#Midterm Assignment -Parallel Computing

Name-Surname : Onur Çetin

ID : 20200808050

1-Purpose of Assignment and My Solution Steps

- The objective of this assignment is to perform parallel matrix multiplication using the Microsoft MPI (MSMPI) library for large matrices loaded from external binary files

- To create binary files containing two (A.bin and B.bin) matrices of size 5000×5000 and to make the multiplication of these matrices more efficient and less time consuming with parallel methods and optimizations.

- To generate these two matrix files, I used a C file called random_matrix.c. The main purpose of this code is to generate 2 5000×5000 matrices for me.

- Then I integrated these binary files into my code and used them in my loadMatrixFromBinary function and performed the operation in my matrixMultiplication function. Then I used them in my printMatrix function to see the results on my terminal.

- For time measurement, I used the MPI_Wtime property specified in the assignment to display the time output on the terminal.

2-The Techniques I used in My Solution

☐ Data Decomposition

```
int rows_per_process = MATRIX_SIZE / size;
int local_rows = (rank == size - 1) ? (MATRIX_SIZE - (size - 1) * rows_per_process) : rows_per_process;
```

- `int rows_per_process = MATRIX_SIZE / size;` is used to determine the number of rows per processor.
- The expression `int local_rows = (rank == size - 1) ? (MATRIX_SIZE - (size - 1) * rows_per_process) : rows_per_process;` is used to determine the number of rows each processor will process locally. This way, the workload is balanced between processors for matrix multiplication.

☐ Communication Minimization

```
MPI_Bcast(A, MATRIX_SIZE * MATRIX_SIZE, MPI_DOUBLE, MASTER_RANK, MPI_COMM_WORLD);
MPI_Bcast(B, MATRIX_SIZE * MATRIX_SIZE, MPI_DOUBLE, MASTER_RANK, MPI_COMM_WORLD);
```

- `MPI_Bcast(A, MATRIX_SIZE * MATRIX_SIZE, MPI_DOUBLE, MASTER_RANK, MPI_COMM_WORLD);` and `MPI_Bcast(B, MATRIX_SIZE * MATRIX_SIZE, MPI_DOUBLE, MASTER_RANK, MPI_COMM_WORLD);` are used to send matrices to all processors in a single broadcast. This reduces communication costs.

☐ Local Storage Usage

```
double* local_C = (double*)malloc(rows_per_process * MATRIX_SIZE * sizeof(double));
  if (local_C == NULL) {
    fprintf(stderr, "Memory allocation failed.\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
}
```

- `double* local_C = (double*)malloc(local_rows * MATRIX_SIZE * sizeof(double));` is used to describe the memory space allocated locally to each processor. This allows each processor to keep only the part of memory that it needs to process, thus speeding up data access.

☐ Loop Restructuring

```
void matrixMultiplication(double* A, double* B, double* C, int rows, int cols, int local_rows) {
    for (int i = 0; i < local_rows; i++) {
        for (int j = 0; j < cols; j++) {
            double temp = 0.0; // Temporary variable to store the sum
            for (int k = 0; k < cols; k++) {
                temp += A[i * cols + k] * B[k * cols + j];
            }
            C[i * cols + j] = temp; // Assign the sum to the result matrix
        }
    }
}
```

- In this example, loop restructuring is achieved in the matrixMultiplication function. It also performs a vectorization-like optimization using a temporary variable. This can help you make more efficient use of the processor cache.

☐ Optimization Flag

```
onuce@DESKTOP-7S6BRTU:~/midtermParallel$ mpirun --oversubscribe -np 20 ./midtermHomework
```

- This flag allows to run with more processors than the number of processors available on the same computer node. Usually, the use of this flag can ensure optimal utilization of computer resources.

