FRANKFURT UNIVERSITY OF APPLIED SCIENCES

Develop a parallel application that does multiply two Matrices C and MPI

Program : High Integrity System

Course: Cloud Computing

Professor : Christian Baun

Present By : Shamima Akhter

ID: 1188387

Message Passing Interface (*MPI*)

- Distributed memory architectures which standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures.

- MPI is a ***specification*** for the developers and users of message passing libraries.

- MPI consists of
  - a header file mpi.h
  - a library of routines and functions,
  - and a runtime system.

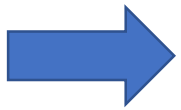
- MPI is for parallel computers, clusters, and heterogeneous networks.

- MPI used in many cases like when a master process needs to broadcast information to all of its worker processes.

- MPI can be used with C/C++, Fortran, and many other languages.

Algorithm : Matrix Multiplication with MPI

- Start with two matrices A is m*n and B is n*p.
- The product C = A*B is a matrix of m*p.
- The multiplication "row by column" gives a complexity of O(m*n*p).

Followed: Parallel Implementation  **Linear Partitioning**

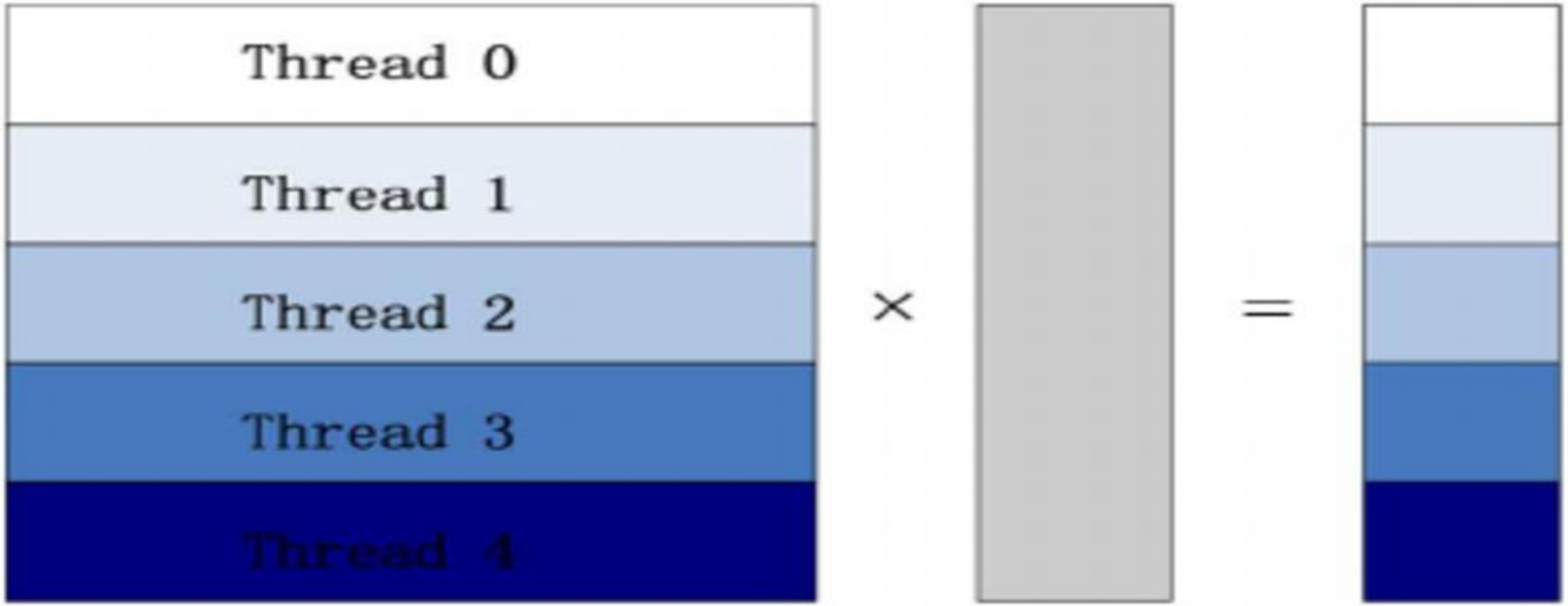
1. Scatter A to LocalA and Broatcast B.
2. Compute LocalC = localA* B;
3. Gather LocalC to C;

Matrix Multiplication

Advantages:

- Execution times reduce and the speedup increases.
- Simple computation for each processor.
- Distribution of each element localA[i][j], the columns of B must be traversed.

