Moduel 1 : Basic MPI programs using point-ot-point library calls and execute on Message Passing Cluster or Multi Core Systems that support MPI library.

Example 1.1	Write MPI Program (SPMD) to get started.
Example 1.3	Write MPI program to find sum of *n* integers on Parallel Processing Platform. You have to use MPI point-to-point blocking communication library calls .
Example 1.4	Write MPI program to find sum of *n* integers on Parallel Processing Platform. You have to use MPI point-to-point blocking communication library calls .
Example 1.5	Write MPI program to find sum of *n* integers on a Parallel Computing System in which procesors are connected with ring topology and use MPI point-to-point blocking communication library calls .
Example 1.6	Write MPI program to find sum of *n* integers on a Parallel Computing System in which procesors are connected with tree topology (Associative-fan-in rule for tree can be assumed) and use MPI point-to-point blocking communication library calls.
Example 1.7	Write MPI program to compute the vlaue of PI by numerical integration using MPI point-to-point blocking communication library calls.
Example 1.8	Write MPI program for prefix sum (scan operation) calculation using MPI point-to-point blocking communciation library calls.

Description of Programs - MPI Point-to-Point Communication Library Calls

Example 1.1: Write MPI program to get started (SPMD) - Hello World to all process

- **Objective**

Write a MPI program to print "*Hello World*"

- **Description**

Process with rank *k* (*k* = 1, 2, ..., *p*-1) will send"*Hello World*" message to process with rank*0*. Process with rank*0*receives the message and prints it. You have to use MPI *point-to-point* blocking communication library calls (MPI_Send and MPI_Recv) to write this program.

- **Input**

None

- **Output**

Process with rank *0* prints the following message with each of the other process's Rank Numbers (i.e. *rank* = 1,2, ..., p-1). Hello World from Process's *rank* where *rank = 1,2, ..., p-1*

Example 1.3: Description for implementation of MPI program to find sum of *n* integers using MPI point-to-point blocking communication library calls

- **Objective**

Write a MPI program to find sum of *n* integers on *p* processors of Message Passing cluster

- **Description**

Each process with rank greater than *0* sends the value of its rank to the process with rank *0*. Process with rank *0* receives the values from each process and calculate the sum. Finally, process with rank *0* prints the sum.

- **Input**

For input data, let each process uses its identifying number, i.e. the value of its rank. For example, process with rank *0* uses the number 0, process with rank *1* uses the number 1, etc.

- **Output**

Process with rank *0* prints the final sum.

Example 1.4: Write MPI program to find sum of *n* integers on a Parallel Processing platform in which processors are connected by linear array topology. You have to use MPI point-to-point blocking communication library calls

- **Description**

In *linear array* interconnection network, each processor (except the processor at the end) has a direct communication link to the immediate next processor. A simple way of communicating a message between processors is, by repeatedly passing message to the processor immediately to either right or left , until it reaches its destination, i.e. last processor in the *linear array*. A simple way to connect processors is illustrated in Figure 1.

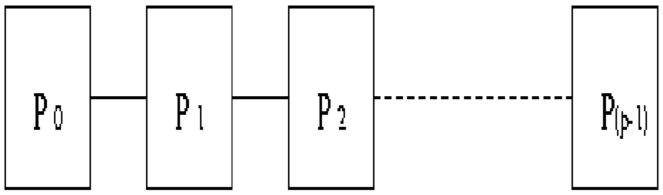

Figure 1 Parallel Computing system - Processors are connected by Linear Array topology

All the processes are involved in communication. The processes with rank *k* (*k* is greater than *0*) receives the accumulated or partial sum from the previous process with rank *k*-

1. Finally, process with rank *p*-1 prints the final sum.

- **Input**

  For input data,let each process use its identifying number, i.e. the value of its rank. For example, process with rank *0* uses the number 0, process with rank *1* uses the number 1, etc.

- **Output**

  Process with rank*p*-1 prints the final sum.

Example 1.5: **Description for implementation of MPI program to find sum of *n* integers on parallel computer in which processors are arranged in *ring topology* using point-to-point blocking communication library calls**

- **Objective**

Write a MPI program to find sum of *n* values using *p* processors of cluster. Assume that *p* processors are arranged in *ring topology*.

- **Description**

In linear array interconnection network (*Refer Example 1.4 for more details on linear array topology*) with a wraparound connection is called as a *ring*. A wraparound connection is often provided between the processors at the end.A simple way of communicating a message between processors is, by repeatedly passing message to the processor immediately to either right or left; depending on which direction yield a shorter path, until it reaches its destination, i.e., first processor in the *ring*.

