King Saud University

Parallel Processing

Project 3

MESSAGE PASSING INTERFACE

Students:

Al-Anoud Al-Subaie
Raghad Al-Suhaibani
Ftoon Al-Rubayea

Table of Contents:

Phase 1

1.Background :

1.1 Message Passing Interface (MPI):

1.1.1 Definition:

The Message passing interface Standard is a message passing library industry standard for parallel programming based on the consensus of the MPI forum which has over 40 participating organization, including vendors, researchers software library developers, and users .

The message passing interface effort began in the summer of 1991 when a small group of researchers started discussions at a mountain retreat in Austria.

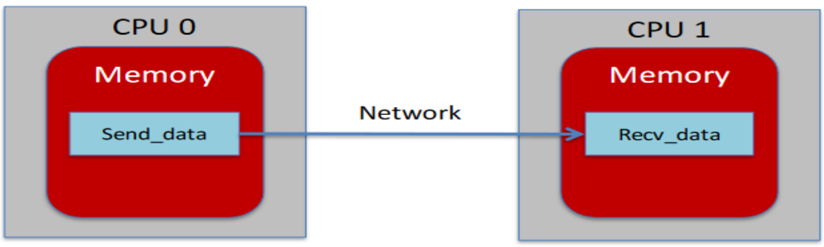
MPI “operations” are expressed as functions, subroutines, or methods, according to the appropriate language bindings for C and Fortran.

Processes communicate via calls to the MPI communication operations.MPI programs have operations to initialize the execution environment, and to control starting and terminating processes.Implementations: OpenMPI, MPICH2 .

An MPI program consists of autonomous processes that are able to execute theirown code in the sense of multiple instruction multiple data (MIMD) paradigm. AnMPI process can be interpreted in this sense as a program counter that addressestheir program instructions in the system memory, which implies that the programcodes executed by each process have not to be the same.

MPI processes can be collected into groups of specificsizethat can communicate in itsown environment where each message sent in a context must be received only in thesame context. Aprocess groupandcontexttogether form an MPI communicator.

A process is identified by itsrankin the group associated with a communicator.There is a default communicator MPI_COMM_WORLD whose group encompass all ini-tial processes, and whose context is default.



1.1.2 Functionality:

Message Passing Interface have many functionality and goles :

- Messages are used to coordinate parallel tasks that run on distributed but interconnected processors .
- Message Passing Interface enables system-independent parallel programming .
- process creation and management .
- language bindings for C and C++ and Fortran .
- point-to-point and collective communications .
- group and communicator concepts .
- Local communication between neighboring processors is faster and more scalable than global communication among all processors .
- The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs in C .
- MPI provides widely used standard for writing message passing programs.
- Standard defining how CPUs send and receive data .
- Vendor specific implementation adhering to the standard .
- Allows CPUs to “talk” to each other .

The MPI interface is meant to provide essential virtual topology, synchronization, and communication functionality between a set of processes (that have been mapped to nodes/servers/computer instances) in a language-independent way, with language-specific syntax (bindings), plus a few language-specific features. MPI programs always work with processes, but programmers commonly refer to the processes as processors. Typically, for maximum performance, each CPU (or core in a multi-core machine) will be assigned just a single process. This assignment happens at runtime through the agent that starts the MPI program, normally called mpirun or mpiexec.

2. Basic MPI Operations :

1. **MPI_INIT (int *argc, char ***argv)** : The operation initiates an MPI library and environment. The arguments argc and argv are required in C language binding only, where they are parameters of the main C program.

2**. MPI_FINALIZE ()** : The operation shuts down the MPI environment. No MPI routine can be called before MPI_INIT or after MPI_FINALIZE, with one exception MPI_INITIALIZED(flag), which queries if MPI_INIT has been called.

3 **. MPI_COMM_SIZE (comm, size) :** The operation determines the number of processes in the current communicator.The input argument commis the handle of communicator; the output argument size returned by the operation MPI_COMM_SIZE is the number of processes in the group of comm. If comm is MPI_COMM_WORLD then it represents the number of all active MPI processes.

4 . **MPI_COMM_RANK (comm, rank):** The operation determines the identifier of the current process within a communi-cator. The input argument commis the handle of the communicator . the output ar-gumentrankis an ID of the process from comm, which is in the range from 0 to size-1.

5**. Measuring time :** MPI_Wtime ();

6. **MPI processes can communicate in four different communication modes:**

- MPI_BARRIER (comm) : This operation is used to synchronize the execution of a group of processes specified within the communicator comm .
- Buffered(MPI_Bsend): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe) .
- Synchronous(MPI_Ssend): the send does not complete until a matching receive has begun. (Unsafe programs deadlock) .
- Ready(MPI_Rsend): user guarantees that a matching receive has been posted .

