined over a group corresponding to the communicator.
- All processors in a communicator must call these operations.

MPI: Collective Communications

- Collective communications transmit (and possibly operate on) data among all processes in a given communications group.
- Barrier (synchronization), global communications, global reduction operations.

MPI_Barrier (barrier synchronization operation)

```
int MPI_Barrier(MPI_Comm comm)
```

- blocks the caller until all group members have called it.
- syncs all processes in a group to some known point.

MPI_Barrier (barrier synchronization operation)

Stop processes until all processes within a communicator reach the barrier.
Almost never required in a parallel program
Occasionally useful in measuring performance and load balancing.

int MPI_Barrier(MPI_Comm comm)

MPI: Global Communications

- only come in blocking mode calls .
- no tag provided, messages are matched by order of execution within the group.
- intercommunicators are not allowed.
- you cannot match these calls with P2P receives.

Collective Message Passing w/MPI

MPI_Bcast()	Broadcast from root to all other processes
MPI_Reduce()	Combine values on all processes to single val
MPI_Scatter()	Scatters buffer in parts to group of processes
MPI_Gather()	Gather values for group of processes
MPI_Alltoall()	Sends data from all processes to all processes
MPI_Allgather()	
MPI_Allreduce()	
MPI_Reduce_Scatter()	Broadcast from root to all other processes

MPI: Broadcast (One-to-all)

- int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
  - One process (root) sends data to all the other processes in the same communicator
  - Must be called by all the processes with the same arguments

MPI: Reduction Operations

- Perform global reduce operations across all members of a group.
- Many predefined operations come with MPI.
- Ability to define your own operations.

MPI: Reduce (all-to-one reduction operation)

- int MPI_Reduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  - One process (root) collects data to all the other processes in the same communicator, and performs an operation on the data
  - MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
  - returns combined value in the output buffer

Predefined Reduction Operations

Operation	Meaning	Datatypes
MPI_MAX	Maximum	C integers and floating point
MPI_MIN	Minimum	C integers and floating point
MPI_SUM	Sum	C integers and floating point
MPI_PROD	Product	C integers and floating point
MPI_LAND	Logical AND	C integers
MPI_BAND	Bit-wise AND	C integers and byte
MPI_LOR	Logical OR	C integers
MPI_BOR	Bit-wise OR	C integers and byte
MPI_LXOR	Logical XOR	C integers
MPI_BXOR	Bit-wise XOR	C integers and byte
MPI_MAXLOC	max-min value -location	Data-pairs
MPI_MINLOC	min-min value-location	Data-pairs

MPI: Gather

Gather

- int MPI_Gather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)
  - One process (root) collects data to all the other processes in the same communicator (i.e each process in *comm* (including *root* itself) sends its *sendbuf* to *root*.)
  - the *root* process receives the messages in *recvbuf* in rank order.
  - Must be called by all the processes with the same arguments

