HW5: MPI Matrix Multiplication

Rahul Kumar, Department of Mechanical Engineering, University of Tennessee, Chattanooga, 37403

Abstract

This study explores the parallelization of matrix multiplication using MPI, comparing sequential and parallel implementations. Two matrices, A of size $N \times 32$ and B of size $32 \times N$, are multiplied to produce an $N \times N$ result. The parallel implementation distributes the workload among multiple processes, allowing each process to compute a subset of the result matrix. Three communication strategies are evaluated: blocking point-to-point (P2P), collective communication, and non-blocking P2P. Execution times for sequential and parallel approaches are measured to demonstrate the performance gains and scalability of parallel computation. The results validate the correctness of parallel algorithms and illustrate that the time complexity of matrix multiplication remains theoretically $O(N^2)$ with improved practical performance due to process-level parallelism.

I. Introduction

Matrix multiplication is a fundamental operation in many scientific and engineering applications, including linear algebra, machine learning, physics simulations, and computer graphics. As the size of the matrices increases, the computational cost of sequential multiplication grows rapidly, with a theoretical time complexity of $O(N^2K)$ for multiplying an $N \times K$ matrix by a $K \times N$ matrix. To improve performance, parallel computing techniques can be applied to distribute the workload across multiple processors.

In this work, we implement and compare three parallel approaches to matrix multiplication using the Message Passing Interface (MPI):

A. Blocking Point-to-Point Communication (P2P)

In the blocking P2P approach, the root process (typically rank 0) explicitly sends portions of the input matrix A to each worker process using `MPI_Send`, and each worker process computes its assigned subset of the output matrix C. Once computation is complete, worker processes send their results back to the root process using `MPI_Send` again. Blocking communication ensures that data transfer is completed before the program proceeds, which simplifies programming but may lead to idle times while processes wait for messages to be delivered.

B. Collective Communication

Collective communication uses built-in MPI routines, such as `MPI_Scatter`, `MPI_Gather`, and `MPI_Bcast`, to handle data distribution and collection among all processes. The root process broadcasts the matrix B to all workers, while portions of A are scattered to each process. After local computation, results are gathered back into the final matrix C on the root process. Collective operations are optimized by MPI implementations and often provide better performance than manually coded P2P communication.

C. Non-Blocking Point-to-Point Communication

Non-blocking P2P communication employs `MPI_Isend` and `MPI_Irecv` to initiate data transfers without waiting for them to complete immediately. This allows computation and communication to overlap, reducing idle time and improving overall performance, especially when communication cost is significant. After initiating the transfers, processes use `MPI_Wait` or `MPI_Waitall` to ensure completion before accessing the data. This approach is more complex to implement but can achieve higher efficiency in large-scale computations.

The main objective of this study is to implement these three parallelization strategies for multiplying an $N \times 32$ matrix A with a $32 \times N$ matrix B, measure and compare their execution times with sequential computation, and verify the correctness of the parallel algorithms. By analyzing the performance under different communication strategies and varying the number of processes, we aim to demonstrate the scalability and efficiency of parallel matrix multiplication.

II. Parallel Matrix Multiplication Using MPI

Matrix multiplication is a fundamental operation in many scientific and engineering applications. When matrix sizes become large, the computational cost grows rapidly, making sequential computation inefficient. Parallel computing with MPI allows the workload to be distributed across multiple processes. In this study, we implement three MPI-based approaches: **blocking point-to-point (P2P)**, **collective communication**, and **non-blocking P2P**.

A. Blocking Point-to-Point Communication (P2P)

In the blocking P2P approach, the root process explicitly sends data to each worker process using `MPI_Send`. Each worker process computes its assigned portion of the output matrix and sends the result back to the root process. Communication is *blocking*, meaning the process waits until the message transfer completes. This approach is simple to implement but may cause idle time while processes wait. Complete main script can be referred from Appendix B.

```
// Send rows of A to worker processes
if(rank == 0) {
    for(int i = 1; i < size; i++)
        MPI_Send(&A[offset*K], send_rows*K, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
    memcpy(A_local, A, local_rows*K*sizeof(float));
} else {
    MPI_Recv(A_local, local_rows*K, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE
        );
}

// Local computation
Multiply_serial(A_local, B, C_local, local_rows, K, N);

// Send results back to root
if(rank == 0) {
    for(int i = 1; i < size; i++)
        MPI_Recv(&C[offset*N], recv_rows*N, MPI_FLOAT, i, 1, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    memcpy(C, C_local, local_rows*N*sizeof(float));
} else {
    MPI_Send(C_local, local_rows*N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
}
```