2.MPI Communicator :

Communicator :is like a box that groups processes together, allowing them to communicate .
Every communication is linked to a communicator, allowing the communication to reach different processes.

Communications can be either of two types :

1. **Point-to-Point (P2P):** Two processes in the same communicator are going to communicate .

- Use send and receive operations .
- Easiest P2P is blocking: process sending the message will wait until the process receiving has finished receiving all the information .
- Other type: nonblocking .

2. **Collective**: All the processes in a communicator are going to communicate together.

2.1 Point-to-Point Operations:

MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.

There are different types of send and receive routines used for different purposes. For example:

1.Synchronous send .

2.Blocking send / blocking receive .

3.Non-blocking send / non-blocking receive .

4.Buffered send .

5.Combined send/receive .

6. "Ready" send .

Any type of send routine can be paired with any type of receive routine.

MPI also provides several routines associated with send - receive operations, such as those used to wait for a message's arrival or probe to find out if a message has arrived.

Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.

Blocking:

- A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. 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 which means there is handshaking occurring with the receive task to confirm a safe send.
- 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.
- **MPI_Send** :

  Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse. Note that this routine may be implemented differently on different systems. The MPI standard permits the use of a system buffer but does not require it. Some implementations may actually use a synchronous send (discussed below) to implement the basic blocking send.
- **MPI_Recv** :

  Receive a message and block until the requested data is available in the application buffer in the receiving task.

- **MPI_Ssend** :

  Synchronous blocking send: Send a message and block until the application buffer in the sending task is free for reuse and the destination process has started to receive the message.

- **MPI_Sendrecv** :

  Send a message and post a receive before blocking. Will block until the sending application buffer is free for reuse and until the receiving application buffer contains the received message.

Non-blocking:

- Non-blocking send and receive routines behave similarly . they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
- 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 (your variable space) 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 and exploit possible performance gains.

Advantage:

•Avoids additional logic

•Overlap communication and computation

•Performance improvement

Disadvantage:

•Complexity; all requests must be handled

Waiting forces the process to go in "blocking mode” . The sending process will simply waitfor the request to finish.

Test Process with testing checks if the request can be completed: If it can, the request is automatically completed and the data transferred.

Blocking sends	**MPI_Send(buffer,count,type,dest,tag,comm)**
Non-blocking sends	**MPI_Isend(buffer,count,type,dest,tag,comm,request)**
Blocking receive	**MPI_Recv(buffer,count,type,source,tag,comm,status)**
Non-blocking receive	**MPI_Irecv(buffer,count,type,source,tag,comm,request)**

Buffer

Program (application) address space that references the data that is to be sent or received. In most cases, this is simply the variable name that is be sent/received. For C programs, this argument is passed by reference and usually must be prepended with an ampersand: **&var1**

2.2 Collective :

Collective functions involve communication among all processes in a process group (which can mean the entire process pool or a program-defined subset).

Types of Collective Operations:

- **Synchronization** - processes wait until all members of the group have reached the synchronization point.
- **Data Movement** - broadcast, scatter/gather, all to all.
- **Collective Computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

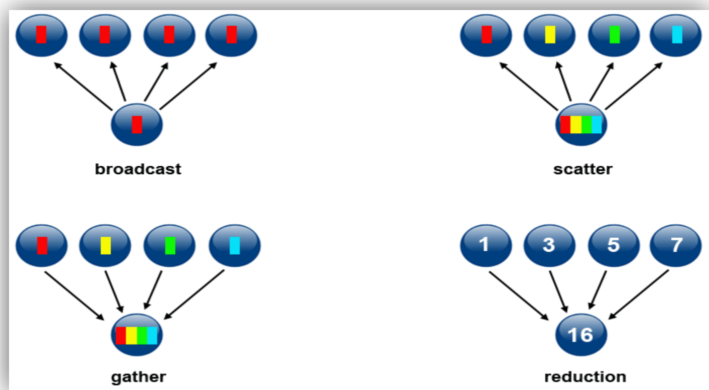
Collective communications allow exchange of information across all processes (of a communicator) :

1.Broadcast: One process sends a message to every other process $O(\log n)$ MPI_Bcast(buffer, 5, MPI_INT, 0, MPI_COMM_WORLD) .