Process : Linear Matrix Multiplication



Linear Matrix Multiplication

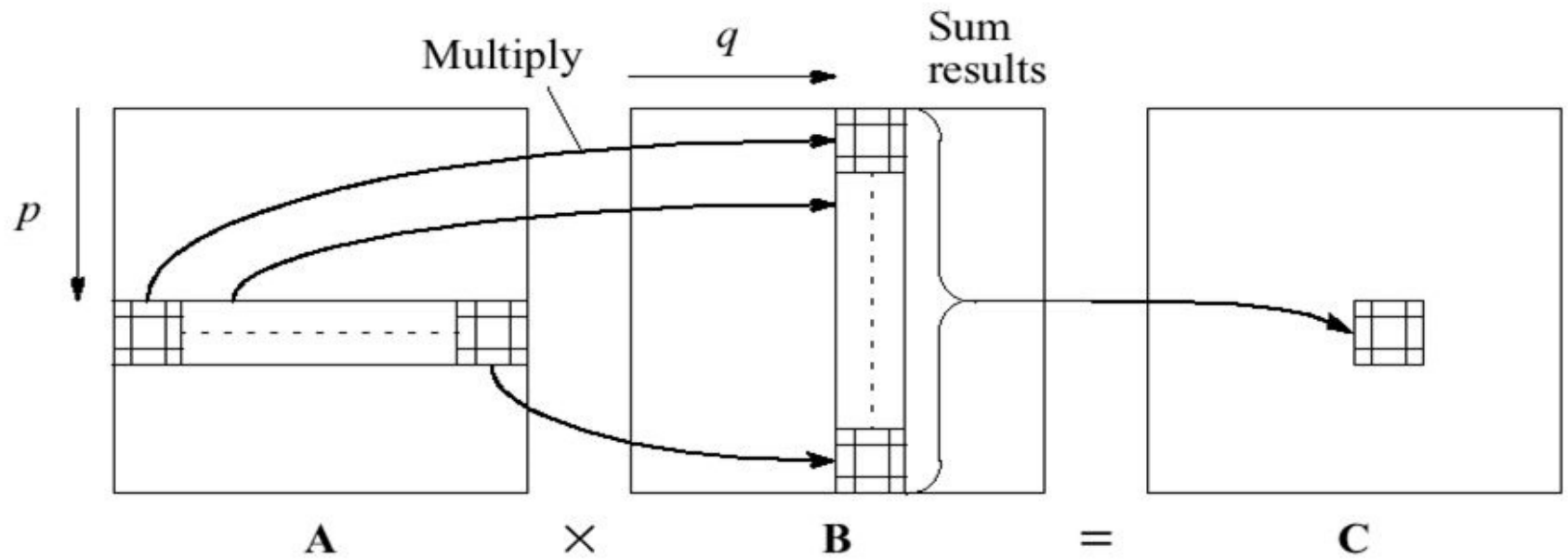


Figure 10.4 Block matrix multiplication.

Algorithm 1: Broadcasting with MPI_Bcast

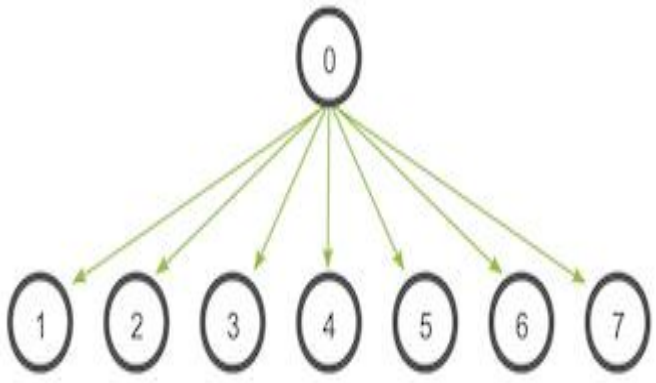


Fig: Communication pattern of a broadcast.