MPI: Scatter

```
int MPI_Scatter(void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- inverse to MPI_Gather.
- *sendbuf* is ignored by all non-*root* processes.

MPI: Allgather

Gather to All

- int MPI_Allgather(void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, MPI_Comm comm)
  - MPI also provides the MPI_Allgather function in which the data are gathered at all the processes.
  - All the processes collects data to all the other processes in the same communicator.
  - *recvbuf* is NOT ignored.
  - Must be called by all the processes with the same arguments

MPI: Allreduce

Reduction to All

- int MPI_Allreduce(void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
  - All the processes collect data to all the other processes in the same communicator, and perform an operation on the data
  - MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
  - MPI_Op_create(): User defined operator

- If the result of the reduction operation is needed by all processes, MPI provides MPI_Allreduce

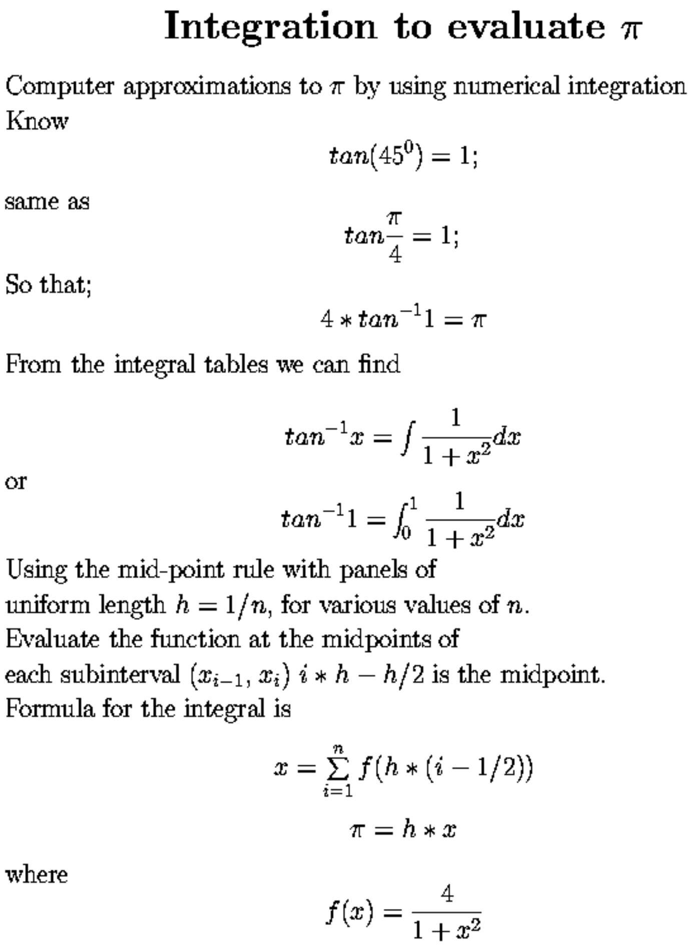
MPI: Alltoall

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

- similar to MPI_Allgather except each process sends distinct data to each of the receivers.
- the j^{th} block sent from process i is received by process j and placed in the i^{th} block of *recvbuf.*

Approximation of Pi

Compute π value using p processors.



Integration to evaluate π

Computer approximations to π by using numerical integration
Know

$$tan(45^0) = 1;$$

same as

$$tan\frac{\pi}{4} = 1;$$

So that;

$$4 * tan^{-1} 1 = \pi$$

From the integral tables we can find

$$tan^{-1} x = \int \frac{1}{1+x^2} dx$$

or

$$tan^{-1} 1 = \int_0^1 \frac{1}{1+x^2} dx$$

Using the mid-point rule with panels of
uniform length $h = 1/n$, for various values of n.
Evaluate the function at the midpoints of
each subinterval (x_{i-1}, x_i) $i * h - h/2$ is the midpoint.
Formula for the integral is

$$x = \sum_{i=1}^{n} f(h * (i - 1/2))$$

$$\pi = h * x$$

where

$$f(x) = \frac{4}{1+x^2}$$

Example: PI in C -1

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
  int done = 0, n, myid, numprocs, i, rc;
  double PI25DT = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, a;
  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;
```

Example: PI in C - 2

```
    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;
}
```

MPI: Scan

```
int MPI_Scan(void *sendbuf, void *recvbuf, int count,
          MPI_Datatype datatype, MPI_Op op,
                    MPI_Comm comm)
```

TIME CALCULATION:

The elapsed (wall-clock) time between two points in an MPI program can be computed using MPI_Wtime

MPI_Wtime

Returns an elapsed time on the calling processor

```
double MPI_Wtime( void );
```

Return value

Time in seconds.

Remarks

MPI_WTIME returns a floating-point number of seconds.

The times returned are local to the node that called them.

```
#include "mpi.h"
#include <windows.h>
#include <stdio.h>
#include<conio.h>
int main( int argc, char *argv[] )
{
   double t1, t2;
   MPI_Init( 0, 0 );
    t1 = MPI_Wtime();
    Sleep(1000);
    t2 = MPI_Wtime();
  printf("MPI_Wtime measured a 1 second sleep to be:
                                                   %1.2f\n", t2-t1);
    fflush(stdout);
    getch();                MPI_Finalize( );
    return 0;      }
```

Handling MPI Errors

The error handler is called every time an MPI error is detected within the communicator.

There is a predefined error handler, which is called **MPI_ERRORS_RETURN**.

error handler can be used by calling function **MPI_Errhandler_set**.

```
MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
```

Once you've done this in your MPI code, the program will no longer abort on having detected an MPI error, instead the error will be returned and you will have to handle it.

Handling MPI Errors (Contd..)

The returned error code is implementation specific. The only error code that MPI standard itself defines is **MPI_SUCCESS**, i.e., no error.