3-Code Parts

Code Dependencies

```
sudo apt install openmpi-bin
sudo apt install libopenmpi-dev
mpicc midtermHomework.c -o midtermHomework
mpirun ./midtermHomework
```

→Files I use

```
onuce@DESKTOP-7S6BRTU:~/midtermParallel$ ls -l
total 390680
-rw-r--r-- 1 onuce onuce 200000000 May  9 18:08 A.bin
-rw-r--r-- 1 onuce onuce 200000000 May  9 18:08 B.bin
-rwxr-xr-x 1 onuce onuce     16888 May  9 18:11 midtermHomework
-rw-r--r-- 1 onuce onuce      3043 May  9 18:04 midtermHomework.c
-rwxr-xr-x 1 onuce onuce     16448 May  9 18:08 random_matrix
-rw-r--r-- 1 onuce onuce       944 May  9 18:07 random_matrix.c
```

→random_matrix.c File

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char **argv) {
 // Get arguments
    if (argc != 4) {
        fprintf(stderr, "usage: %s nrows ncols filename\n", argv
        exit(EXIT_FAILURE);
    }
```

```
    long long nrows = atoll(argv[1]);
    long long ncols = atoll(argv[2]);
    const char *path = argv[3];
 // Open file for writing
    FILE *file = fopen(path, "w");
    if (file == NULL) {
        perror("fopen failed");
        exit(EXIT_FAILURE);
    }
 // Set seed for random number generator
    srand48(time(0));
 // Write random numbers
    long long length = nrows * ncols;
    for (long long i = 0; i < length; i++) {
        double random_number = drand48();
        if (fwrite(&random_number, sizeof (double), 1, file) !=
            fprintf(stderr, "write failed\n");
            exit(EXIT_FAILURE);
        }
    }
 // Close file
    fclose(file);
}
```

→**First version : midtermHomework.c (Without any optimization)**

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

#define MATRIX_SIZE 5000
#define MASTER_RANK 0
#define FILENAME_A "A.bin"
```

```
#define FILENAME_B "B.bin"

void loadMatrixFromBinary(const char* filename, double* matrix,
    FILE* file = fopen(filename, "rb");
    if (file == NULL) {
        fprintf(stderr, "Error opening file %s.\n", filename);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    size_t elements_read = fread(matrix, sizeof(double), rows *
    if (elements_read != rows * cols) {
        fprintf(stderr, "Error reading file %s. Expected %d eler
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    fclose(file);
}

void matrixMultiplication(double* A, double* B, double* C, int r
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            C[i * cols + j] = 0.0;
            for (int k = 0; k < cols; k++) {
                C[i * cols + j] += A[i * cols + k] * B[k * cols
            }
        }
    }
}

void printMatrix(double* matrix, int rows, int cols) {
     for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%.2f ", matrix[i * cols + j]);
        }
        printf("\n");
    }
```

```
}

int main(int argc, char** argv) {
    int size, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (size < 2) {
        fprintf(stderr, "Program requires at least 2 MPI process
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    int rows_per_process = MATRIX_SIZE / size;
    double* A = (double*)malloc(MATRIX_SIZE * MATRIX_SIZE * size
    double* B = (double*)malloc(MATRIX_SIZE * MATRIX_SIZE * size
    double* C = (double*)malloc(MATRIX_SIZE * MATRIX_SIZE * size

    if (A == NULL || B == NULL || C == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    loadMatrixFromBinary(FILENAME_A, A, MATRIX_SIZE, MATRIX_SIZE
    loadMatrixFromBinary(FILENAME_B, B, MATRIX_SIZE, MATRIX_SIZE

    MPI_Bcast(A, MATRIX_SIZE * MATRIX_SIZE, MPI_DOUBLE, MASTER_R
    MPI_Bcast(B, MATRIX_SIZE * MATRIX_SIZE, MPI_DOUBLE, MASTER_R

    double* local_C = (double*)malloc(rows_per_process * MATRIX_
      if (local_C == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    matrixMultiplication(A + rank * rows_per_process * MATRIX_SI
```

```
    MPI_Gather(local_C, rows_per_process * MATRIX_SIZE, MPI_DOUB
        printf("Result matrix C:\n");
        printMatrix(C, MATRIX_SIZE, MATRIX_SIZE);
    }

    free(A);
    free(B);
    free(C);
    free(local_C);
    MPI_Finalize();

    return 0;
}
```

→**Then After Optimization**

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

#define MATRIX_SIZE 5000
#define MASTER_RANK 0
#define FILENAME_A "A.bin"
#define FILENAME_B "B.bin"

void loadMatrixFromBinary(const char *filename, double *matrix,
{
    FILE *file = fopen(filename, "rb");
    if (file == NULL)
    {
        fprintf(stderr, "Error opening file %s.\n", filename);
```

