Lecture 6: Message Passing Interface

- Introduction
- The basics of MPI
- Some simple problems
- More advanced functions of MPI
- A few more examples
When is Parallel Implementation Useful

• In general it is useful for Large problems
• Problems suitable for parallelisation, i.e. you know what speed-up to expect
• You need to be able to recognise them
• Three types of problems are suitable:
  – Parallel Problems
  – Regular and Synchronous Problems
  – Irregular and/or Asynchronous Problems
When is Parallel Implementation Useful: Type I

• Parallel problems:
  – The problem can be broken down into subparts
  – Each subpart is independent of the others
  – No communication is required, except to split up the problem and combine the final results
  – Linear speed-up can be expected

• Example of this is: Monte-Carlo simulations
When is Parallel Implementation Useful: Type II

• Regular and Synchronous Problems:
  – Same instruction set (regular algorithm) applied to all data
  – Synchronous communication (or close to): each processor finishes its task at the same time
  – Local (neighbour to neighbour) and collective (combine final results) communication

• Speed-up based on the computation:communication ratio

• If it is large, expect good speed-up for local communications & ok speed-up for non-local communications

• Ex: Fast Fourier transforms (synchronous), matrix-vector products, sorting (loosely synch.)
When is Parallel Implementation Useful: Type III

• Irregular and/or Asynchronous Problems:
  – Irregular algorithm which cannot be implemented efficiently except with message passing and high communication overhead
  – Communication is usually asynchronous and requires careful coding and load balancing
  – Often dynamic repartitioning of data between processors is required
  – Speed-up is difficult to predict; if the problem can be split up into regular and irregular parts, this makes things easier

• Ex: Melting ice problem (or any moving boundary simulation)
Example 1: matrix-vector product

\[
\begin{pmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{pmatrix}
\times
\begin{pmatrix}
b_1 \\
b_2 \\
b_3 \\
b_4
\end{pmatrix}
= 
\begin{pmatrix}
c_1 \\
c_2 \\
c_3 \\
c_4
\end{pmatrix}
\]

with

\[
\begin{align*}
c_1 &= a_{11} \times b_1 + a_{12} \times b_2 + a_{13} \times b_3 + a_{14} \times b_4 \\
c_2 &= a_{21} \times b_1 + a_{22} \times b_2 + a_{23} \times b_3 + a_{24} \times b_4 \\
c_3 &= a_{31} \times b_1 + a_{32} \times b_2 + a_{33} \times b_3 + a_{34} \times b_4 \\
c_4 &= a_{41} \times b_1 + a_{42} \times b_2 + a_{43} \times b_3 + a_{44} \times b_4
\end{align*}
\]

• A parallel approach:
  • Each element of vector \( c \) depends on vector \( b \) and only one line of \( A \)
  • Each element of \( c \) can be calculated independently from the others
  • Communication only needed to split up the problem and combine the final results
  • \( \Rightarrow \) a linear speed-up can be expected for large matrices
Example 2: Monte-Carlo calculation of Pi

- $\pi = 3.14159\ldots$ = area of a circle of radius 1
- $\pi/4 \approx$ fraction of the points within the circle quadrant
- The more points, the more accurate the value for $\pi$ is
Example 2: Monte-Carlo calculation of Pi (Cont’d)

• A parallel approach:
  – Each point is randomly placed within the square
  – The position of each point is independent of the position of the others
  – We can split up the problem by letting each node randomly place a given number of points
  – Communication is only needed to specify the number of points and combine final results

• => a linear speed-up can be expected, allowing for a larger number of points and therefore a greater accuracy in the estimation of π.
Example 3: A More Real Problem

- After each move, the chess software must find the best move within a set =>This set is large, but finite
- Each move from this set can be evaluated independently & the set can be partitioned
- Communication is only needed to split up the problem and combine the final results
- => A linear speed-up can be expected
- => This means that, in a reasonable time, moves can be studied more thoroughly
- => This depth of evaluation is what makes the software more competitive
Some background on MPI

- Developed by MPI forum (made up of Industry, Academia and Govt.)
- They established a standardised Message-Passing Interface (MPI-1) in 1994
- It was intended as an interface to both C and FORTRAN.
- C++ bindings were deprecated in MPI-2. Some Java bindings exist but are not standard yet.
- Aim was to provide a specification which can be implemented on any parallel computer or cluster; hence portability of code was a big aim.
Advantages of MPI