MPI standard defines so called *error classes*. Every error code, must belong to some error class, and the error class for a given error code can be obtained by calling function **MPI_Error_class**. Error classes can be converted to comprehensible error messages by calling MPI_Error_string. Meaning of an error code can be extracted by calling function **MPI_Error_string**.

```
#include "mpi.h"
  #include <stdio.h>
#include<conio.h>
          void ErrorHadler(int error_code)
          {
          }
  int main(int argc,char *argv[])
  {
          int  C=3;
          int  numtasks, rank, len, error_code;
          error_code = MPI_Init(&argc,&argv);
          MPI_Errhandler_set(MPI_COMM_WORLD, MPI_ERRORS_RETURN
          MPI_Comm_rank(MPI_COMM_WORLD,&rank);
          error_code = MPI_Comm_size(C, &numtasks);
          ErrorHadler(error_code);
          printf ("Number of tasks= %d My rank= %d \n", numtasks,rank);
          MPI_Finalize();
  }
```

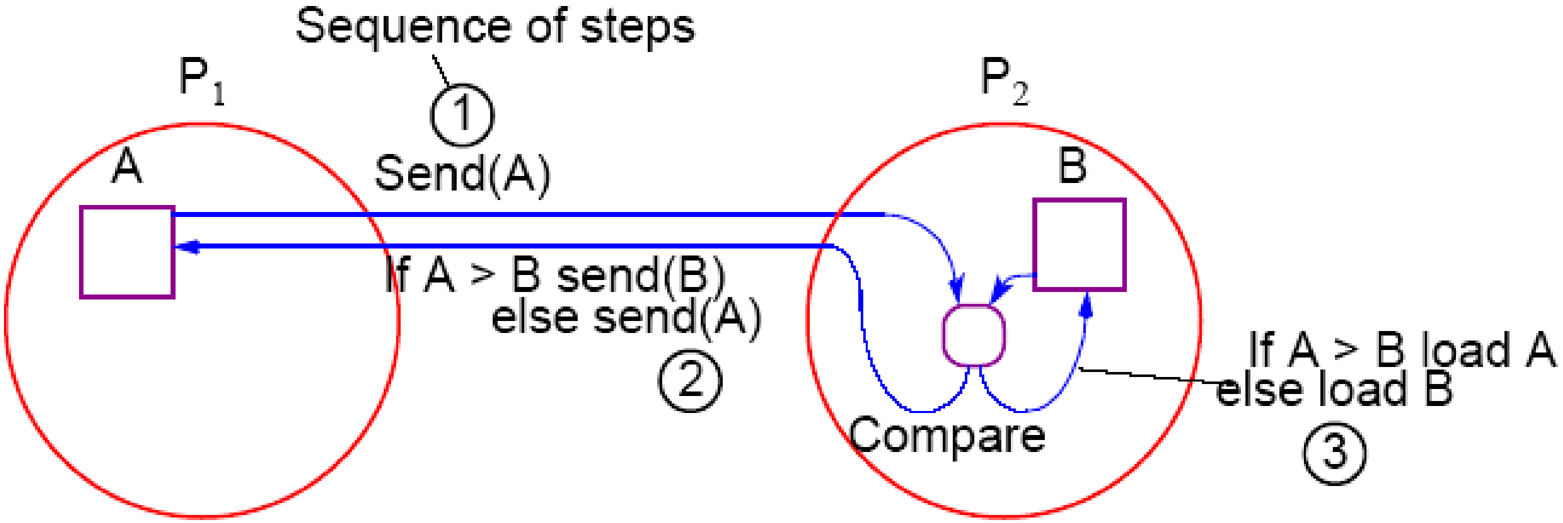
```
#include "mpi.h"
#include <stdio.h>
#include<conio.h>
    void ErrorHadler(int error_code)
    {
      if (error_code != MPI_SUCCESS)
        {
          char error_string[BUFSIZ];
          int length_of_error_string, error_class;
          MPI_Error_class(error_code, &error_class);
          MPI_Error_string(error_class, error_string, &length_of_error_strin
          fprintf(stderr, "  %s %d\n",  error_string,length_of_error_string);
          MPI_Error_string(error_code, error_string, &length_of_error_strin
          fprintf(stderr, "HELLO_ERRORCODE %s\n",  error_string);

         }
    }
```

