



Programming Using the Message-Passing Paradigm

Distributed memory Architecture

- Distributed-memory
 - Collection of core-memory pairs connected by a network.
 - The memory associated with a core is directly
 - accessible only to that core
- Shared-memory systems
 - collection of cores connected to a globally
 - accessible memory, in which each core can have access to any memory location.

Distributed memory Architecture

- Each **processor** has its own private **memory**.
- Computational tasks can only operate on local data, if remote data is required, the computational task must communicate with one or more remote processors. Communication through the **message passing**
- **Communication between Process:**
 - a program running on one core-memory pair is usually called a process
 - Two processes can communicate by calling functions:
 - one process calls a send function and the other calls a receive function
 - **MPI- Message-Passing Interface**
 - It defines a library of functions

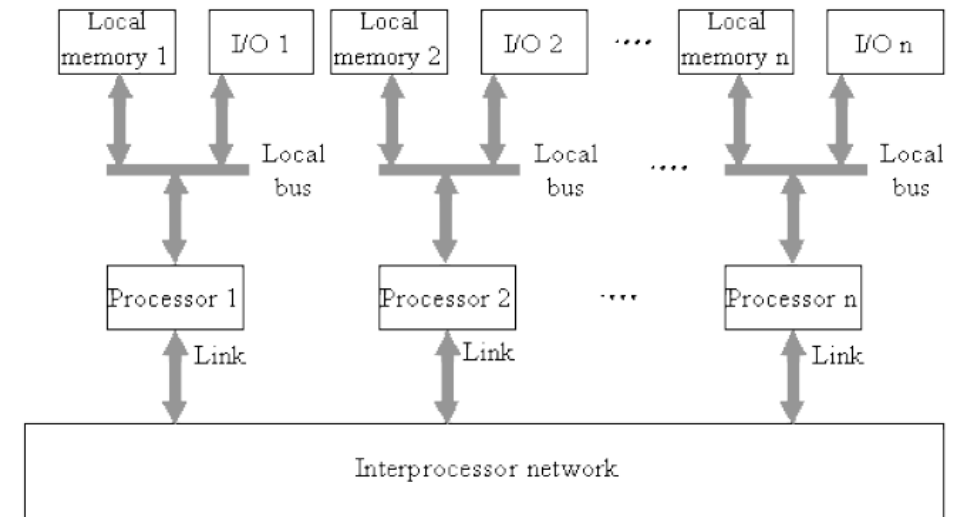


Fig: A multiprocessor system with a distributed memory (loosely coupled system)¹

Programming using the Message-Passing Paradigm

■ Principles of Message-Passing Programming

- Message-passing paradigm consists of p processes, each with its own exclusive address space.
- All **interactions** (read-only or read/write) require cooperation of two processes
- The programmer is fully aware of all the costs of non-local interactions by Two way interactions.

Structure of Message-Passing Programs

Message-passing programs are often written using the asynchronous or loosely synchronous paradigms.

■ In the asynchronous paradigm :

- all concurrent tasks execute asynchronously

loosely synchronous :

- Tasks or subsets of tasks synchronize to perform interactions.
- Between these **interactions**, tasks execute **completely asynchronously**

What is MPI?

- A message-passing library specification
- extended message-passing model
- For parallel computers, clusters, and heterogeneous networks
- MPI provides a powerful, efficient, and portable way to express parallel programs



Two important questions that arise early in a parallel program are:

- How many processes are participating in this computation?
- Which one am I?

- **MPI_Comm_size** reports the number of processes.
- **MPI_Comm_rank** reports the *rank*, a number between 0 and $\text{size}-1$, identifying the calling process

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Example

The provided code snippet is a MPI (Message Passing Interface) program written in C that demonstrates the

- **Basic structure of a parallel program.**
- The program initializes MPI, retrieves the rank of each process,
- prints a "Hello from process" message, and then finalizes MPI.
- Each process prints its rank along with the message.