+ Portable, hence protection of software investment
+ A standard, agreed by everybody
+ Designed using optimal features of existing message-passing libraries
+ “Kitchen-sink” functionality, very rich environment (129 functions)
+ Implementations for F77, C and C++ are freely downloadable

& It’s Disadvantages

− “Kitchen-sink” functionality, makes it hard to learn all (unnecessary: a bare dozen are needed in most cases)
− Implementations on shared-memory machines is often quite poor, and does not suit the programming model
− Has rivals in other message-passing libraries (e.g. PVM)
MPI Preliminaries...

• MPI provides support for:
  – Point-to-point & collective (i.e. group) communications
  – Inquiry routines to query the environment (how many nodes are there, which node number am I, etc.)
  – Constants and data-types

• We will start with the basics: initialising MPI, and using point-to-point communication
MPI Preliminaries... (Cont’d)

• Naming convention
  – All MPI identifiers are prefixed by ‘\texttt{MPI}_’.
  – C routines contain lower case (i.e. ‘\texttt{MPI\_Init}’),
  – Constants are all in upper case (e.g. ‘\texttt{MPI\_FLOAT}’ is an MPI C data-type).
  – C routines are actually integer functions which return a status code (you are strongly advised to check these for errors!).

• Running MPI
  – Number of processors used is specified in the command line, when running the MPI loader that loads the MPI program onto the processors, to avoid hard-coding this into the program
  – e.g. \texttt{mpirun -np N exec}
MPI Preliminaries... (Cont’d)

• Writing a program using MPI: what is parallel, what is not
  – Only one program is written. By default, every line of the code is executed by each node running the program.
  – E.g. if the code contains `int result = 0`, each node will locally create a variable and assign the value.
• When a section of the code needs to be executed by only a subset of nodes, it has to be explicitly specified.
• E.g., providing that we are using 8 nodes, and that `MyID` is a variable storing the rank of the node (from 0 to 7, we will see how to get it later), this section of code assigns to `result` to zero for the first half of them, and 1 for the second.

```c
int result;
if(MyID < 4) result = 0;
else result = 1;
```
Common MPI Routines

- MPI has a ‘kitchen sink’ approach of 129 different routines
- Most basic programs can get away with using six.
- As usual use \#include “mpi.h” in C.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Init</td>
<td>Initialise MPI computation</td>
</tr>
<tr>
<td>MPI_Finalize</td>
<td>Terminate MPI computation</td>
</tr>
<tr>
<td>MPI_Comm_size</td>
<td>Determine number of processes</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>Determine my process number</td>
</tr>
<tr>
<td>MPI_Send, MPI_Isend</td>
<td>Blocking, non-blocking send</td>
</tr>
<tr>
<td>MPI_Recv, MPI_Irecv</td>
<td>Blocking, non-blocking</td>
</tr>
</tbody>
</table>
Common MPI Routines (Cont’d): MPI Initialisation, Finalization

• In all MPI-written programs, MPI must be initialised before use, and finalised at the end.

• All MPI-related commands and types must be handled within this section of code:

```c
MPI_Init ...

MPI_Finalize
```

Initialise MPI computation

Terminate MPI computation

• **MPI_Init** takes two parameters as input (`argc` and `argv`),
  – It is used to start the MPI environment, create the default communicator (more later) and assign a rank to each node.

• **MPI_Finalize** cleans up all MPI state. Once this routine is called, no MPI routine (even `MPI_Init`) may be called.

• The user must ensure that all pending communications involving a process completes before the process calls `MPI_Finalize`. 
Common MPI Routines (Cont’d): Basic Inquiry Routines

• At various stages in a parallel-implemented function, it may be useful to know how many nodes the program is using, or what the rank of the current node is.

• The `MPI_Comm_size` function returns the number of processes/nodes as an integer, taking only one parameter, a communicator.

• In most cases you will only use the default Communicator: `MPI_COMM_WORLD`.

• The `MPI_Comm_rank` function is used to determine what the rank of the current process/node on a particular communicator.

• E.g. if there are two communicators, it is possible, and quite usual, that the ranks of the same node would differ.