All the processes are involved in communication. The process with rank *k* (*k* is greater than *0*) receives the accumulated or partial sum from the previous process with rank *k*-1. Process with rank *p*-1 sends the final sum to process with rank *0*. Finally, process with rank *0* prints the final sum.

- **Input**

For input data, let each process use its identifying number, i.e. the value of its rank. For example, process with rank *0* uses the number 0, process with rank *1* uses the number 1, etc.

- **Output**

Process with rank *0* prints the final sum.

Example 1.6: **Description for implementation of MPI program to find sum of *n* integers on parallel computer in which processors are arranged in *binary tree topology (associative fan-in rule)* using MPI point-to-point blocking communication library calls**

- **Objective**

Write a MPI program to find sum of *n* ($n = 2^i$; *i* = 3) integers on *p* (*p*=*n*) processors of cluster using *associative fan-in* rule.

- **Description**

The *first step* is to group the total number of processors in ordered pairs such as (P_0, P_1), (P_2, p_3), (P_4, P_5), (P_6, p_7), ..., (P_{p-2}, P_{p-1}). Then compute the partial sums in all pairs of processors using MPI point-to-point blocking communication and accumulate the partial sum on the processors P_0, P_2, ...,P_{p-2}. For example, the processor P_i (*i* is even) computes the partial sum for the pair (P_i, P_{i+1}) by performing MPI point-to-point blocking communication library calls.

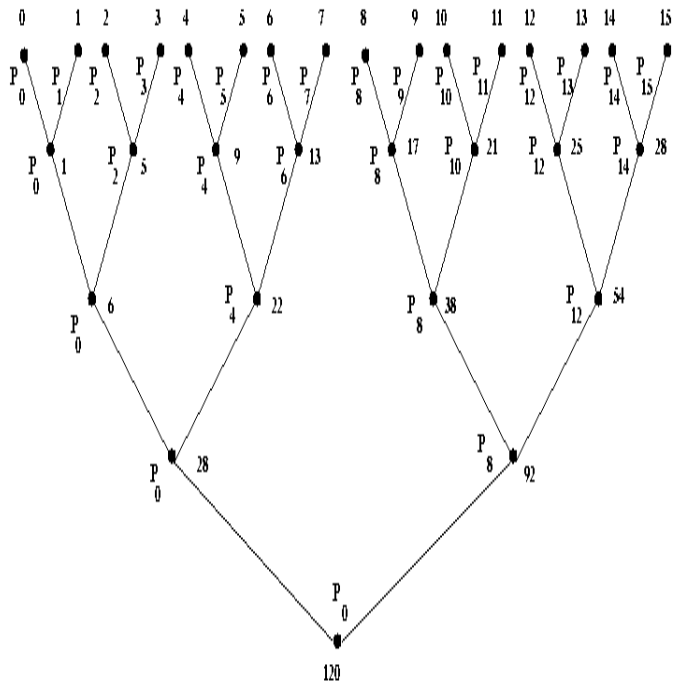

Figure 2. Tree Structure for sum of *n* integers by Associativ fan-in rule

In the *second step*, consider the pair of processors (P_0, P_2), (P_4, P_6), ..., (P_{p-4}, P_{p-2}) obtain the new partial sum by considering the existing accumulated partial sums on the processors P_0, P_2, P_4, ..., P_8as explained in the previous step. This procedure is repeated till two processors are left out and finally accumulate the global sum on the processor P_0.In this example, MPI point-to-point blocking communication library calls such as,**MPI_Send** and **MPI_Recv**, are used. An example of *associative fan-in* rule is described as a tree structure in the Figure 2 for *n*= 16

- **Input**

For input data, let each process use its identifying number, i.e. the value of its rank. For example, process with rank *0* uses the number 0, process with rank *1* uses the number 1, etc.

- **Output**

process with rank *0* prints the final sum.

Example 1.7: Description for implementation of MPI program to compute the value of *pie* by Numerical Integration using MPI point-to-point communication library calls

- **Objective**

Write a MPI program to compute the value of *pie* function by numerical integration of a function $f(x) = 4/(1+x^2)$ between the limits 0 and 1.

- **Description**

There are several approaches to parallelizing a serial program. One approach is to partition the data among the processes. That is we partition the interval of integration [0,1] among the processes, and each process estimates local integral over its own subinterval. The local calculations produced by the individual processes are combined to produce the final result. Each process sends its integral to process 0, which adds them and prints the result.