B. Collective Communication

Collective communication uses MPI routines that manage data distribution and collection automatically. The root process broadcasts matrix B to all workers using `MPI_Bcast`, scatters rows of A with `MPI_Scatterv`, and gathers results back using `MPI_Gatherv`. Collective communication is optimized by MPI libraries and often performs better than manual P2P communication. Complete main script can be referred from Appendix C.

```
// Broadcast B to all processes
MPI_Bcast(B, K*N, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Scatter rows of A
MPI_Scatterv(A, sendcounts, displs, MPI_FLOAT, A_local,
             local_rows*K, MPI_FLOAT, 0, MPI_COMM_WORLD);

// Local computation
Multiply_serial(A_local, B, C_local, local_rows, K, N);

// Gather results back to root
MPI_Gatherv(C_local, local_rows*N, MPI_FLOAT, C, recvcounts, recvdispls,
            MPI_FLOAT, 0, MPI_COMM_WORLD);
```

C. Non-Blocking Point-to-Point Communication

Non-blocking P2P communication allows computation to overlap with data transfer. Using `MPI_Isend` and `MPI_Irecv`, processes initiate communication without waiting for completion. After performing computations,

they use `MPI_Wait` to ensure messages have been sent or received before accessing the data. This approach reduces idle time and can improve performance for large-scale problems, though it is slightly more complex to implement. Complete main script can be referred from Appendix D.

```
MPI_Request send_req, recv_req;

// Root initiates non-blocking sends
if(rank == 0) {
    for(int i = 1; i < size; i++)
        MPI_Isend(&A[offset*K], send_rows*K, MPI_FLOAT, i, 0, MPI_COMM_WORLD, &
            send_req);
    memcpy(A_local, A, local_rows*K*sizeof(float));
} else {
    MPI_Irecv(A_local, local_rows*K, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &recv_req);
}

// Local computation
Multiply_serial(A_local, B, C_local, local_rows, K, N);

// Wait for communication to complete
MPI_Wait(&recv_req, MPI_STATUS_IGNORE);

// Send results back non-blocking
MPI_Isend(C_local, local_rows*N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &send_req);
if(rank == 0) {
    for(int i = 1; i < size; i++) {
        MPI_Irecv(&C[offset*N], recv_rows*N, MPI_FLOAT, i, 1, MPI_COMM_WORLD, &
            recv_req);
        MPI_Wait(&recv_req, MPI_STATUS_IGNORE);
    }
    memcpy(C, C_local, local_rows*N*sizeof(float));
}
```

By implementing these three methods, we can evaluate the trade-offs between programming complexity, communication overhead, and overall performance for parallel matrix multiplication.

III. Results and Discussion

A. Discussion of Results

Tables I, II, and III present the execution times of sequential and parallel matrix multiplication for different matrix sizes (N) and values of K. It is evident from Table I that all three MPI-based parallel implementations significantly outperform the sequential version. For instance, for $N = 10000$ and $K = 32$, the sequential execution time is approximately 2.7 seconds, whereas the parallel execution times range from 0.678 seconds for non-blocking P2P to 0.844 seconds for blocking P2P.

TABLE I
COMPARISON OF SEQUENTIAL AND PARALLEL EXECUTION TIMES FOR MATRIX MULTIPLICATION WITH N = 10000 AND K = 32

Method	Sequential Time (s)	Parallel Time (s)
Blocking P2P	2.67793	0.844758
Collective Communication	3.073263	0.879117
Non-blocking P2P	2.705896	0.677149

When comparing the parallel methods, the blocking point-to-point (P2P) approach provides good performance but is slightly slower than the non-blocking P2P in most cases. This is because processes must wait for each message to complete before continuing computation, introducing idle time. Collective communication, while simplifying code and being optimized in MPI libraries, shows running times comparable to blocking P2P, sometimes slightly slower due to library overhead. The non-blocking P2P method consistently achieves the fastest parallel times, as it allows computation and communication to overlap, reducing idle time and improving CPU utilization. These observations can be confirmed from Tables I and III, where non-blocking P2P consistently has the lowest parallel execution times across different N values.