2.Reduction: One process gets data from all the other processes and applies an operation on it (sum, minimum, maximum, etc.MPI_Reduce(&tmp, &result, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD);

3.Scatter: A single process partitions the data to send pieces to every other processint MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatypesendtype, void *recvbuf, int recvcount, MPI_Datatyperecvtype, int root, MPI_Commcomm);

4.Gather: A single process assembles the data from different process in a bufferint MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatypesendtype, void *recvbuf, int recvcount, MPI_Datatyperecvtype, int root, MPI_Commcomm) .



Phase 2

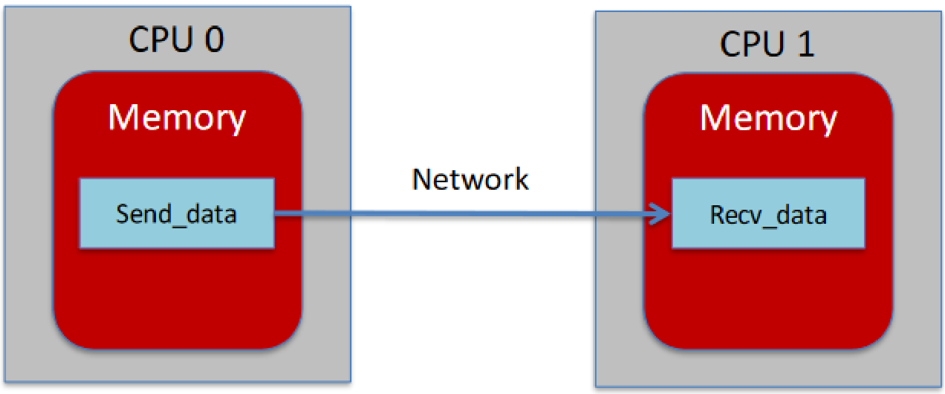
1.Background :

1.1 Message Passing Interface (MPI):

The Message passing interface Standard is a message passing library industry standard for parallel programming based on the consensus of the MPI forum which has over 40 participating organization, including vendors, researchers software library developers, and users .

Communications can be either of two types :

1. **Point-to-Point (P2P):** Two processes in the same communicator are going to communicate .
2. **Collective**: All the processes in a communicator are going to communicate together.



2. Problem :

P2 . Use MPI point-to-point communication to implement the broadcast and reduce functions. Compare the performance of your implementation with that of the MPI global operations MPI_BCAST and MPI_REDUCE for different data sizes and different number of processes. Use data sizes up to 10^4 doubles and up to allavailable number of processes. Plot and explain obtained results .

- In this project we will compare the performance point to point broadcast and reduce funcation with the global opraction MPI_BCAST and MPI_REDUCE for different data size and different number of processes and see what is more performance it point to point or a global opractions .

3.Libraries and functions we will use :

-Basic Libraries :

1. **MPI_INIT (int *argc, char ***argv)** : The operation initiates an MPI library and environment. The arguments argc and argvare required in C language binding only, where they are parameters of the main C program.

2**. MPI_FINALIZE ()** : The operation shuts down the MPI environment. No MPI routine can be called before MPI_INIT or after MPI_FINALIZE , with one exception MPI_INITIALIZED(flag), which queries if MPI_INIT has been called.

3**. MPI_COMM_SIZE (comm, size) :** The operation determines the number of processes in the current communicator.The input argument commis the handle of communicator; the output argument size returned by the operation MPI_COMM_SIZE is the number of processes in the group of comm. If comm is MPI_COMM_WORLD then it represents the number of all active MPI processes.

4. **MPI_COMM_RANK (comm, rank):** The operation determines the identifier of the current process within a communi-cator. The input argument commis the handle of the communicator . the output ar-gument rank is an ID of the process from comm, which is in the range from 0 to size-1.

-For caulcalet time of the function :

5**. Measuring time :** MPI_Wtime ();

6**.MPI_BARRIER (comm)** : This operation is used to synchronize the execution of a group of processes specified within the communicator comm .

-For Point-to-Point we will use :

7.**MPI_Send** : Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse.

8.**MPI_Recv :** Receive a message and block until the requested data is available in the application buffer in the receiving task.

Blocking sends	**MPI_Send(buffer,count,type,dest,tag,comm)**
Blocking receive	**MPI_Recv(buffer,count,type,source,tag,comm,status)**

- We will comper point-to-point with this funcations “Collective” :

9.**MPI_Bcast(buffer, 5, MPI_INT, 0, MPI_COMM_WORLD)** : One process sends a message to every other process. “one -to-all” .

10**. MPI_Reduce(&tmp, &result, 1, MPI_FLOAT, MPI_SUM, 0, MPI_COMM_WORLD)**: One process gets data from all the other processes and applies an operation on it (sum, minimum, maximum). ” all-to-one”.