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

```
int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello from process %d\n", rank);
    MPI_Finalize();
    return 0;
}
```

Build, Compile, Run, and analyze performance.

```
configure
```

```
make
```

```
mpicc -mpitrace myprog.c
```

```
mpirun -np 10 myprog
```

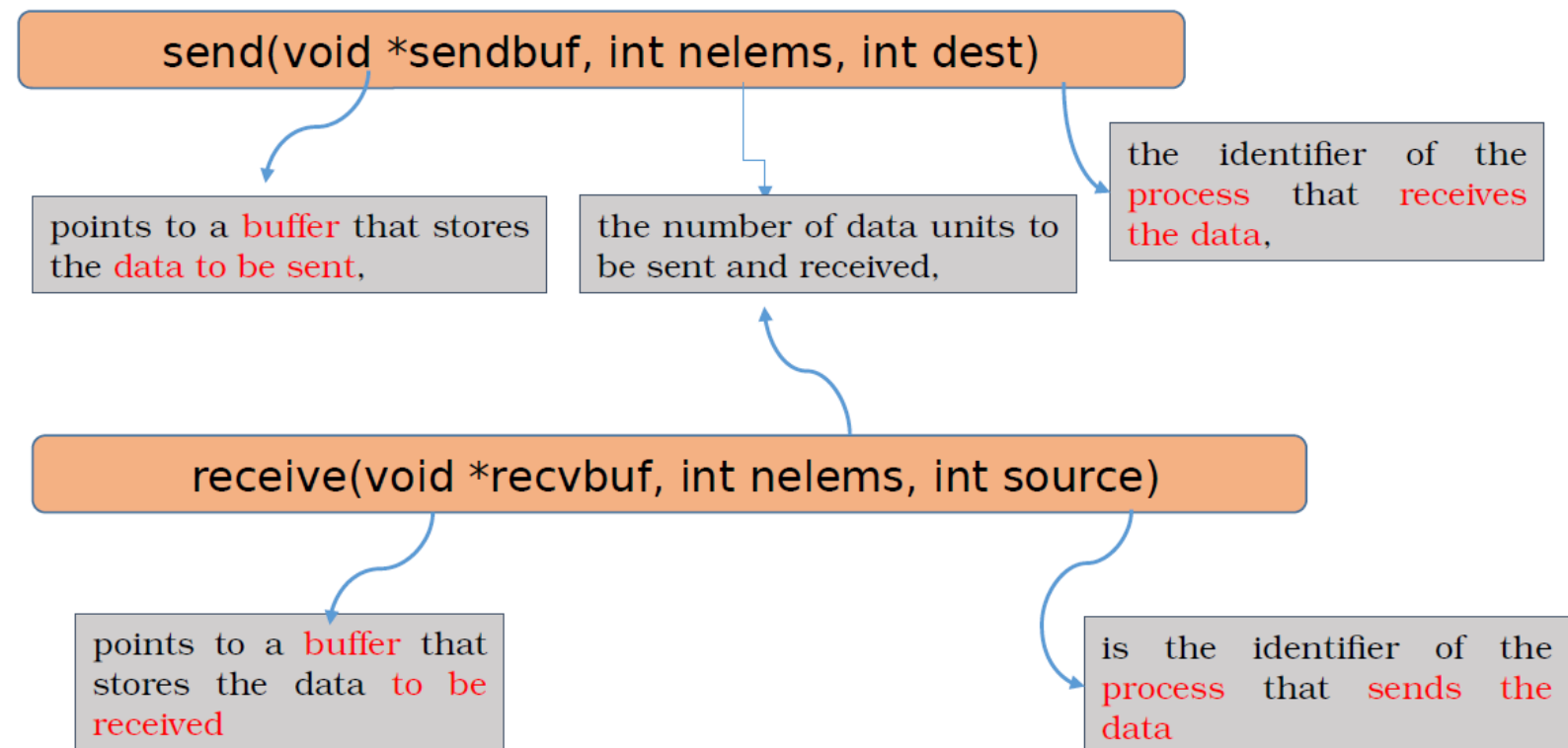
```
upshot myprog.log
```

-
- There is a default communicator whose group contains all initial processes, called **MPI_COMM_WORLD**.
 - Processes can be collected into *groups*.
 - Each message is sent in a *context*, and must be received in the same context.
 - A group and context together form a *communicator*.