TABLE II
COMPARISON OF SEQUENTIAL AND PARALLEL EXECUTION TIMES FOR MATRIX MULTIPLICATION WITH N = 10000 AND K = 64

Method	Sequential Time (s)	Parallel Time (s)
Blocking P2P	6.661408	1.827455
Collective Communication	6.548881	1.633688
Non-blocking P2P	6.586764	1.640891

The effect of matrix size is also clear from Table III: as N increases, execution times grow for both sequential and parallel computations. However, parallelization mitigates this increase effectively. This behavior aligns with the theoretical time complexity of $O(N^2)$ for matrix multiplication with fixed K, where doubling N approximately quadruples the sequential computation time. Increasing K from 32 to 64, as shown in Table II, increases computation for both sequential and parallel implementations, but the parallel methods scale well and maintain significant speedup.

TABLE III
PRACTICAL RUNNING TIMES (SECONDS) OF MATRIX MULTIPLICATION METHODS FOR DIFFERENT VALUES OF N AND K = 32

Matrix Size N	Serial	MPI Blocking P2P	MPI Collective	MPI Non-blocking P2P
1000	0.025	0.0002	0.009	0.0065
5000	0.612	0.194	0.218:	0.153
10000	2.698	0.839	0.881	0.678
20000	11.84	3.587	3.422	2.937

In conclusion, the results indicate that MPI-based parallelization substantially reduces computation time for matrix multiplication. Among the tested methods, non-blocking P2P communication demonstrates the best performance for small to medium-scale parallelism. Collective communication offers simplified code and can provide benefits for larger process counts, while blocking P2P remains straightforward and reliable.

APPENDIX A

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <string.h>

#define N 1000
#define K 32

void Multiply_serial(float *A, float *B, float *C, int m, int n, int p) {
    for (int i = 0; i < m; i++)
        for (int j = 0; j < p; j++) {
            C[i*p + j] = 0.0f;
            for (int k = 0; k < n; k++)
                C[i*p + j] += A[i*n + k] * B[k*p + j];
        }
}

int IsEqual(float *A, float *B, int m, int n) {
    for (int i = 0; i < m*n; i++)
        if (A[i] != B[i])
            return 0;
    return 1;
}
```

APPENDIX B

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

```
    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    float *A = NULL, *B = NULL, *C = NULL, *C_serial = NULL;

    int rows_per_proc = N / size;
    int remainder = N % size;

    // Number of rows handled by this process
    int local_rows = rows_per_proc + (rank < remainder ? 1 : 0);
    int start_row = rank * rows_per_proc + (rank < remainder ? rank : remainder);

    // Allocate memory
    float *A_local = (float*)malloc(local_rows * K * sizeof(float));
    float *C_local = (float*)malloc(local_rows * N * sizeof(float));
    B = (float*)malloc(K * N * sizeof(float));

    if (rank == 0) {
        A = (float*)malloc(N * K * sizeof(float));
        C = (float*)malloc(N * N * sizeof(float));
        C_serial = (float*)malloc(N * N * sizeof(float));

        srand(time(NULL));
        for (int i = 0; i < N*K; i++) A[i] = (float)rand() / RAND_MAX;
        for (int i = 0; i < K*N; i++) B[i] = (float)rand() / RAND_MAX;

        double t1 = MPI_Wtime();
        Multiply_serial(A, B, C_serial, N, K, N);
        double t2 = MPI_Wtime();
        printf("Sequential multiplication time: %f seconds\n", t2 - t1);
    }

    // Broadcast B
    MPI_Bcast(B, K*N, MPI_FLOAT, 0, MPI_COMM_WORLD);

    // Synchronize before parallel timing starts
    MPI_Barrier(MPI_COMM_WORLD);
    double t_parallel_start = MPI_Wtime();