Message-Passing Compare and Exchange

Version 1

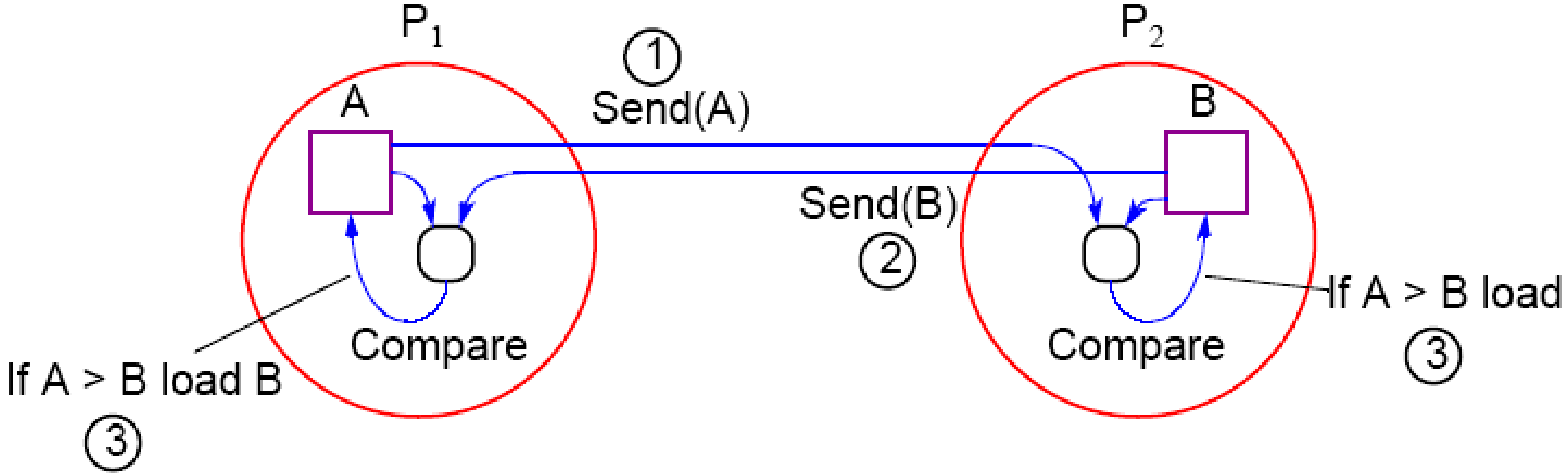
*P*1 sends *A* to *P*2, which compares *A* and *B* and sends back *B* to *P*1 if *A* is larger than *B* (otherwise it sends back *A* to *P*1):



Alternative Message Passing Method

Version 2

For *P*1 to send *A* to *P*2 and *P*2 to send *B* to *P*1. Then both processes perform compare operations. *P*1 keeps the larger of *A* and *B* and *P*2 keeps the smaller of *A* and *B*:



ODD-EVEN TRANSPOSITION SORTING:-

This is designed for processor array model in which the processing elements are organized in one-dimensional mesh.

Let A=(a_0,a_1,….,a_{n-1}) is the set of n elements to be sorted.

Each PE (totally *n* PEs) contain two local variables :*a* and *t* ; *a* unique element of array A and a variable *t* containing a value retrieved from a neighboring PE.

Algorithm performs (*n* / 2) iterations and each iteration will have two phases:

1st Phase: called **odd-even exchange**, value of *a* in every odd numbered processor (except processor *n*-1) is compared with the value of *a* stored in successor processor.

Values are exchanged if lower numbered processor contains the larger value.

2nd Phase: called **even-odd exchange**, value of *a* in every even numbered processor is compared with the value of *a* stored in successor processor. Values are exchanged if lower numbered processor contains the larger value.