The Building Blocks: Send and Receive operations

❑ Basic operations in the message-passing programming paradigm are send and receive

❑ In their simplest form, the prototypes of these operations are defined as follows:

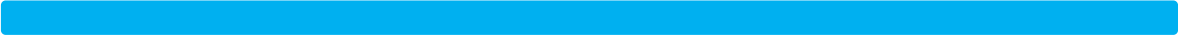


MPI Datatypes

The data in a message to sent or received : triple (address, count, **datatype**)

MPI *datatype* is Defined as:

- Predefined data type (e.g., MPI_INT, MPI_DOUBLE_PRECISION)
- a contiguous array of MPI datatypes
- a strided block of datatypes
- an indexed array of blocks of datatypes
- an arbitrary structure of datatypes



Many parallel programs can be written using just these six functions, only two of which are non-trivial:

- `MPI_INIT`
- `MPI_FINALIZE`
- `MPI_COMM_SIZE`
- `MPI_COMM_RANK`
- `MPI_SEND`
- `MPI_RECV`

1 P0
2
3 a = 100;
4 send(&a, 1, 1);
5 a=0;

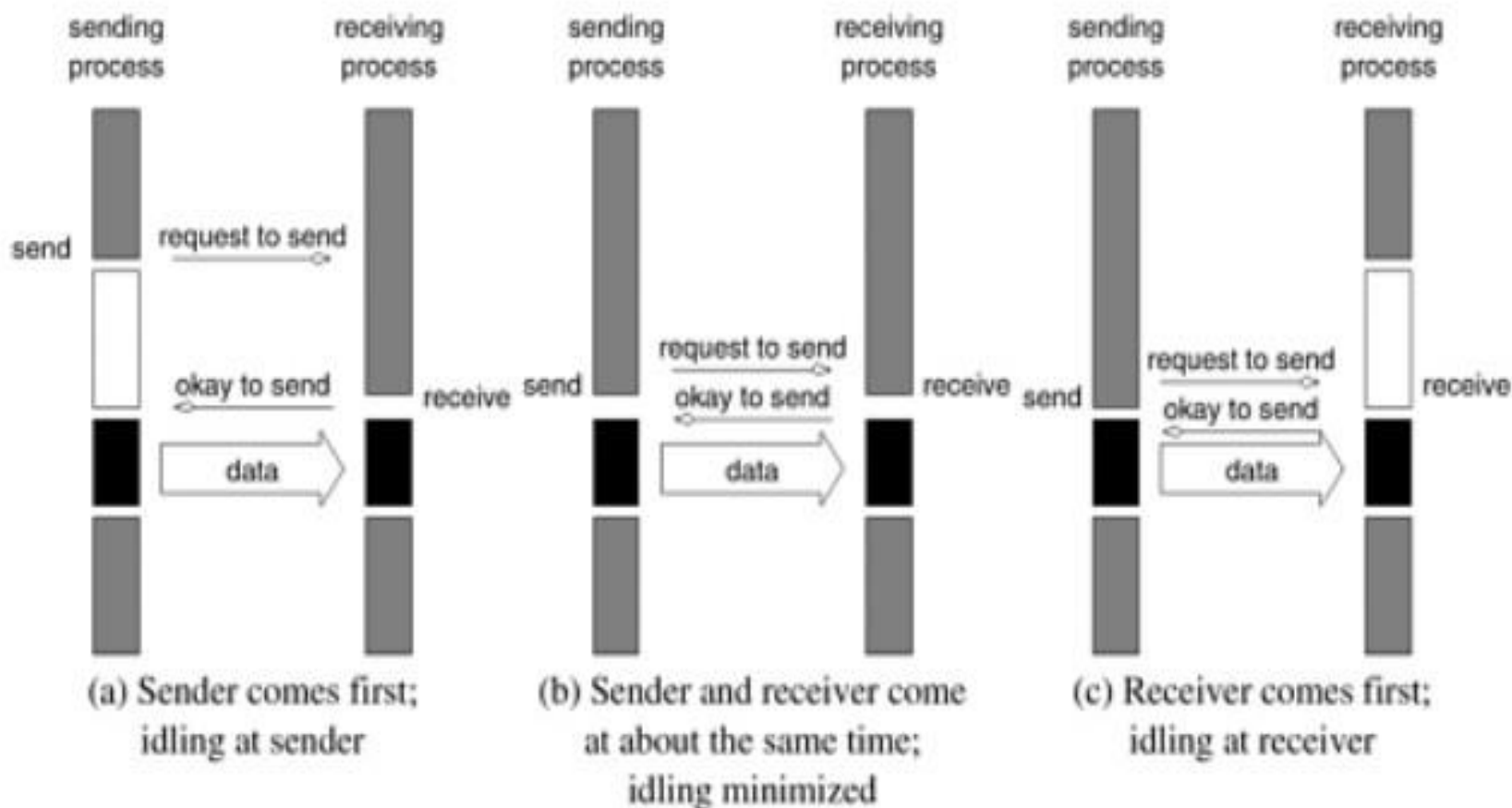
P1

receive(&a, 1, 0)
printf("%d\n", a);