- A *broadcast* is one of the standard collective communication techniques.
- During a broadcast, one process sends the same data to all processes in a communicator.
- One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.
- In MPI, broadcasting can be accomplished by using MPI_Bcast

In this example, process zero is the root process, and it has the initial copy of data. All of the other processes receive the copy of data

Algorithm 1 : Broadcasting with MPI_Bcast

Broadcasting with MPI_Send and MPI_Recv

```
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);
  }
}
int main(int argc, char** argv) {
  MPI_Init(NULL, NULL);

  int world_rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

  int data;
  if (world_rank == 0) {
    data = 100;
    printf("Process 0 broadcasting data %d\n", data);
    my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
  } else {
    my_bcast(&data, 1, MPI_INT, 0, MPI_COMM_WORLD);
    printf("Process %d received data %d from root process\n", world_rank, data);
  }

  MPI_Finalize();
}
```

MPI_Bcast implementation & Comparison

```
for (i = 0; i < num_trials; i++) {
  // 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();
}

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

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

Comparison of MPI_Bcast with MPI_Send and MPI_Recv

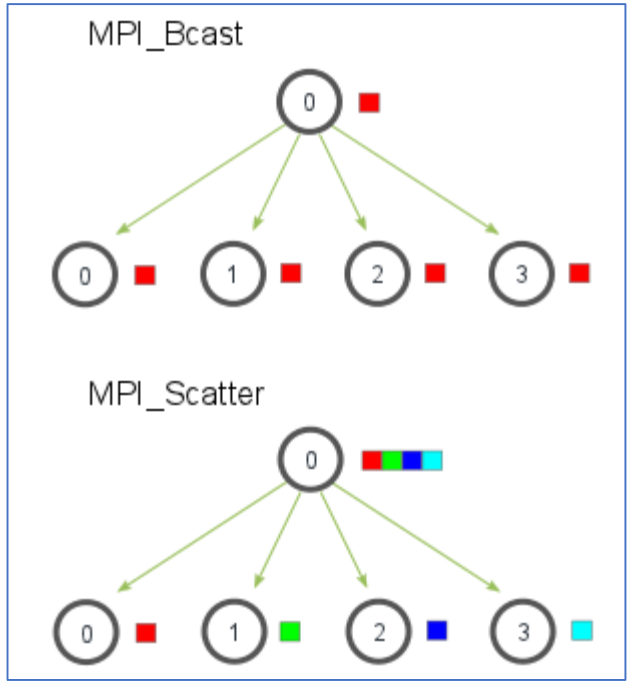
Processors	my_bcast	MPI_Bcast
2	0.0344	0.0344
4	0.1025	0.0817
8	0.2385	0.1084
16	0.5109	0.1296

Algorithm 2: Scatter with MPI_Scatter

- MPI_Scatter is a collective routine that is very similar to MPI_Bcast.
- MPI_Scatter involves a designated root process sending data to all processes in a communicator.

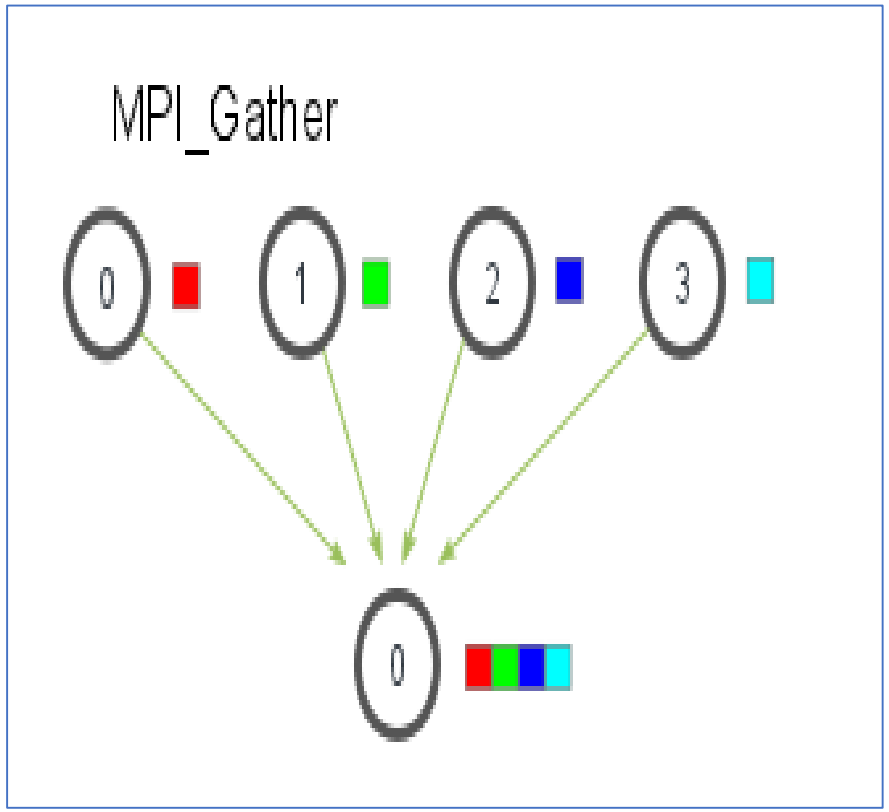
Small difference between MPI_Bcast and MPI_Scatter.

- MPI_Bcast sends the same piece of data to all processes while MPI_Scatter sends chunks of an array to different processes.



Algorithm 2: MPI_Gather

- MPI_Gather is the inverse of MPI_Scatter.
- Instead of spreading elements from one process to many processes, MPI_Gather takes elements from many processes and gathers them to one single process.
- Similar to MPI_Scatter, MPI_Gather takes elements from each process and gathers them to the root process.
- The elements are ordered by the rank of the process from which they were received.
- The function prototype for MPI_Gather is identical to that of MPI_Scatter



MPI Functions

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype datatype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

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