To perform this integration numerically, divide the interval from 0 to 1 into *n* subintervals and add up the areas of the rectangles as shown in the Figure 3 (*n* = 5). Large values of *n* give more accurate approximations of *pi* . Use MPI point-to-point communication library calls.

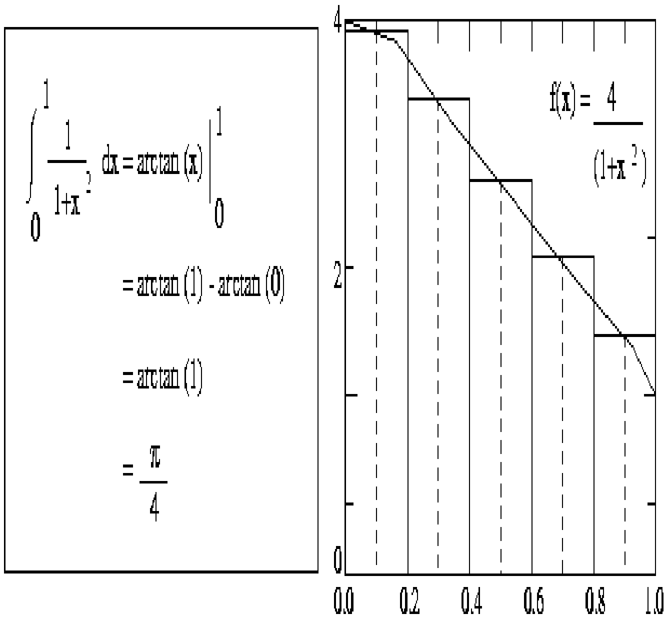

Figure 3 Numerical integration of *pie* function

We assume that *n* is total number of subintervals, *p* is the number of processes and $p < n$. One simple way to distribute the total number of subintervals to each process is to dividen by *p*. There are two kinds of mappings that balance the load. One is a *block mapping*, partitions the array elements into blocks of consecutive entries and assigns the block to the processes. The other mapping is a *cyclic mapping*. It assigns the first element to the first process, the second element to the second, and so on. If $n > p$, we get back to the first process, and repeat the assignment process for remaining elements. This process is repeated until all the elements are assigned. We have used a *cyclic mapping* for partition of interval [0,1] onto*p* processes.

- **Input**

Process with rank*0* reads the input parameter *n*, the number of intervals on command line.

- **Output**

Process with rank *0* prints the computed value of *pi* function.

Example 1.8: **Write MPI program for prefix sum calculation (scan operation) on a Parallel computing system with Hypercube as interconnection network topology using MPI point-to-point blocking communciation library calls.**

- **Description**

you will write your own parallel program, for computing the **prefix sums** on Parallel Computer with HyperCube as interconnect. Even though you will be programming and running your program using MPI on **COW** with interconnection net work Fast EtherNet or Gigabit or on Message Passing Cluster, assume that program will run on a HyperCube network.

Finding **prefix sums** is another important problem that can be solved by using a communication pattern similar to that used in reduction. Given *p* numbers n_0, n_1 , n_2, n_1 , n_3, ... , n_{p-2} , n_{p-1} , (one on each processor), then the problem is compute the **SUMS** s_k (= **SUMMATION** { n_i }, for $i = 0,1,...$, *k* and for all *k* between 0 and *p-1*).
For example, if the original sequence of numbers is (0,1,2,3,4), then the sequence of **prefix sums** is (0, 3, 6,10). Initially, n_k resides on the processor labeled *k*, and at the end of the procedure, the same processor holds s_k . The below Figure 3 illustrates the prefix sums procedure for an eight-processor HyperCube and dimension of hypercube (*d*) is 3.

```
procedure PREFIX_SUMS_HCUBE(my_id, my_number, d, result)
begin
    result:= my_number;
    msg:= result;
    for i:= 0 to d do
    begin
        partner:= my_id XOR $2^i$ ;
        send msg to partner ;
        receive number from partner ;
        msg:= msg+number ;
        if (partner<my_id) then result:= result+number ;
    endfor;
end PREFIX_SUMS_HCUBE
```