Send and Receive Operations

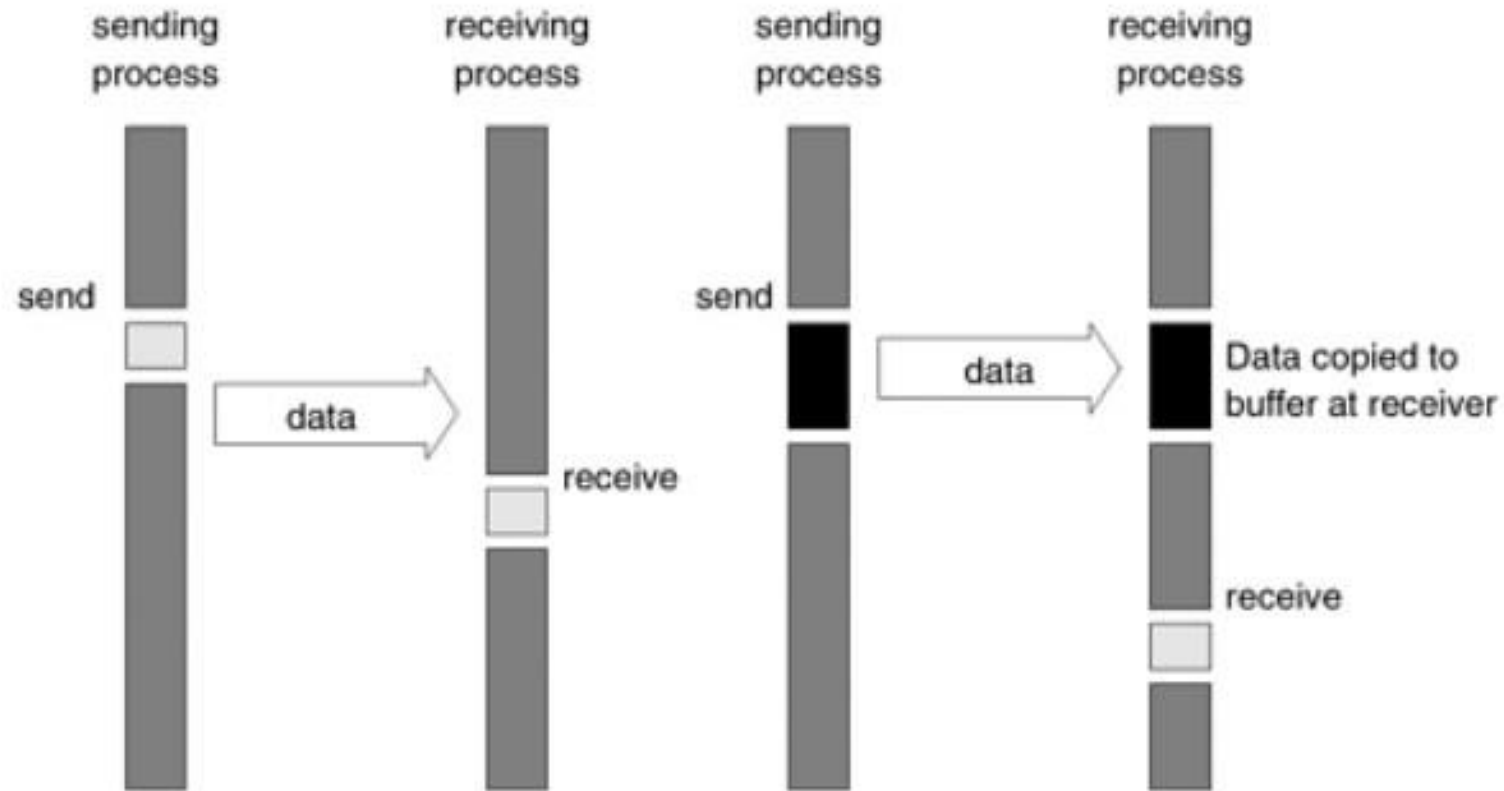
- Blocking Message Passing Operations
 - •Blocking Non-Buffered Send/Receive
 - •Blocking Buffered Send/Receive.
- Non-Blocking Message Passing Operations

Figure 6.1. Handshake for a blocking non-buffered send/ receive operation. It is easy to see that in cases where sender and receiver do not reach communication point at similar times, there can be considerable idling overheads.