    // Distribute rows of A
    if (rank == 0) {
        for (int i = 1; i < size; i++) {
            int send_rows = rows_per_proc + (i < remainder ? 1 : 0);
            int offset = i*rows_per_proc + (i < remainder ? i : remainder);
            MPI_Send(&A[offset*K], send_rows*K, MPI_FLOAT, i, 0, MPI_COMM_WORLD);
        }
        for (int i = 0; i < local_rows*K; i++) A_local[i] = A[i];
    } else {
        MPI_Recv(A_local, local_rows*K, MPI_FLOAT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    }

    // Local computation
    Multiply_serial(A_local, B, C_local, local_rows, K, N);

    // Gather results
    if (rank == 0) {
        for (int i = 0; i < local_rows*N; i++) C[i] = C_local[i];
        for (int i = 1; i < size; i++) {
            int recv_rows = rows_per_proc + (i < remainder ? 1 : 0);
            int offset = i*rows_per_proc + (i < remainder ? i : remainder);
            MPI_Recv(&C[offset*N], recv_rows*N, MPI_FLOAT, i, 1, MPI_COMM_WORLD,
```

```
                MPI_STATUS_IGNORE);
        }
    } else {
        MPI_Send(C_local, local_rows*N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD);
    }

    // End parallel timing
    MPI_Barrier(MPI_COMM_WORLD);
    double t_parallel_end = MPI_Wtime();
    if (rank == 0)
        printf("Parallel multiplication time (Blocking P2P): %f seconds\n",
            t_parallel_end - t_parallel_start);

    // Check correctness
    if (rank == 0) {
        if (IsEqual(C, C_serial, N, N))
            printf("Parallel result matches serial computation!\n");
        else
            printf("Parallel result does NOT match serial computation!\n");

        free(A);
        free(C);
        free(C_serial);
    }

    free(A_local);
    free(B);
    free(C_local);

    MPI_Finalize();
    return 0;
}
```

APPENDIX C

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

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    float *A = NULL, *B = NULL, *C = NULL, *C_serial = NULL;

    int rows_per_proc = N / size;
    int remainder = N % size;

    int *sendcounts = malloc(size * sizeof(int));
    int *displs = malloc(size * sizeof(int));
    int *recvcounts = malloc(size * sizeof(int));
    int *recvdispls = malloc(size * sizeof(int));

    int offset = 0;
    for(int i = 0; i < size; i++){
        sendcounts[i] = (rows_per_proc + (i < remainder ? 1 : 0)) * K;
        displs[i] = offset * K;
        recvcounts[i] = (rows_per_proc + (i < remainder ? 1 : 0)) * N;
        recvdispls[i] = offset * N;
        offset += rows_per_proc + (i < remainder ? 1 : 0);
    }

    int local_rows = sendcounts[rank] / K;

    float *A_local = malloc(local_rows * K * sizeof(float));
```

```
    float *C_local = malloc(local_rows * N * sizeof(float));
    B = malloc(K * N * sizeof(float));

    if(rank == 0){
        A = malloc(N * K * sizeof(float));
        C = malloc(N * N * sizeof(float));
        C_serial = malloc(N * N * sizeof(float));

        srand(time(NULL));
        for(int i=0;i<N*K;i++) A[i]=(float)rand()/RAND_MAX;
        for(int i=0;i<K*N;i++) B[i]=(float)rand()/RAND_MAX;

        double t1 = MPI_Wtime();
        Multiply_serial(A,B,C_serial,N,K,N);
        double t2 = MPI_Wtime();
        printf("Sequential multiplication time: %f sec\n", t2-t1);
    }

    MPI_Bcast(B, K*N, MPI_FLOAT, 0, MPI_COMM_WORLD);

    MPI_Scatterv(A, sendcounts, displs, MPI_FLOAT,
                 A_local, sendcounts[rank], MPI_FLOAT, 0, MPI_COMM_WORLD);

    double t3 = MPI_Wtime();
    Multiply_serial(A_local, B, C_local, local_rows, K, N);
    double t4 = MPI_Wtime();

    MPI_Gatherv(C_local, recvcounts[rank], MPI_FLOAT,
                C, recvcounts, recvdispls, MPI_FLOAT, 0, MPI_COMM_WORLD);

    if(rank == 0){
        printf("Parallel (Collective) multiplication time: %f sec\n", t4 - t3);

        if(IsEqual(C, C_serial, N,N))
            printf("Result matches!\n");
        else
            printf("Result mismatch!\n");

        free(A); free(C); free(C_serial);
    }

    free(A_local); free(B); free(C_local);
    free(sendcounts); free(displs); free(recvcounts); free(recvdispls);

    MPI_Finalize();
    return 0;
}
```