```
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    size_t elements_read = fread(matrix, sizeof(double), rows *
    if (elements_read != rows * cols)
    {
        fprintf(stderr, "Error reading file %s. Expected %d elem
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    fclose(file);
}

void matrixMultiplication(double *A, double *B, double *C, int r
{
    for (int i = 0; i < local_rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            double temp = 0.0; // Temporary variable to store th
            for (int k = 0; k < cols; k++)
            {
                temp += A[i * cols + k] * B[k * cols + j];
            }
            C[i * cols + j] = temp; // Assign the sum to the res
        }
    }
}

void printMatrix(double *matrix, int rows, int cols)
{
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            printf("%.2f ", matrix[i * cols + j]);
```

```
        }
        printf("\n");
    }
}

int main(int argc, char **argv)
{
    int size, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (size < 2)
    {
        fprintf(stderr, "Program requires at least 2 MPI process
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    double start_time, end_time; // Variables to hold start and

    start_time = MPI_Wtime(); // Start measuring time

    // Load matrices A and B from binary files
    double *A = (double *)malloc(MATRIX_SIZE * MATRIX_SIZE * siz
    double *B = (double *)malloc(MATRIX_SIZE * MATRIX_SIZE * siz
    double *C = (double *)malloc(MATRIX_SIZE * MATRIX_SIZE * siz

    if (A == NULL || B == NULL || C == NULL)
    {
        fprintf(stderr, "Memory allocation failed.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    loadMatrixFromBinary(FILENAME_A, A, MATRIX_SIZE, MATRIX_SIZE
    loadMatrixFromBinary(FILENAME_B, B, MATRIX_SIZE, MATRIX_SIZE
```

```
    MPI_Bcast(A, MATRIX_SIZE * MATRIX_SIZE, MPI_DOUBLE, MASTER_R
    MPI_Bcast(B, MATRIX_SIZE * MATRIX_SIZE, MPI_DOUBLE, MASTER_R

    int rows_per_process = MATRIX_SIZE / size;
    int local_rows = (rank == size - 1) ? (MATRIX_SIZE - (size -

    double *local_C = (double *)malloc(local_rows * MATRIX_SIZE
    if (local_C == NULL)
    {
        fprintf(stderr, "Memory allocation failed.\n");
        MPI_Abort(MPI_COMM_WORLD, 1);
    }

    matrixMultiplication(A + rank * rows_per_process * MATRIX_SI

    MPI_Gather(local_C, local_rows * MATRIX_SIZE, MPI_DOUBLE, C,

    end_time = MPI_Wtime(); // Stop measuring time

    if (rank == MASTER_RANK)
    {
        printf("Result matrix C:\n");
        printMatrix(C, MATRIX_SIZE, MATRIX_SIZE);
        printf("Time taken for matrix multiplication: %f seconds
    }

    free(A);
    free(B);
    free(C);
    free(local_C);

    MPI_Finalize();