Blocking Buffered Send/Receive

Figure 6.2. Blocking buffered transfer protocols: (a) in the presence of communication hardware with buffers at send and receive ends; and (b) in the absence of communication hardware, sender interrupts receiver and deposits data in buffer at receiver end.






blocking protocols

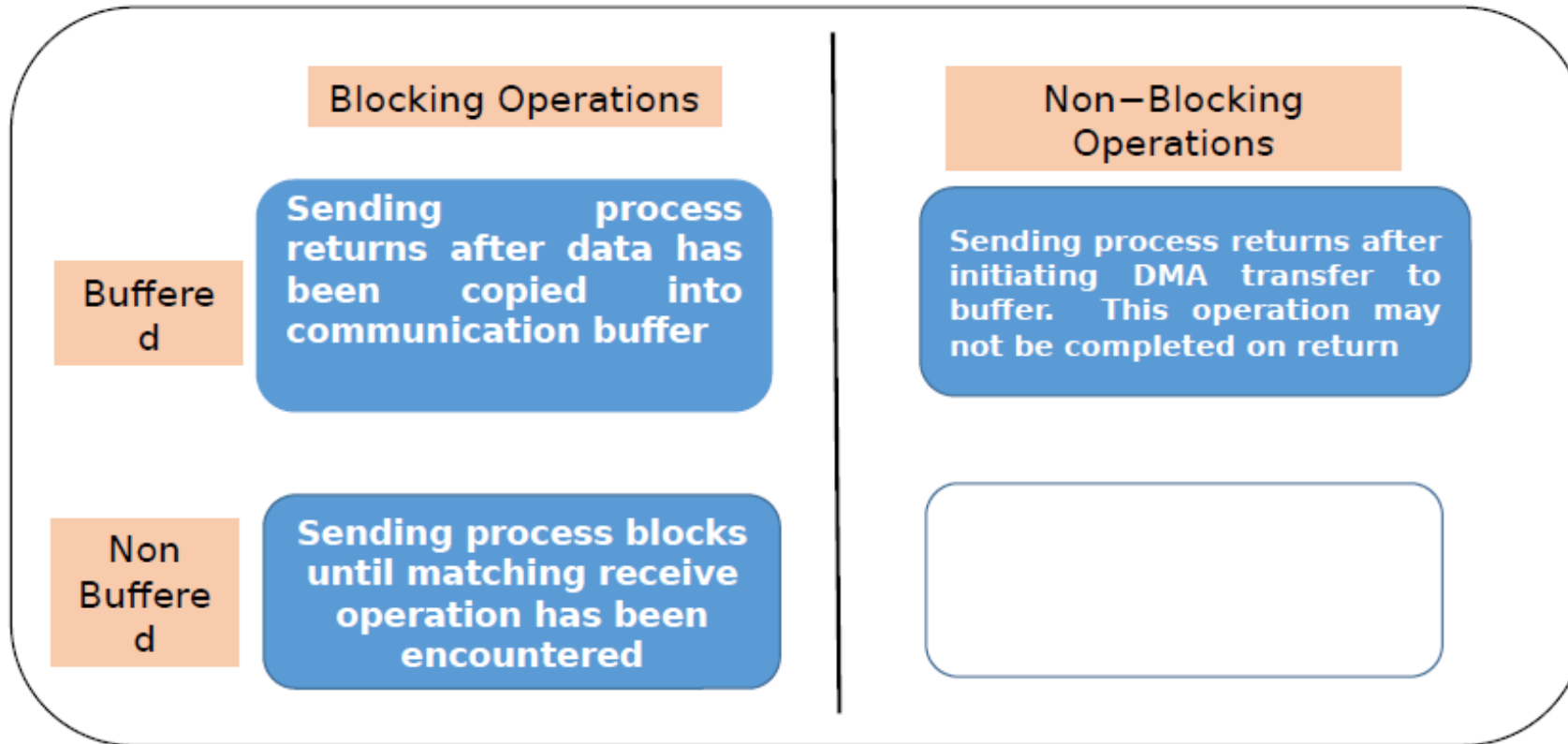
- overhead of guaranteeing semantic correctness was paid in the form of idling (non-buffered) or buffer management (buffered).