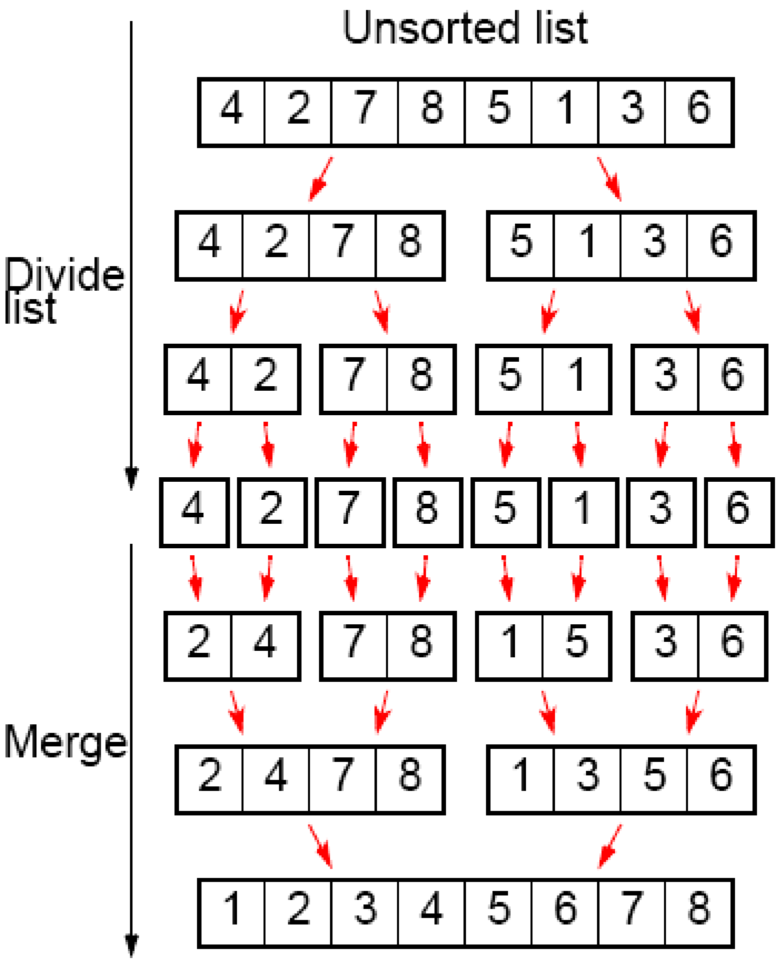
After *n*/2 iterations the values are observed to be sorted.

Odd-even transposition sort(one dimensional mesh processor array):

```
parameter n
global i
local a,t
{
for(i=1; i ≤ n/2; i++)
    { for all $P_j$ , where 0 ≤ j ≤ n-1 do
       {
             if j <(n-1) and odd(j) then
             {
                 t←successor(a);
                 successor(a)←max(a,t);
                 a ←min(a,t);
             }
             if even(j) then
             {
                 t←successor(a);
                 successor(a)←max(a,t);
                 a ←min(a,t);
             }
     }
}  }
```

Parallelizing Mergesort

Using tree allocation of processes



Unsorted list

4 2 7 8 5 1 3 6

4 2 7 8 | 5 1 3 6

4 2 | 7 8 | 5 1 | 3 6

4 | 2 | 7 | 8 | 5 | 1 | 3 | 6

2 4 | 7 8 | 1 5 | 3 6

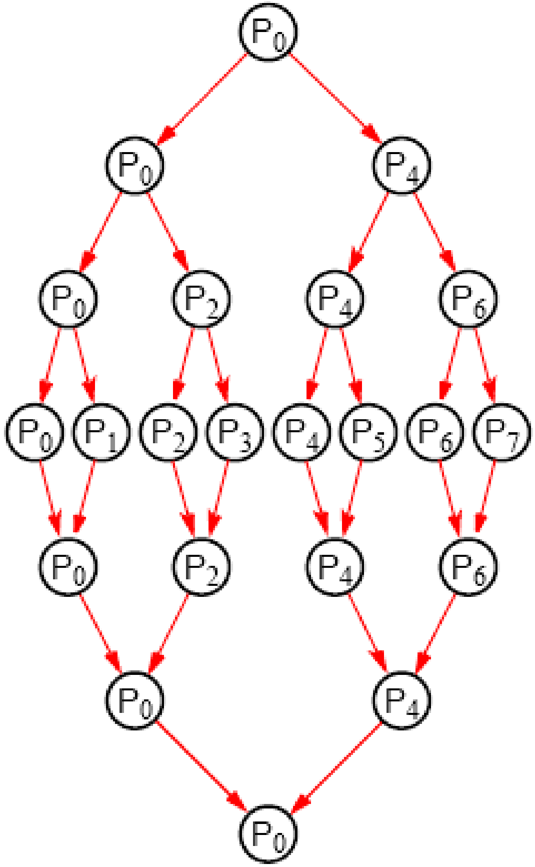
2 4 7 8 | 1 3 5 6

1 2 3 4 5 6 7 8

Divide list

Merge

Sorted list



Process allocation

Matrix Multiplication in MPI