Functionality of each student :

NAME	FUNCTIONALITY
Al-Anoud Al-Subaie	implementation, Report , presentation
Raghad Al-Suhaibani	implementation, Report , presentation
Ftoon Al-Rubayea	implementation, Report , presentation

Phase 3

1. Problem :

P2 . Use MPI point-to-point communication to implement the broadcast and reduce functions. Compare the performance of your implementation with that of the MPI global operations MPI_BCAST and MPI_REDUCE for different data sizes and different number of processes. Use data sizes up to 10^4 doubles and up to allavailable number of processes. Plot and explain obtained results .

2.MPI code :

2.1 Broadcasting :

1.main method :

```
int main(int argc, char** argv) {
    if (argc != 2) {
        fprintf(stderr, "Usage: compare_bcast num_elements\n");
        exit(1);
    }

    int num_elements = atoi(argv[1]);


    MPI_Init(NULL, NULL);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

```

The first things in our code we will doning initiates MPI by called function MPI_init then we will call function MPI_Comm_rank to determines the identifier of the current process within a communicator.

```c
    double total_my_bcast_time = 0.0;
    double total_mpi_bcast_time = 0.0;
    int i;
    int* data = (int*)malloc(sizeof(int) * num_elements);
    assert(data != NULL);


      // Time my_bcast
      // Synchronize before starting timing
      MPI_Barrier(MPI_COMM_WORLD);
      total_my_bcast_time -= MPI_Wtime();
      my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
      // Synchronize again before obtaining final time
      MPI_Barrier(MPI_COMM_WORLD);
      total_my_bcast_time += MPI_Wtime();

      // Time MPI_Bcast
      MPI_Barrier(MPI_COMM_WORLD);
      total_mpi_bcast_time -= MPI_Wtime();
      MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
      MPI_Barrier(MPI_COMM_WORLD);
      total_mpi_bcast_time += MPI_Wtime();
```

In this part will are called Broadcast function using point to point and the global MPI_Bcast to send data and calculate time using MPI_wtime() and for Synchronized we are using MPI_Barrier .the MPI_Bcast it's global operation used to One process sends a message to every other process .

```c

    // Print off timing information
    if (world_rank == 0) {
      printf("Data size = %d\n", num_elements * (int)sizeof(int));
      printf("my_bcast time = %lf\n", total_my_bcast_time);
      printf("MPI_Bcast time = %lf\n", total_mpi_bcast_time);
    }

    free(data);
    MPI_Finalize();
```

When the world rank equally zeor it's mean I am root and all processes are received . "one-to-all". Then we are called MPI_Finalize() to shut down the MPI environment .

2.my_bcast function point to point :

```c
void my_bcast(void* data, int count, MPI_Datatype datatype, int root,
              MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);

  if (world_rank == root) {
    // If we are the root process, send our data to everyone
    int i;
    for (i = 0; i < world_size; i++) {
      if (i != world_rank) {
        MPI_Send(data, count, datatype, i, 0, communicator);
      }
    }
  } else {
    // If we are a receiver process, receive the data from the root
    MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);
  }
}
```

The function my_bcast it's work like MPI_Bcast but work like point-to-point using send and receive function . the first things we are use MPI_Comm_size to determine the number of processes in the current communicator and use MPI_Send to Routine returns only after the application buffer in the sending task is free for reuse .MPI_Recv **to r**eceive a message and block until the requested data is available in the application buffer

2.1.1 Result :

```
anoud@anoud://home/anoud/Desktop$ time mpirun -n 2  ./my_bcast 10000
Data size = 40000
my_bcast time = 0.000037
MPI_Bcast time = 0.000031

real	0m0.425s
user	0m0.065s
sys	0m0.075s
anoud@anoud://home/anoud/Desktop$ time mpirun -n 2  ./my_bcast 100
Data size = 400
my_bcast time = 0.000007
MPI_Bcast time = 0.000003

real	0m0.433s
user	0m0.087s
sys	0m0.082s
```

We are using 2 processes with different data size and 10000 take more time than 100 and my_bcast function "point to point " take more time than MPI_Bcast .

```
anoud@anoud://home/anoud/Desktop$ time mpirun -n 3  ./my_bcast 10000
Data size = 40000
my_bcast time = 0.000055
MPI_Bcast time = 0.000047

real    0m0.411s
user    0m0.102s
sys     0m0.082s
anoud@anoud://home/anoud/Desktop$ time mpirun -n 3  ./my_bcast 100
Data size = 400
my_bcast time = 0.000010
MPI_Bcast time = 0.000003

real    0m0.446s
user    0m0.099s
sys     0m0.086s
```

We are using 3 processes with different data size and 10000 take more time than 100 and my_bcast function “point to point ” take more time than MPI_Bcast .