• Again, in most cases, this function will only be used with the default communicator as an input (`MPI_COMM_WORLD`), and it will return (as an integer) the rank of the node on that communicator.
Common MPI Routines (Cont’d):
Point-to-Point communications in MPI

• This involves communication between two processors, one sending, and the other receiving.
• Certain information is required to specify the message:
  – Identification of sender processor
  – Identification of destination/receiving processor
  – Type of data (MPI_INT, MPI_FLOAT etc)
  – Number of data elements to send (i.e. array/vector info)
  – Where the data to be sent is in memory (pointer)
  – Where the received data should be stored in (pointer)
Common MPI Routines (Cont’d):
Sending data **MPI_Send, MPI_Isend**

- **MPI_Send** is used to perform a blocking send, (i.e. process waits for the communication to finish before going to the next command).

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

- This functions takes six parameters:
  - the location of the data to be sent i.e. a pointer (*input parameter*)
  - the number of data elements to be sent (*input parameter*)
  - the type of data e.g. **MPI_INT, MPI_FLOAT**, etc. (*input parameter*)
  - the rank of the receiving/destination node (*input parameter*)
  - a tag for identification of the communication (*input parameter*)
  - the communicator to be used for transmission (*input parameter*)

- **MPI_Isend** is non-blocking, so an additional parameter, to allow for verification of communication success is needed.

- It is a pointer to an element of type **MPI_Request**.
Common MPI Routines (Cont’d):
Receiving data MPI_Recv, MPI_Irecv

- **MPI_Recv** is used to perform a blocking receive, (i.e. process waits for the communication to finish before going to the next command).

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

- This functions takes seven parameters:
  - the location of the receive buffer i.e. a pointer (output parameter)
  - the max number of data elements to be received (input parameter)
  - the type of data e.g. **MPI_INT, MPI_FLOAT**, etc. (input parameter)
  - the rank of the source/sending node (input parameter)
  - a tag for identification of the communication (input parameter)
  - the communicator to be used for transmission (input parameter)
  - a pointer to a structure of type **MPI_Status**, contains source processor’s rank, communication tag, and error status (output parameter)

- For the non-blocking **MPI_Irecv**, **MPI_Request** replaces **MPI_Status**.
A first MPI example: Hello World.

```c
#include <mpi.h>
int main(int argc, char *argv[]) {
    int myid, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("process %d out of %d says Hello\n", myid, size);
    MPI_Finalize();
    return 0;
}
```
First “real” MPI program: Exchanging 2 Values

```c
#include <mpi.h>
int main(int argc, char *argv[]) {
    int rank, value, size, length = 1, tag = 1;
    MPI_Status status;
    /* initialize MPI and get own id (rank) */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (size!=2) {
        printf("use exactly two processes\n");
        exit(1);
    }
    if (myid == 0) {
        otherid = 1; myvalue = 14;
    } else {
        otherid = 0; myvalue = 25;
    }
    printf("process %d sending %d to process %d\n", myid, myvalue, otherid);
    /* Send one integer to the other node (i.e. “otherid”) */
    MPI_Send(&myvalue, 1, MPI_INT, otherid, tag, MPI_COMM_WORLD);
    /* Receive one integer from any other node */
    MPI_Recv(&othervalue, 1, MPI_INT, MPI_ANY_SOURCE,
             MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    printf("process %d received a %d\n", myid, othervalue);
    MPI_Finalize(); /* Terminate MPI */
    return 0;
}
```
Compiling and Running MPI Programmes

- To compile programs using MPI, you need an “MPI-enabled” compiler.
- On our cluster, we use `mpicc` to compile C programs containing MPI commands or `mpicxx` for C++.
- Before running an executable using MPI, you need to make sure the "multiprocessing daemon" (MPD) is running.
- It makes the workstations into "virtual machines" to run MPI programs.
- When you run an MPI program, requests are sent to MPD daemons to start up copies of the program.
- Each copy can then use MPI to communicate with other copies of the same program running in the virtual machine. Just type “mpd &” in the terminal.
- To run the executable, type “`mpirun -np N./executable_file`”, where N is the number to be used to run the program.
- This value is then used in your program by `MPI_Init` to allocate the nodes and create the default communicator.
#include <mpi.h>