● non-blocking protocols

- – Returns from the send or receive operation before it is semantically safe to do so.
 - – User must be careful not to alter data that may be potentially participating in a communication operation.
 - – process can check, If non-blocking **operation has not completed, and then wait for its completion.**
 - – Non-blocking operations are **generally accompanied by a check-status operation**
- 

Blocking Send and Receive Protocols

Possible protocols for send and receive operations.



MPI Basic (Blocking) Send

`MPI_SEND (start, count, datatype, dest, tag, comm)`

The message buffer is described by (**start**, **count**, **datatype**).

The target process is specified by **dest**, which is the rank of the target process in the communicator specified by **comm**.

When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI Isend (): Nonblocking send. asynchronous



Process 0

Send (1)
Recv (1)

Process 1


Recv (0)
Send (0)

Process 0

Isend (1)
Irecv (1)
Waitall

Process 1

Isend (0)
Irecv (0)
Waitall



MPI Basic (Blocking) Receive

- `MPI_RECV(start, count, datatype, source, tag, comm, status)`
- Waits until a matching (on **source** and **tag**) message is received from the system, and the buffer can be used.
- **source** is rank in communicator specified by **comm**, or **MPI_ANY_SOURCE**.
- **status** contains further information
- Receiving fewer than **count** occurrences of **datatype** is OK, but receiving more is an error.

```
int
main (int argc, char **argv)
{
    int num_procs;
    int rank;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    printf("%d: hello (p=%d)\n", rank, num_procs);
    round_robin(rank, num_procs);
    printf("%d: goodbye\n", rank);

    MPI_Finalize();
}
```

```
void  
round_robin(int rank, int procs)  
{  
    long int rand_mine, rand_prev;  
    int rank_next = (rank + 1) % procs;  
    int rank_prev = rank == 0 ? procs - 1 : rank - 1;  
    MPI_Status status;  
  
    srand(time(NULL) + rank);  
    rand_mine = random() / (RAND_MAX / 100);  
    printf("%d: random is %ld\n", rank, rand_mine);
```




Introduction to Collective Operations in MPI

■ **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

Data Distribution

Example :

- Data distributions
- Suppose we want to write a function that computes a vector sum serial.

$$\begin{aligned}\mathbf{x} + \mathbf{y} &= (x_0, x_1, \dots, x_{n-1}) + (y_0, y_1, \dots, y_{n-1}) \\ &= (x_0 + y_0, x_1 + y_1, \dots, x_{n-1} + y_{n-1}) \\ &= (z_0, z_1, \dots, z_{n-1}) \\ &= \mathbf{z}\end{aligned}$$

How could we implement this using MPI?

```
1 void Vector_sum(double x[], double y[], double z[], int n) {  
2     int i;  
3  
4     for (i = 0; i < n; i++)  
5         z[i] = x[i] + y[i];  
6 } /* Vector_sum */
```

Step 1:

- Aggregating the tasks and assigning them to the cores
- If the number of components is **n** and we have **comm_sz** cores or processes
 - **define local Task**
 - $n = n / \text{comm_sz}$ i.e block partition

Step 2: Data distributions

- process 0 can prompt the user, read in the value, and broadcast the value to the other processes.
- **Better Approach :**
 - Entire vector that is on process 0 but only sends the needed components to each of the other processes.

Table 3.4 Different Partitions of a 12-Component Vector among Three Processes

Process	Components											
	Block				Cyclic				Block-Cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	+6'	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

Basic Communication Operations

- Processes need to exchange data with other processes.
- This exchange of data can significantly **impact the efficiency of parallel programs** by introducing **interaction delays** during their execution.
 - **One-to-All Broadcast and All-to-One Reduction**
 - Parallel algorithms often require a single process to send identical data to all other processes or to a subset of them.
 - **Scatter and Gather**
 - In the scatter operation, a single node sends a unique message of size m to every other node.
 - gather operation, or concatenation, in which a single node collects a unique message from each node.