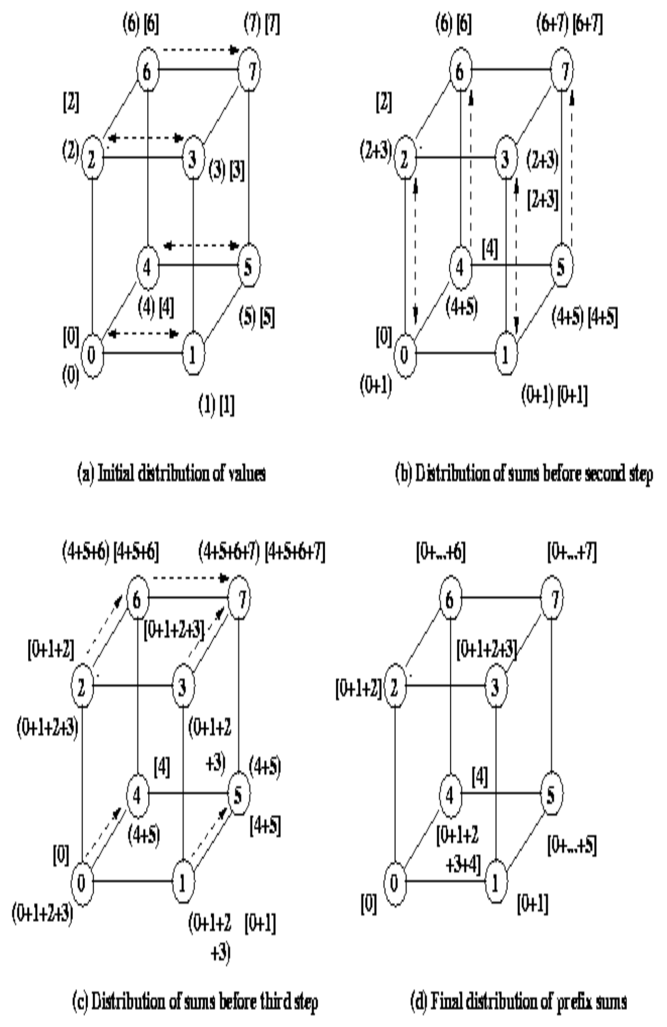
(a) Initial distribution of values
(b) Distribution of sums before second step
(c) Distribution of sums before third step
(d) Final distribution of prefix sums

Figure 4 : Computing prefix sums on an eight-processor HyperCube.

In the figure 4, at each processor, square brackets show the local prefix sum accumulated in a buffer and parentheses enclose the contents of the ; outgoing message buffer for the next step. In prefix sums the processor with label *k*, uses information from only the *k*-processor subset of those processors whose labels are less than or equal to *k*. To accumulate the correct prefix sum, every processor maintains an additional result buffer. This buffer is denoted by square brackets in the above figure. At the end of communication step, the content of an incoming messageis added to the result buffer only if the message comes from a processor with a smaller label than that of the recipient processor. The contents of the outgoing message(denoted by parentheses in the figure) are updated with every incoming message. For instance after the first communication step, processor 0, 2 and 4 do not add the data received from processors 1, 3 and 5 to their result buffers. However, the contents of the outgoing messages for the next step are updated.

Since not all the messages received by a processor contribute to its final result, some of the messages it receives may be redundant. We have omitted these steps of the standard all-to-all broadcast communication pattern, although the presence or absence of these messages does not affect the results of the algorithm. Above algorithm gives the procedure to solve the prefix sums problem on a *d*-dimensional HyperCube.

Let the dimension of the HyperCube be a parameter in your code. It is not necessary to pass this parameter to the program on the command line. It is sufficient to define a constant in one spot in your code, representing the dimension of the target machine. It is acceptable to use more than one process on a physical machine.

In order to run your code you will have to somehow number your processes from *0* to *p-1* where *p* is the total number of nodes in the presumed HyperCube. Assume that these numbers form a standard numbering system for a HyperCube, as shown in the Figure 3.

- **Input**

Let the dimension of your target machine be four, giving total of 16 nodes (processors) in your HyperCube. The input *n* integers are partitioned among *p (p=n)* processors such that every processor has exactly one integer. In fact, for data, let each node uses its identifying number, so that node 0 uses the number 0, node 1 uses the number 1, etc.

- **Output**

Print the result of **prefix sums** , followed by task *id* (*process id number*) in ascending order of the node number followed by the name of the host machine that process was executing on. Each set of the data should be printed on a single line by itself so that 16 and only 16 lines of data are printed out. You can format your data for readability and make sure that your data is printed out in order of the node number, starting with node number 0 and ending with node number *p* -1.