int main(int argc, char *argv[]) {

    int rank, value, size;
    MPI_Status status;
    /* initialize MPI and get own id (rank) */
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    do {
        if (rank == 0) {
            scanf("%d", &value);
            /* Master Node sends out the value */
            MPI_Send(&value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
        } else {
            /* Slave Nodes block on receive the send on the value */
            MPI_Recv(&value, 1, MPI_INT, rank - 1, 0, MPI_COMM_WORLD, &status);
            if (rank < size - 1) {
                MPI_Send(&value, 1, MPI_INT, rank + 1, 0, MPI_COMM_WORLD);
            }
            printf("process %d got %d\n", rank, value);
        }
    } while (value >= 0);
    /* Terminate MPI */
    MPI_Finalize();
    return 0;
}
```c
#include <mpi.h>

int main(int argc, char *argv[]) {
    int A[4][4], b[4], c[4], line[4], temp[4], local_value, myid;
    MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if (myid == 0) {
        for (int i=0; i<4; i++) {
            b[i] = 4 - i;
            for (int j=0; j<4; j++)
                A[i][j] = i + j; /* set some notional values for A, b */
        }
        line[0]=A[0][0]; line[1]=A[0][1];
    }
    if (myid == 0) {
        for (int i=1; i<4; i++) { /* slaves do most of the multiplication */
            MPI_Send( temp, 4, MPI_INT, i, i, MPI_COMM_WORLD);
            MPI_Send( b, 4, MPI_INT, i, i, MPI_COMM_WORLD);
        }
    }
    else {
        MPI_Recv( line, 4, MPI_INT, 0, myid, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        MPI_Recv( b, 4, MPI_INT, 0, myid, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        /* master node does its share of multiplication too*/
    }
    if (myid != 0) {
        MPI_Send(&c[myid], 1, MPI_INT, 0, myid, MPI_COMM_WORLD);
    }
    else {
        for (int i=1; i<4; i++) {
            MPI_Recv( &c[i], 1, MPI_INT, i, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        }
    }
    MPI_Finalize();
    return 0;
}
```

Example 4: Matrix-Vector Product Implementation
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    #define INT_MAX_ 1000000000
    int myid, size, inside=0, outside=0, points=10000;
    double x,y, Pi_comp, Pi_real=3.141592653589793238462643;
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (myid == 0) {
        for (int i=1; i<size; i++) /* send out the value of points to all slaves */
            MPI_Send(&points, 1, MPI_INT, i, i, MPI_COMM_WORLD);
    } else
        MPI_Recv(&points, 1, MPI_INT, 0, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    rands=new double[2*points];
    for (int i=0; i<2*points; i++){
        rands[i]=random();
        if (rands[i]<=INT_MAX_) i++ /* this random is within range */
    }
    for (int i=0; i<points;i++ ){
        x=rands[2*i]/INT_MAX ;
        y=rands[2*i+1]/INT_MAX ;
        if((x*x+y*y)<1) inside++ /* point is inside unit circle so incr var inside */
    }
    delete[] rands;
    if (myid == 0) {
        for (int i=1; i<size; i++) {
            int temp; /* master receives all inside values from slaves */
            MPI_Recv(&temp, 1, MPI_INT, i, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            inside+=temp; } /* master sums all insides sent to it by slaves */
    } else
        MPI_Send(&inside, 1, MPI_INT, 0, i, MPI_COMM_WORLD); /* send inside to master */
    if (myid == 0) {
        Pi_comp = 4 * (double) inside / (double)(size*points);
        cout << "Value obtained: " << Pi_comp << endl << "Pi: " << Pi_real << endl;
    }
    MPI_Finalize(); return 0;
}
Collective communications in MPI

• Groups are sets of processors that communicate with each other in a certain way.
• Such communications permit a more flexible mapping of the language to the problem (allocation of nodes to subparts of the problem etc).
• MPI implements Groups using data objects called Communicators.
• A special Communicator is defined (called ‘MPI_COMM_WORLD’) for the group of all processes.
• Each Group member is identified by a number (its Rank 0..n-1).
• There are three steps to create new communication structures:
  – accessing the group corresponding to MPI_COMM_WORLD,
  – using this group to create sub-groups,
  – allocating new communicators for this group.
• We will see this in more detail in the last examples.
Some Sophisticated MPI Routines

• The advantage of the global communication routines below is that the MPI system can implement them more efficiently than the programmer, involving far less function calls.

• Also the system will have more opportunity to overlap message transfers with internal processing and to exploit parallelism that might be available in the communications network.