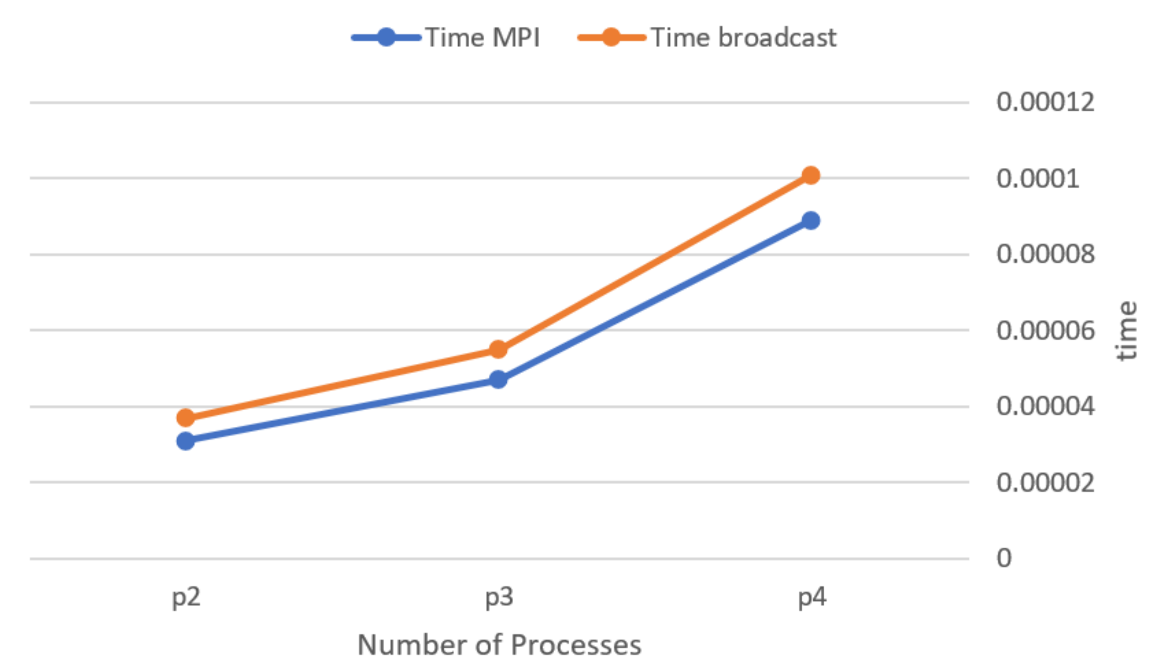
```
anoud@anoud: //home/anoud/Desktop
anoud@anoud://home/anoud/Desktop$ time mpirun -n 4  ./my_bcast 10000
Data size = 40000
my_bcast time = 0.000101
MPI_Bcast time = 0.000089

real    0m0.452s
user    0m0.137s
sys     0m0.118s
anoud@anoud://home/anoud/Desktop$ time mpirun -n 4  ./my_bcast 100
Data size = 400
my_bcast time = 0.000047
MPI_Bcast time = 0.000017

real    0m0.470s
user    0m0.106s
sys     0m0.135s
```

We are using 4 processes with different data size and 10000 take more time than 100 and my_bcast function “point to point ” take more time than MPI_Bcast .

