IIT DELHI

COL380: Introduction to Parallel and Distributed Programming (Self Study)

Assignment-2 Report

Prof. Rijurekha Sen

Souvagya Ranjan Sethi (2019CS10405)
Department of Computer Science & Engineering

July 03, 2023

1 Algorithm

The algorithm is parallelized using MPI Proccesses. The matrices which are to be multiplied are distributed among the processes and each process computes a particular section of the output matrix.

Algorithm 1: Parallel Matrix multiplication using MPI processes

Data: **n** ← the length of the matrix
Result: Output result
if *(rank == 0)* **then**
  Initialise matrix A(n*32) and B(32*n);
  Broadcast matrix B to all other processes;
  Distribute the rows of matrix A(n*32) among the worker processes and send respective rows to the worker process;
  Create local matrices C_serial and C_parallel;
  Compute the C_serial by performing matrix multiplication of A B;
  Gather the local C matrices from the worker processes and store in appropriate position of the C_parallel;
  Check whether C_serial is equal to C_parallel;
else
  Receive the Brodcasted matrix B from the root process(rank=0);
  Receive the respective rows of matrix A from the root process;
  Perform matrix multiplication of the respective rows of A with matrix B and store in local C;
  Send the local matrix C to the root process;
end

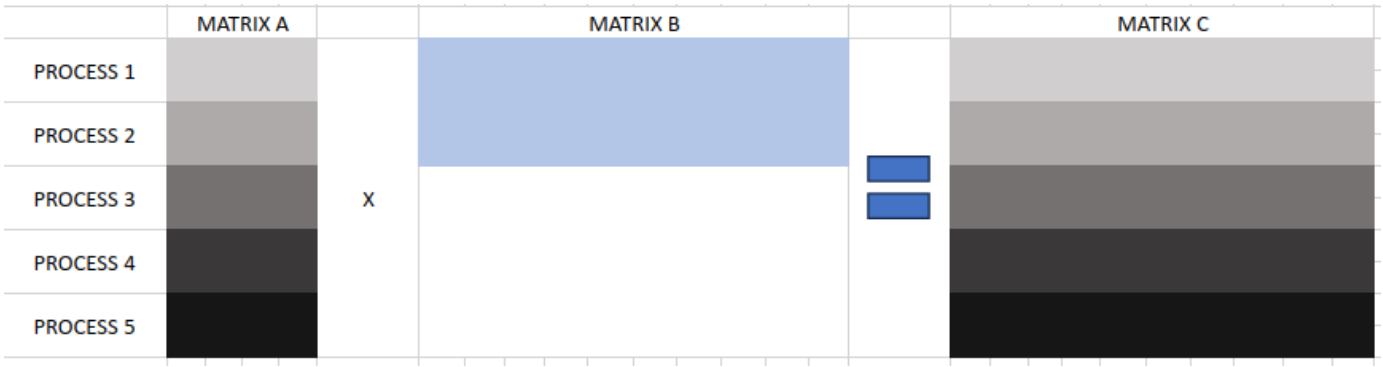


Figure 1: Pictorial representation of the algorithm

1.1 Key Points in Algorithm

- The matrices initialised in the root process contains floating point numbers in the range [0,1].
- The matrices are stored in the form of 1D arrays in row-major order. The length of the array is rowxcol.

- The root process computed the time taken for parallel matrix multiplication only and not the running time of whole program.
- The worker process perform normal matrix multiplication of a limited number of rows of A with B.
- There is no restriction of the value of n. The number of rows can be any integer value. I have handled the variations in n in the algorithm where I have distributed the rows unevenly so that all the rows are calculated properly and no error is occured.
- The number of MPI processes N has to be greater than 1. It can be 2,4,8 etc. This is because the root process doesnot do any computation. It only scatters and gathers the matrices to/from the worker processes. So, if N = 1, it means that the worker process has to do the computations too.
- The distribution of the rows of matrix A has been as such:

```
    for(int i = 1; i < size; i++){
        start_index = (i-1) * n / (size-1);
        end_index = i * n / (size-1);
        if(i == size-1){
            end_index = n;
        }
    }
```

- The Time taken by the parallel matrix multiplication is calculated using the high resolution clock module of the chrono library.

```
    #include <chrono>
    chrono::high_resolution_clock::time_point start_time, end_time;
    start_time = chrono::high_resolution_clock::now();
    end_time = chrono::high_resolution_clock::now();
    chrono::duration<double> time_elapsed = end_time - start_time;
    double seconds = time_elapsed.count();
    cout << "Time elapsed: " << seconds << " nano-seconds" << endl;
```