    return 0;
}
```

4-Time Comparison

→Some outputs

```
1244.90 1243.24 1261.52 1246.99 1251.53 1252.19 1261.61 1265.85 1228.99 1253.09 1247.89 1247.62 1235.40 1225.12 1231.32 1236.69 1240.65 1248.39
1230.14 1249.03 1242.09 1230.85 1248.78 1242.79 1227.45 1252.00 1249.37 1247.89 1233.74 1243.86 1255.96 1253.93 1247.97 1253.59 1242.89 1243.58
1231.75 1238.22 1237.81 1267.93 1239.21 1232.46 1235.06 1248.88 1233.67 1237.46 1250.05 1244.18 1277.03 1225.11 1231.83 1243.27 1228.57 1241.31
1241.33 1230.16 1258.26 1240.72 1260.59 1236.33 1254.56 1255.20 1222.24 1217.31 1231.45 1261.23 1256.99 1236.24 1237.70 1259.10 1234.11 1231.43
1235.96 1245.12 1279.53 1239.26 1256.37 1225.15 1245.73 1226.52 1260.83 1245.95 1260.87 1222.20 1250.74 1268.42 1255.68 1238.29 1236.95 1253.55
1231.14 1254.75 1221.37 1250.65 1229.70 1259.71 1223.72 1224.68 1233.49 1269.65 1239.28 1225.29 1235.97 1243.15 1251.45 1251.12 1246.56 1231.90
1226.00 1255.46 1241.45 1253.14 1252.92 1247.60 1266.26 1254.84 1239.81 1266.84 1264.23 1270.81 1248.76 1238.97 1234.66 1245.46 1228.26 1236.88
1243.01 1261.65 1247.68 1231.16 1253.90 1257.42 1241.34 1246.16 1235.23 1271.11 1246.36 1249.23 1232.79 1253.30 1257.99 1257.02 1251.37 1252.86
1248.58 1242.23 1224.23 1234.15 1242.61 1230.05 1216.91 1249.61 1244.74 1247.95 1241.23 1233.41 1241.04 1232.06 1250.22 1246.65 1233.39 1232.15
1251.18 1261.10 1248.13 1249.09 1244.92 1224.73 1236.44 1217.58 1235.08 1244.02 1257.39 1233.95 1249.46 1245.02 1257.61 1235.79 1235.35 1231.63
1238.50 1247.22 1264.70 1230.35 1227.35 1248.06 1240.54 1254.41 1237.86 1249.77 1250.28 1259.98 1240.81 1246.42 1240.30 1238.30 1249.14 1247.75
1244.63 1237.02 1237.79 1256.28 1242.96 1251.52 1238.61 1217.80 1241.84 1244.19 1266.42 1232.04 1240.74 1251.50 1246.99 1264.14 1245.83 1226.85
1228.58 1234.31 1241.79 1256.72 1248.06 1237.85 1240.42 1251.13 1238.80 1253.50 1260.57 1259.06 1232.75 1234.86 1215.42 1242.78 1235.01 1248.34
1232.78 1253.81 1237.52 1255.39 1240.01 1242.10 1236.21 1259.58 1247.97 1247.72 1241.66 1239.30 1240.01 1252.86 1235.76 1246.95 1244.88 1228.43
1254.27 1244.61 1231.16 1240.73 1253.44 1242.65 1243.04 1223.62 1228.17 1239.07 1243.89 1245.93 1251.96 1236.46 1266.80 1256.00 1232.00 1267.44
1236.66 1244.43 1238.11 1259.37 1225.45 1241.03 1236.43 1265.30 1237.74 1252.62 1238.87 1251.87 1233.77 1253.08 1239.98 1239.97 1231.76 1244.35
1257.68 1232.41 1217.78 1238.80 1247.58 1237.25 1263.81 1247.46 1249.70 1231.52 1263.66 1237.75 1221.93 1256.89 1276.40 1243.93 1237.53 1243.53
1261.89 1241.56 1228.57 1248.12 1243.71 1243.25 1235.38 1236.65 1250.43 1233.92 1243.48 1251.11 1241.13 1229.67 1249.52 1279.16 1254.55 1240.26
```

- **Without any optimization**

```
Time taken for matrix multiplication: 612.377312 seconds
onuce@DESKTOP-7S6BRTU:~/midtermParallel$
```

- **After optimization but without flags**

→ `mpirun ./midtermHomework`