APPENDIX D

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

    int rank, size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    float *A = NULL, *B = NULL, *C = NULL, *C_serial = NULL;

    int rows_per_proc = N / size;
    int remainder = N % size;

    int local_rows = rows_per_proc + (rank < remainder ? 1 : 0);
    int start_row = rank * rows_per_proc + (rank < remainder ? rank : remainder);
```

```
15
16      float *A_local = (float*) malloc(local_rows * K * sizeof(float));
17      float *C_local = (float*) malloc(local_rows * N * sizeof(float));
18      B = (float*) malloc(K * N * sizeof(float));
19
20      if(rank == 0){
21          A = (float*) malloc(N * K * sizeof(float));
22          C = (float*) malloc(N * N * sizeof(float));
23          C_serial = (float*) malloc(N * N * sizeof(float));
24
25          srand(time(NULL));
26          for(int i=0;i<N*K;i++) A[i]=(float)rand()/RAND_MAX;
27          for(int i=0;i<K*N;i++) B[i]=(float)rand()/RAND_MAX;
28
29          double t1 = MPI_Wtime();
30          Multiply_serial(A, B, C_serial, N, K, N);
31          double t2 = MPI_Wtime();
32          printf("Sequential multiplication time: %f seconds\n", t2 - t1);
33      }
34
35      MPI_Bcast(B, K*N, MPI_FLOAT, 0, MPI_COMM_WORLD);
36
37      MPI_Request req_send_A[1], req_recv_A[1];
38
39      if(rank == 0){
40          for(int i = 1; i < size; i++){
41              int send_rows = rows_per_proc + (i < remainder ? 1 : 0);
42              int offset = i*rows_per_proc + (i < remainder ? i : remainder);
43              MPI_Isend(&A[offset*K], send_rows*K, MPI_FLOAT, i, 0, MPI_COMM_WORLD, &
                    req_send_A[0]);
44          }
45          memcpy(A_local, A, local_rows*K*sizeof(float));
46      } else {
47          MPI_Irecv(A_local, local_rows*K, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &req_recv_A
                [0]);
48          MPI_Wait(&req_recv_A[0], MPI_STATUS_IGNORE);
49      }
50
51      double t3 = MPI_Wtime();
52      Multiply_serial(A_local, B, C_local, local_rows, K, N);
53      double t4 = MPI_Wtime();
54
55      MPI_Request req_send_C[1], req_recv_C[1];
56
57      if(rank == 0){
58          memcpy(&C[start_row*N], C_local, local_rows*N*sizeof(float));
59
60          for(int i=1;i<size;i++){
61              int recv_rows = rows_per_proc + (i < remainder ? 1 : 0);
62              int offset = i*rows_per_proc + (i < remainder ? i : remainder);
63              MPI_Irecv(&C[offset*N], recv_rows*N, MPI_FLOAT, i, 1, MPI_COMM_WORLD, &
                    req_recv_C[0]);
64              MPI_Wait(&req_recv_C[0], MPI_STATUS_IGNORE);
65          }
66      } else {
67          MPI_Isend(C_local, local_rows*N, MPI_FLOAT, 0, 1, MPI_COMM_WORLD, &req_send_C
                [0]);
68          MPI_Wait(&req_send_C[0], MPI_STATUS_IGNORE);
69      }
70
71      if(rank == 0){
72          printf("Parallel (Non-Blocking P2P) time: %f seconds\n", t4 - t3);
73          if(IsEqual(C, C_serial, N, N))
74              printf("Parallel result MATCHES serial result.\n");
```

```
75          else
76              printf("Parallel result does NOT match serial result.\n");
77
78          free(A); free(C); free(C_serial);
79      }
80
81      free(A_local); free(B); free(C_local);
82      MPI_Finalize();
83      return 0;
84  }
```