2 Different Parallel Implementation

2.1 Point-to-Point Communication

2.1.1 Blocking Communication

The Blocking P2P communication between MPI processes include:

1. MPI_Send
2. MPI_Recv

In this Implementation, MPI_Send & MPI_Recv was used to communicate information between Processes. The Root Process splits the Matrix A and sends the respective chunks into the worker process.

```
for(int i = 1; i < size; i++){
    MPI_Send(b, 32*n, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
}
for(int i = 1; i < size; i++){
    start_index = (i-1) * n / (size-1);
    end_index = i * n / (size-1);
    if(i == size-1){
        end_index = n;
    }
    MPI_Send(&a[start_index*32], (end_index - start_index)*32, MPI_FLOAT, i,
    0, MPI_COMM_WORLD);
}
for(int i = 1; i < size; i++){
    start_index = (i-1) * n / (size-1);
    end_index = i * n / (size-1);
    if(i == size-1){
        end_index = n;
    }
    MPI_Recv(&c_parallel[start_index*n], (end_index - start_index)*n,
    MPI_FLOAT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

The worker Processes accordingly call MPI_Recv & MPI_Send to communicate.

```
float *b_worker = new float[32*n];
MPI_Recv(b_worker, 32*n, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
start_index = (rank-1) * n / (size-1);
end_index = rank * n / (size-1);
if(rank == size-1){
    end_index = n;
}
float *a_worker = new float[(end_index - start_index)*32];
MPI_Recv(a_worker, (end_index - start_index)*32, MPI_FLOAT, 0, 0,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);

float *c_worker = new float[(end_index - start_index)*n];
for(int i = 0; i < end_index - start_index; i++){
    for(int j = 0; j < n; j++){
        c_worker[i*n + j] = 0;
        for(int k = 0; k < 32; k++){
            c_worker[i*n + j] += a_worker[i*32 + k] * b_worker[k*n + j];
        }
    }
}
MPI_Send(c_worker, (end_index - start_index)*n, MPI_FLOAT, 0, 0,
    MPI_COMM_WORLD);
```

2.1.2 Non-Blocking Communication

MPI_Isend & MPI_Irecv are the non-blocking point-to-point communication methods between MPI Processes. Here, the root process send the matrix to the worker process. But instead of waiting till all the data has been transferred, it performs the next opeation.

So, in order to synchronise the communication, we need to ensure that all the data from the previous instruction has been tranferred. So, we need to wait.

```
    MPI_Wait(&request, &status);
```

2.2 Collective Communication

The Collective Communication between MPI Processes include:

1. MPI_Bcast
2. MPI_Scatter
3. MPI_Gather
4. MPI_AlltoAll

In this Implementation, instead of Send and Recv, Broadcasting the common message to all the worker Process is done using MPI_Bcast. Also if different messages are to be send to the worker processes, then MPI_Scatterv and MPI_Gatherv is used. In this, all the processes simultaneously call the MPI Instructions.

```
MPI_Bcast(b, 6*n, MPI_FLOAT, 0, MPI_COMM_WORLD);
int sendcounts[size];
int displs[size];
sendcounts[0] = 0;
displs[0] = 0;
for(int i = 1; i < size; i++){
    start_index = (i-1) * n / (size-1);
    end_index = i * n / (size-1);
    if(i == size-1){
        end_index = n;
    }
    sendcounts[i] = (end_index - start_index)*6;
    displs[i] = start_index*6;
}
int recvcount = sendcounts[rank];
a_worker = new float[recvcount];
MPI_Scatterv(a, sendcounts, displs, MPI_FLOAT, a_worker, recvcount,
   MPI_FLOAT, 0, MPI_COMM_WORLD);
if(rank != 0){
    c_worker = new float[(sendcounts[rank]/6)*n];
```