```
MPI_Bcast(
    void* data,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm communicator)
```

```
MPI_Scatter(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```

```
MPI_Gather(
    void* send_data,
    int send_count,
    MPI_Datatype send_datatype,
    void* recv_data,
    int recv_count,
    MPI_Datatype recv_datatype,
    int root,
    MPI_Comm communicator)
```

Configurations

Settings to Run MPI on Local Virtual

- Install oracle Virtual Box to install ubuntu.
- Install MPI on Master node.
- Run Code With MPI in local Virtual
  - Compile command: mpicc filename.c –o compile_filename
  - Run Command: mpiexec –np process_number ./compile_filename

Test on our University Network

- Install OpenVPN
- Install ssh – to connect with remote node.
- Connect OpenVPN and transfer file with ssh and test.

TEST CASES

Process	Matrix Size	Algorithm 1 : Sequential Algorithm with MPI_Send and MPI_Receive (Seconds)	Algorithm 2 : MPI_BroadCast , MPI_Scatter/Gather Matrix Algorithm (Seconds)
2	10	0.001274	0.000292
10	100	0.275533	0.122409
100	100	5.537235	2.791060
10	500	3.727987	4.329222
100	500	21.43198	9.163678
10	1000	43.39623	39.553845
100	1000	67.97839	41.028296

Test Summary

From the last test cases , noticed that :

- It would not always beneficial, if we grow the number of process proportional to our problem size.
- Sometimes less number of process work faster for a particular size of problem.
- If our problem size is more big enough then we can increase our process to divide the problems.
- For a small problem, specific process or small number of process is perform faster.

References

1. https://www.mathsisfun.com/algebra/matrix-multiplying.html
2. http://mpitutorial.com/tutorials/mpi-broadcast-and-collective-communication/
3. http://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/
4. http://mpi.deino.net/mpi_functions/MPI_Barrier.html
5. https://stackoverflow.com/questions/9269399/sending-blocks-of-2d-array-in-c-using-mpi
6. https://stackoverflow.com/questions/40080362/how-to-use-mpi-scatter-and-gather-with-array
7. https://stackoverflow.com/questions/29415663/how-does-mpi-in-place-work-with-mpi-scatter
8. http://www.cs.umanitoba.ca/~comp4510/examples.html

Questions