   - **MPI_Barrier**   Synchronise
   - **MPI_Bcast**    Broadcast same data to all procs
   - **MPI_Gather**   Get data from all procs
   - **MPI_Scatter**  Send different data to all procs
   - **MPI_Reduce**   Combine data from all onto one proc
   - **MPI_Allreduce** Combine data from all procs onto all procs
Sophisticated MPI Routines:  
**MPI_Barrier**

- **MPI_Barrier** is used to synchronise a set of nodes.
  ```c
  int MPI_Barrier( MPI_Comm comm )
  ```
- It blocks the caller until all group members have called it.
- ie call returns at any process only after all group members have entered the call.
- This functions takes only parameter, the communicator (i.e. group of nodes) to be synchronised.
- As we previously saw with other functions, it will most of the times be used with the default communicator,  
  **MPI_COMM_WORLD**.
Sophisticated MPI Routines: **MPI_Bcast**

- **MPI_Bcast** used to send data from one node to all the others in one single command.

```c
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )
```

- This functions takes five parameters:
  - location of data to be sent i.e. a pointer *(input/output parameter)*
  - number of data elements to be sent *(input parameter)*
  - type of data *(input parameter)*
  - rank of the broadcast node *(input parameter)*
  - communicator to be used *(input parameter)*
Sophisticated MPI Routines: **MPI_Gather**

- **MPI_Gather** is used to gather on a single node data scattered over a group of nodes.

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

- This functions takes eight parameters:
  - location of data to be sent i.e. a pointer (*input* parameter)
  - number of data elements to be sent (*input* parameter)
  - type of data to be sent (*input* parameter)
  - location of the receive buffer i.e. a pointer (*output parameter*)
  - number of elements to be received (*input* parameter)
  - type of data to be received (*input* parameter)
  - rank of the sending node (*input parameter*)
  - communicator to be used for transmission. (*input parameter*)

![Diagram](image.png)
Sophisticated MPI Routines: **MPI_Scatter**

- **MPI_Scatter** used to scatter data from single node to a group

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

- This function takes eight parameters:
  - location of data to be sent i.e. a pointer (**input** parameter)
  - number of data elements to be sent (**input** parameter)
  - type of data to be sent (**input** parameter)
  - location of the receive buffer i.e. a pointer (**output parameter**)
  - number of elements to be received (**input** parameter)
  - type of data to be received (**input** parameter)
  - rank of the sending node (**input** parameter)
  - communicator to be used for transmission. (**input** parameter)
Sophisticated MPI Routines: **MPI_Reduce**

- **MPI_Reduce** used to reduce values on all nodes of a group to a single value on one node using some reduction operation (sum etc).

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

- This function takes six parameters:
  - location of the data to be sent i.e. a pointer (*input* parameter)
  - location of the receive buffer i.e. a pointer (*output* parameter)
  - number of elements to be sent (*input* parameter)
  - type of data e.g. **MPI_INT, MPI_FLOAT**, etc. (*input* parameter)
  - operation to combine the results e.g. **MPI_SUM** (*input* parameter)
  - communicator used for transmission (*input* parameter)
Sophisticated MPI Routines: **MPI_Allreduce**

- **MPI_Allreduce** is used to reduce values on all group nodes to a one value, and send it back to all (i.e. equals **MPI_Reduce+MPI_Bcast**)

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

- This functions takes six parameters:
  - location of the data to be sent i.e. a pointer (*input* parameter)
  - location of the receive buffer i.e. a pointer (*output* parameter)
  - number of elements to be sent (*input* parameter)
  - type of data e.g. **MPI_INT**, **MPI_FLOAT**, etc. (*input* parameter)
  - operation to combine the results e.g. **MPI_SUM** (*input* parameter)
  - communicator used for transmission (*input* parameter)
```c
#include <mpi.h>

int main(int argc, char *argv[]) {
    int A[4][4], b[4], c[4], line[4], temp[4], local_value, myid;
    MPI_Init(&argc, &argv); MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    if (myid == 0) {
        for (int i=0; i<4; i++) {
            b[i] = 4 - i;
            for (int j=0; j<4; j++)
                A[i][j] = i + j; /* set some notional values for A, b */
        }
        line[0]=A[0][0]; line[1]=A[0][1];
    }
    MPI_Bcast(b, 4, MPI_INT, 0, MPI_COMM_WORLD);
    if (myid == 0) {
        for (int i=0; i<4; i++) { /* slaves do most of the multiplication */
            MPI_Send( temp, 4, MPI_INT, i, i, MPI_COMM_WORLD);
            /* No need to send vector b here */
        }
    } else {
        MPI_Recv( line, 4, MPI_INT, 0, myid, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        /* No need to receive vector b here */
        /* master node does its share of multiplication too*/
    } else {
    }
    if (myid != 0) {
        MPI_Send(&c[myid], 1, MPI_INT, 0, myid, MPI_COMM_WORLD);
    } else {
        
        
    }
    MPI_Finalize(); return 0;
}
```