```
    for(int i = 0; i < sendcounts[rank]/6; i++){
        for(int j = 0; j < n; j++){
            c_worker[i*n + j] = 0;
            for(int k = 0; k < 6; k++){
                c_worker[i*n + j] += a_worker[i*6 + k] * b[k*n + j];
            }
        }
    }
}
int recvcounts_2[size];
int displs_2[size];
recvcounts_2[0] = 0;
displs_2[0] = 0;
for(int i = 1; i < size; i++){
    start_index = (i-1) * n / (size-1);
    end_index = i * n / (size-1);
    if(i == size-1){
        end_index = n;
    }
    recvcounts_2[i] = (end_index - start_index)*n;
    displs_2[i] = start_index*n;
}
MPI_Gatherv(c_worker, recvcounts_2[rank], MPI_FLOAT, c_parallel,
   recvcounts_2, displs_2, MPI_FLOAT, 0, MPI_COMM_WORLD);
```

3 Observations

The submitted zip file contains 3 code files, 1 Makefile a report. The Makefile contains the instruction to compile the code. In order to change the arguments of the program or to compile different programs, you can modify the Makefile.

```
CC=mpic++
CFLAGS=-std=c++11 -g -lmpi -O2
exec: code1.cpp
        $(CC) $(CFLAGS) code1.cpp -o exec
run: code1.cpp
        mpirun -np 4 ./exec 10000
clean:
        rm -rf *.o exec
```

In order to compile & run the code, you need to do the following steps:

```
    make clean
    make
    make run
```

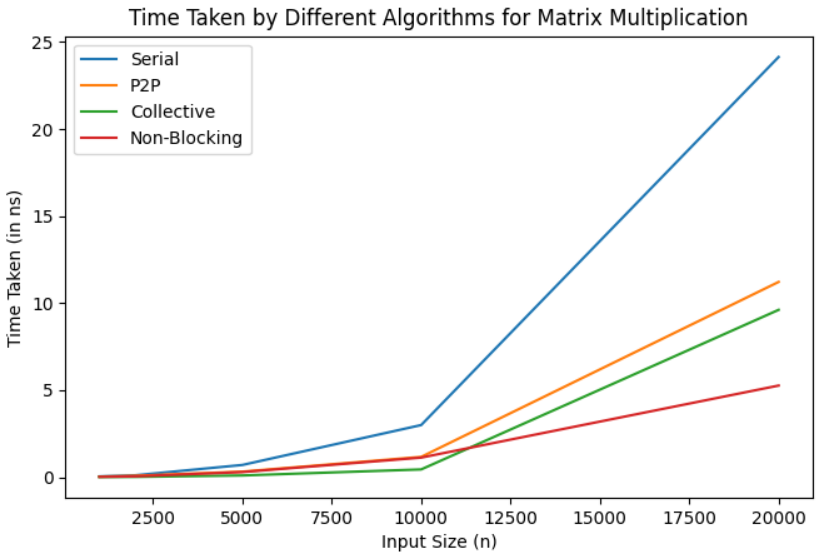
The Code files include:

- **P2P.cpp**: This file contains the parallel implementation of the matrix multiplication using the Blocking Point-to-Point Communication between MPI Processes. The root process calculated the time taken for matrix multiplication for both the serial and parallel methods.
- **Collective.cpp**: This file contains the implementation of the matrix multiplication using the MPI Collective communication between the processes. All the collectively calculate time but the root process displays the time it had calculated.
- **NonBlocking.cpp**: This file contains the non-blocking implementation using MPI_Isend & MPI_Irecv instructions. Here again, the root process calculates and displays the time.

The codes were run using the make run instructions for different values of N. The number of processes was fixed to 4.

Table 1: Time Taken by Different Implementations (in ns)

n	Serial	P2P	Collective	Non-Blocking
1000	0.0425	0.03224	0.00912	0.0174943
2000	0.10757	0.07823	0.0160754	0.047785
5000	0.7013	0.3181	0.0970647	0.292637
10000	2.9930	1.1646	0.444095	1.12285
20000	24.1369	11.216	9.61187	5.26195



4 Findings

- As the matrix size nx32 increases, the time taken by all the processes increase.
- Among different parallel implementations of the matrix multiplications using MPI Processes, the non-blocking point-to-point communication between the processes is the most efficient. It does not wait for the communication to complete but does other operations which increases efficiency. But it needs to be synchronised to prevent race conditions.
- The MPI Collective takes less time than MPI Blocking point-to-point communication. This is because, in MPI_Send MPI_Recv, we send the data one by one to all processes. But in collective such as MPI_Bcast, all the processes simultaneosuly communicate.
- the time taken by parallel implemenation in not exactly one-fourth (N=4) of the serial implementaion. This is due to the overhead of message communication between proceses.