Collective Communication and Computation Operations

- MPI provides an extensive set of functions for performing many commonly used collective communication operations.
- All the processes that belong to this communicator participate in the operation, and all of them must call the collective communication function.
- Barrier :
 - The barrier synchronization operation is performed in MPI using the MPI_Barrier function.
- Broadcast :
 - one-to-all broadcast operation is performed in MPI using the MPI_Bcast function.
 - MPI_Bcast sends the data stored in the buffer `buf` of process `source` to all the other processes in the group.

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

Collective Communication and Computation Operations

■ Reduction

- The **all-to-one reduction** operation is performed in MPI using the MPI_Reduce function.

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

■ Predefined reduction operations.

Value	15	17	11	12	17	11
Process	0	1	2	3	4	5

`MinLoc(Value, Process) = (11, 2)`

`MaxLoc(Value, Process) = (17, 1)`

Operation	
<code>MPI_MAX</code>	Maximum
<code>MPI_MIN</code>	Minimum
<code>MPI_SUM</code>	Sum
<code>MPI_PROD</code>	Product
<code>MPI_LAND</code>	Logical AND
<code>MPI_BAND</code>	Bit-wise AND
<code>MPI_LOR</code>	Logical OR
<code>MPI_BOR</code>	Bit-wise OR
<code>MPI_LXOR</code>	Logical XOR
<code>MPI_BXOR</code>	Bit-wise XOR
<code>MPI_MAXLOC</code>	max-min value-location
<code>MPI_MINLOC</code>	min-min value-location

Collective Communication and Computation Operations

■ Gather

- The **all-to-one reduction** operation is performed in MPI using the MPI_Reduce function.

```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int recvcount, MPI_Datatype recvdatatype, int target, MPI_Comm comm)
```

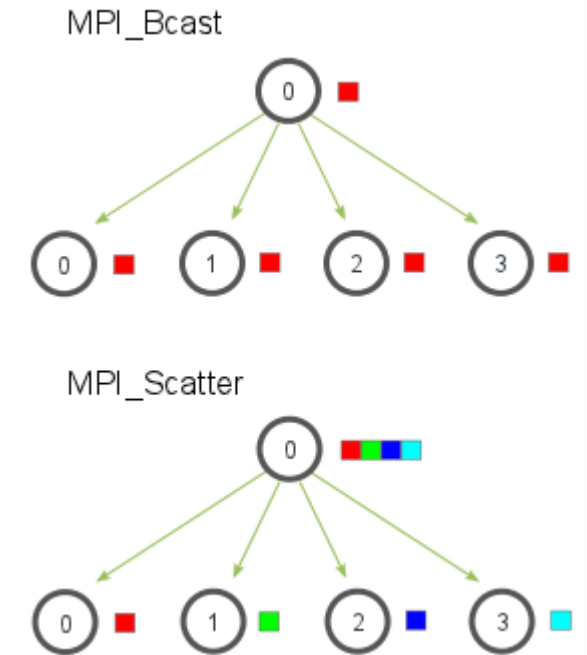
- Each process, including the `target` process, sends the data stored in the array `sendbuf` to the `target` process.
- As a result, if p is the number of processors in the communication `comm`, the target process receives a total of p buffers.
- The data is stored in the array `recvbuf` of the target process, in a rank order.
- The data is stored in the array `recvbuf` of the target process, in a rank order.

Collective Communication and Computation Operations

■ Scatter

- The scatter operation is performed in MPI using the MPI_Scatter function.

```
int MPI_Scatter(void *sendbuf, int sendcount,  
               MPI_Datatype senddatatype, void *recvbuf, int recvcount,  
               MPI_Datatype recvdatatype, int source, MPI_Comm comm)
```





Example : **Computing average of numbers with MPI_Scatter and MPI_Gather**

Tutorial 3

- **Write MPI program to print hello from each process in the comm world**
- **Write MPI program to do point-to-point communication:**
 - Master process: send msg with tag 1134
 - Master process: wait for msg with tag 4114
 - Slave process: receive msg with tag 1134
 - Slave process: send back msg with tag 4114
- **Write MPI program for sending and receiving**
 - Process 1 sends 4 characters to process 0
 - Process 0 receives an integer (4 bytes)