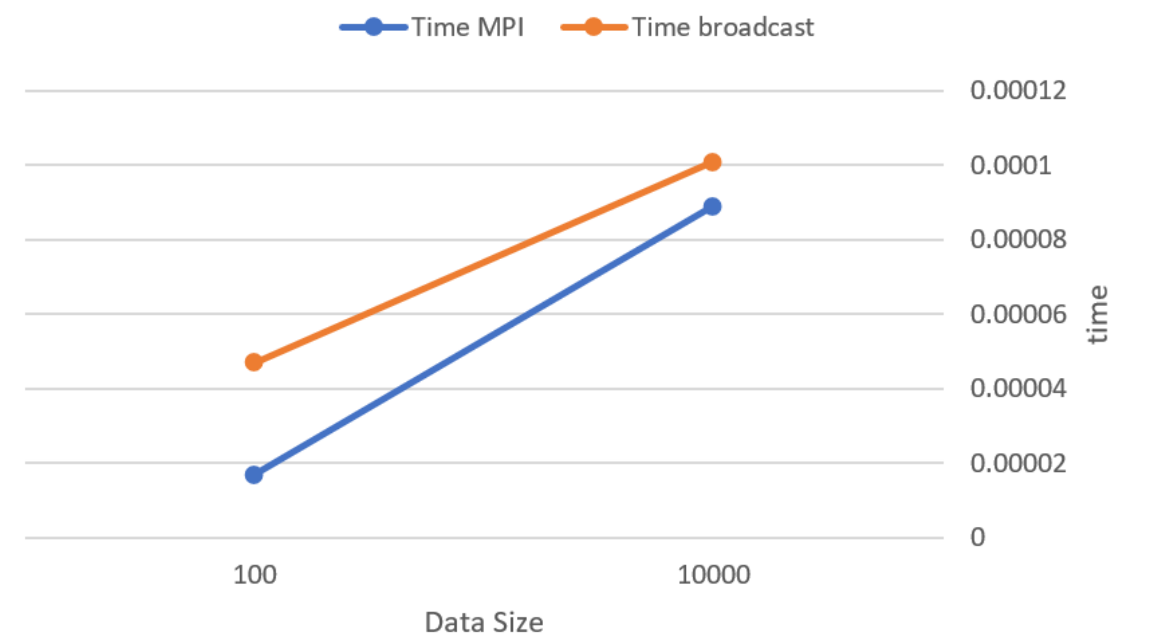
When we are add more prosess it will tack more time .

Plot broadcast:

Number of processes and data size 10000 :



Data Size and number of processes 4 :



2.2 Reduction :

1.main :

```c
int main(int argc, char** argv) {


    if (argc != 2) {
        fprintf(stderr, "Usage: avg num_elements_per_proc\n");
        exit(1);
    }

    int num_elements_per_proc = atoi(argv[1]);
    MPI_Init(NULL, NULL);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    double total_my_reduce_time = 0.0;
    double total_mpi_reduce_time = 0.0;

    // Create a random array of elements on all processes.
    srand(time(NULL)*world_rank);   // Seed the random number generator
    float *rand_nums = NULL;
    rand_nums = create_rand_nums(num_elements_per_proc);

    // Sum the numbers locally
    float local_sum = 0;
    int i;
    for (i = 0; i < num_elements_per_proc; i++) {
        local_sum += rand_nums[i];
    }

    // Print the random numbers on each process
    printf("Local sum for process %d - %f, avg = %f\n",
           world_rank, local_sum, local_sum / num_elements_per_proc);

```

The first things in our code we will doning initiates MPI by called function MPI_init then we will call function MPI_Comm_rank to determines the identifier of the current process within a communicator. We are use MPI_Comm_size to determine the number of processes in the current communicator . The srand it used to create a random array of elements on all processes .

```
// Reduce all of the local sums into the global sum
float global_sum = 0;

 // Time my_bcast
 // Synchronize before starting timing
MPI_Barrier(MPI_COMM_WORLD);
total_my_reduce_time -= MPI_Wtime();
 my_reduce(local_sum,&global_sum, 1, MPI_FLOAT, 0,MPI_COMM_WORLD);
// Synchronize again before obtaining final time
  MPI_Barrier(MPI_COMM_WORLD);
  total_my_reduce_time += MPI_Wtime();

  // Time MPI_Bcast
  MPI_Barrier(MPI_COMM_WORLD);
  total_mpi_reduce_time -= MPI_Wtime();
  MPI_Reduce(&local_sum , &global_sum , 1 , MPI_FLOAT,MPI_SUM ,0,MPI_COMM_WORLD);
  MPI_Barrier(MPI_COMM_WORLD);
  total_mpi_reduce_time += MPI_Wtime();

// Print the result
if (world_rank == 0) {

  printf("Total sum = %f, avg = %f\n", global_sum, global_sum / (world_size * num_elements_per_proc));
  printf("Data size = %d\n", num_elements_per_proc * (int)sizeof(int));
  printf("my_reduce time = %lf\n", total_my_reduce_time );
  printf("MPI_Reduce time = %lf\n", total_mpi_reduce_time);

}

// Clean up
free(rand_nums);

MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
//gettimeofday(&t1 , NULL);
//printf("process in %.2g seconds\n", t1.tv_sec - t0.tv_sec + 1E-6 * (t1.tv_usec - t0.tv_usec));
}
```

In this part will are called Reduce function using point to point and the global MPI_Reduce to send data and calculate time using MPI_wtime() and for Synchronized we are using MPI_Barrier .The MPI_Reduce it's global operation used to One process gets data from all the other processes . "all-to-one" .