Example 6: A New Matrix-Vector Product
Example 5: A New Pi Calculation Implementation

```c
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    #define INT_MAX_ 1000000000
    int myid, size, inside=0, outside=0, points=10000;
    double x,y, Pi_comp, Pi_real=3.141592653589793238462643;
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    /* Again send/receive replaced by MPI_Bcast */
    MPI_Bcast(&points,1,MPI_INT, 0, MPI_COMM_WORLD);
    rands=new double[2*points];
    for (int i=0; i<2*points; i++){
        rands[i]=random();
        if (rands[i]<=INT_MAX_) i++ /* this random is within range */
    }
    for (int i=0; i<points;i++){
        x=rands[2*i]/INT_MAX ;
        y=rands[2*i+1]/INT_MAX ;
        if((x*x+y*y)<1) inside++ /* point is inside unit circle so incr var inside */
    }
    delete[] rands;
    if (myid == 0) {
        for (int i=1; i<size; i++) {
            int temp; /* master gets all inside values from slaves */
            MPI_Recv(&temp, 1, MPI_INT, i, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            inside+=temp; } /* master sums all insides sent to it by slaves */
    } else
    MPI_Send(&inside, 1, MPI_INT, 0, i, MPI_COMM_WORLD); /* send inside to master */
    MPI_Reduce(&inside,&total,1,MPI_INT,MPI_SUM,0, MPI_COMM_WORLD);
    if (myid == 0) {
        Pi_comp = 4 * (double) inside / (double)(size*points);
        cout << "Value obtained: " << Pi_comp << endl << "Pi:" << Pi_real << endl;
        MPI_Finalize(); return 0;
    }
}
```

Using Communicators

• Creating a new group (and communicator) by excluding the first node:

```c
#include <mpi.h>
int main(int argc, char *argv[]) {
    // ...
    MPI_Comm comm_world, comm_worker;
    MPI_Group group_world, group_worker;
    comm_world = MPI_COMM_WORLD;
    MPI_Comm_group(comm_world, &group_world);
    MPI_Group_excl(group_world, 1, 0, &group_worker);
    /* process 0 not member */
    MPI_Comm_create(comm_world, group_worker, &comm_worker);
    // ...
}
```

• Warning:

`MPI_Comm_create()` is a collective operation, so all processes in the old communicator must call it - even those not going in the new communicator.
Example 8: Using Communicators

```c
#include <mpi.h>
#include <stdio.h>
#define NPROCS 8
int main(int argc, char *argv[]) {

    int rank, newrank, sendbuf, recvbuf;
    ranks1[4]={0,1,2,3}, ranks2[4]={4,5,6,7};
    MPI_Group orig_group, new_group;
    MPI_Comm new_comm

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    sendbuf = rank;
        /* Extract the original group handle */
    MPI_Comm_group(MPI_COMM_WORLD, &orig_group);
    if (rank < NPROCS/2) /* Split tasks into 2 distinct groups based on rank */
        MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
    else
        MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);

        /* Create new communicator and then perform collective communications */
    MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
    MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);
    MPI_Group_rank (new_group, &new_rank);

    printf("rank= %d newrank= %d recvbuf= %d\n", rank, newrank, recvbuf);

    MPI_Finalize();
}
```
Final Reminder

- MPI programs need specific compilers (e.g. `mpicc`), MPD and `mpirun`.
- MPI programs start with `MPI_Init` and finish with `MPI_Finalize`,
- Four functions for point-to-point communication,
- Six more advanced functions, for synchronise, and perform collective communication,
- Nine functions (at least three!) to create new groups and communicators,
- Too many examples to remember everything.