```
Time taken for matrix multiplication: 530.639449 seconds
```

- **After optimization and run with flags**

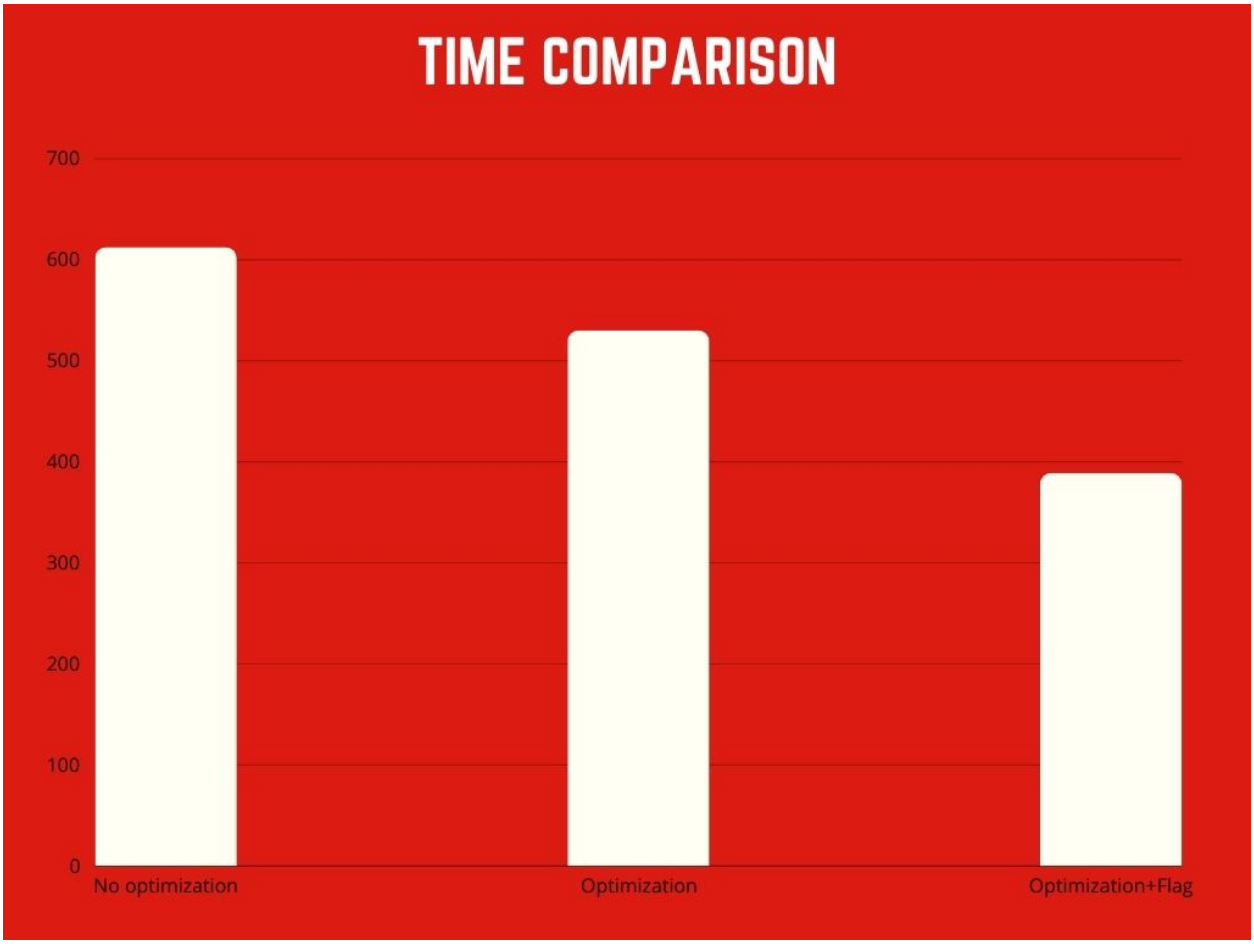
→ `mpirun --oversubscribe -np 20 ./midtermHomework`

```
Time taken for matrix multiplication: 389.147821
onuce@DESKTOP-7S6BRTU:~/midtermParallel$
```

No Optimization	Optimization	Optimization+Flag
612.4 seconds	530.6 seconds	389.1 seconds

5-Data Visualization

- As can be seen in the graph, each optimization we make creates a noticeable decrease and we perform a more efficient process with time savings.



6-Conclusion

- The use of the --oversubscribe flag may give different results depending on the hardware and software environment used. In some cases it may not be appropriate or may cause operating system dependent problems. Therefore, it is important to check the system documentation and recommendations before using this flag.(Because when I first tried it as -np 40, I realized that it was not suitable and ran it as -np 20.)

- Small touches such as Data Decomposition, Communication Minimization, Local Storage Usage, Loop Restructuring, which are among the parallelization algorithms we use, provide good time savings and increase efficiency even in 5000×5000 matrix size. When the matrix size increases, this efficiency difference becomes much more noticeable.

- On top of these optimizations, if we use our flags, we can notice that the processing time is even shorter.