2.my_reduce function point to point :

```
void my_reduce(float Ldata, float *Gdata,int count, MPI_Datatype datatype, int root,
              MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);
int i ;
float Gd = 0 ;
    for (i = 1; i < world_size; i++) {
      if (i != world_rank) {
        MPI_Send(&Ldata, count, datatype,root, 0, communicator);
      }
    }
  if (world_rank == root) {
    // If we are the root process, send our data to everyone
    Gd+=Ldata;
    int i;

    for (i = 0; i < world_size; i++) {
      if (i != world_rank) {
        MPI_Recv(&Ldata, count, datatype,i, 0, communicator, MPI_STATUS_IGNORE);
        Gd+=Ldata;
      }

    }

printf("sum = %f \n", Gd);
*Gdata = Gd;
 } else {
    // If we are a receiver process, receive the data from the root


    MPI_Send(&Gdata, count, datatype, root, 0, communicator);

  }
}
```

The function my_reduce is work like MPI_Reduce but work like point-to-point using send and receive function . the first things we are use MPI_Comm_rank to determines the identifier of the current process within a communicator. Then we are use MPI_Comm_size to determine the number of processes in the current communicator and use MPI_Send to Routine returns only after the application buffer in the sending task is free for reuse .MPI_Recv to receive a message and block until the requested data is available in the application buffer .

```c
// Creates an array of random numbers. Each number has a value from 0 - 1
float *create_rand_nums(int num_elements) {
  float *rand_nums = (float *)malloc(sizeof(float) * num_elements);
  assert(rand_nums != NULL);
  int i;
  for (i = 0; i < num_elements; i++) {
    rand_nums[i] = (rand() / (float)RAND_MAX);
  }
  return rand_nums;
}
```

This function is use for creates an array of random number and each number has a value from 0 -1 .

2.2.1 Result :

```
anoud@anoud: ~/Desktop
anoud@anoud:~/Desktop$ time mpirun -n 2  ./reduce_avg 10000
Local sum for process 0 - 4971.319336, avg = 0.497132
sum = 4971.319336
Local sum for process 1 - 4996.559570, avg = 0.499656
Total sum = 9967.878906, avg = 0.498394
Data size = 40000
my_reduce time = 0.000029
MPI_Reduce time = 0.000002

real    0m0.418s
user    0m0.060s
sys     0m0.089s
anoud@anoud:~/Desktop$ time mpirun -n 2  ./reduce_avg 100
Local sum for process 0 - 54.682476, avg = 0.546825
sum = 54.682476
Total sum = 105.168777, avg = 0.525844
Data size = 400
my_reduce time = 0.000011
MPI_Reduce time = 0.000002
Local sum for process 1 - 50.486298, avg = 0.504863

real    0m0.460s
user    0m0.112s
sys     0m0.070s
```

We are using 2 processes with different data size and 10000 take more time than 100 and my_reduce function “point to point ” take more time than MPI_Reduce.

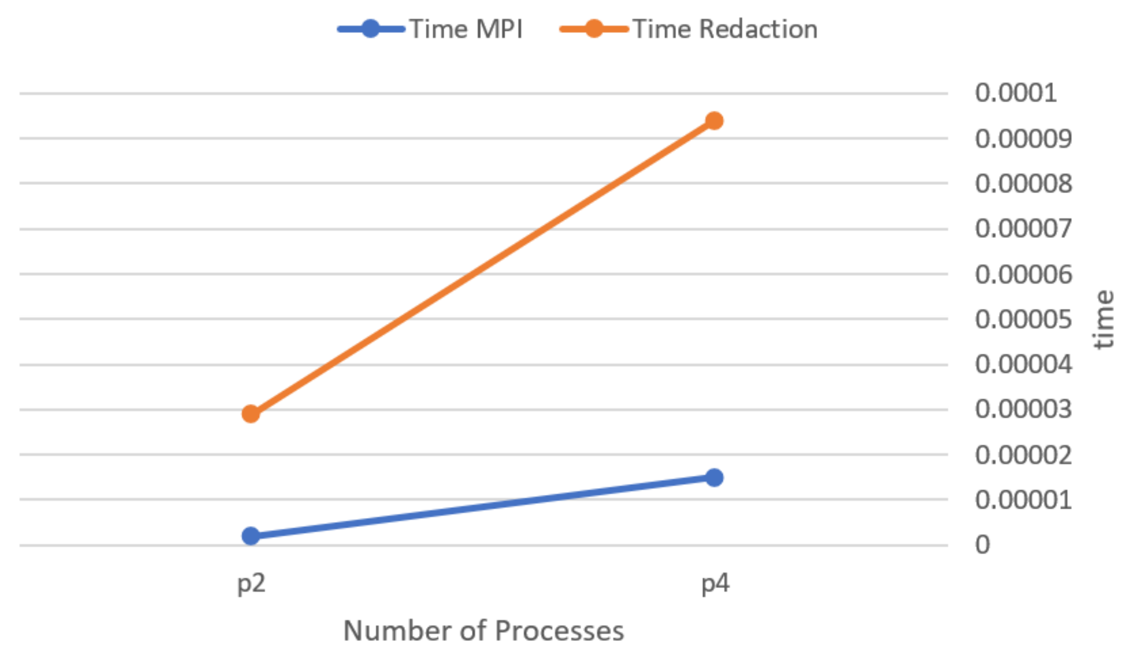
```
anoud@anoud:~/Desktop$ time mpirun -n 4  ./reduce_avg 10000
Local sum for process 2 - 5001.732910, avg = 0.500173
Local sum for process 0 - 4971.319336, avg = 0.497132
Local sum for process 1 - 5012.421875, avg = 0.501242
Local sum for process 3 - 5013.390625, avg = 0.501339
sum = 19998.865234
Total sum = 19998.863281, avg = 0.499972
Data size = 40000
my_reduce time = 0.000094
MPI_Reduce time = 0.000015

real    0m0.469s
user    0m0.140s
sys     0m0.137s
anoud@anoud:~/Desktop$ time mpirun -n 4  ./reduce_avg 100
Local sum for process 0 - 54.682476, avg = 0.546825
Local sum for process 1 - 53.466312, avg = 0.534663
Local sum for process 3 - 48.139839, avg = 0.481398
Local sum for process 2 - 43.484722, avg = 0.434847
sum = 199.773346
Total sum = 199.773346, avg = 0.499433
Data size = 400
my_reduce time = 0.000060
MPI_Reduce time = 0.000005
```

We are using 4 processes with different data size and 10000 take more time than 100 and my_reduce function “point to point ” take more time than MPI_Reduce .

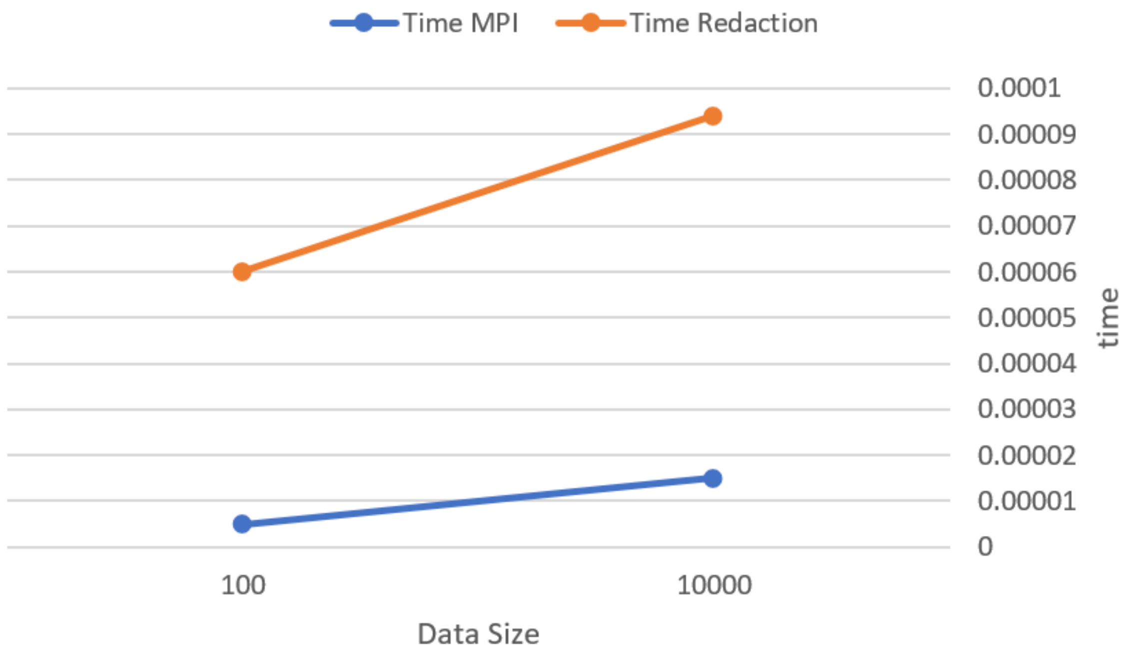
-When we are add more prosess it will tack more time .

Plot :

Number of processess and data size 10000 :



Data Size and number of processes 4 :



3. Conclusion :

Based on the result when we add more processes or add more data size it will take more time than little processes and little date size .The MPI_Redac and MPI_Bcast it more performance than point to point implemention.