Lab Session Week 11

|  |  |  |
| --- | --- | --- |
| **Student ID** | **Student Name** | **Student Email** |
| 32844700 | Teh Jia Xuan | Jteh0015@student.monash.edu |

**Task 1 Code**

//////////////////////////////////////////////////////////////////////////////////////

// MatrixMul\_1D\_bin.c

// ----------------------------------------------------------------------------------

//

// Multiplies two matrices and writes the resultant multiplication into a binary file.

//

//////////////////////////////////////////////////////////////////////////////////////

#include <stdio.h>

#include <stdlib.h>

#include <memory.h>

#include <time.h>

#include <pthread.h>

#define NUM\_THREADS 8

int \*pMatrixA, \*pMatrixB;

unsigned long long \*pMatrixC;

int rowA, rowB, rowC, colA, colB, colC, commonPoint;

void \*ThreadFunc2(void \*pArg){

    // Assume the matrix arrays, row, column and commonPoint variables are global variables and initialised in main function.

    int i, j, k;

    int my\_rank = \*((int\*)pArg);

    //calculate the tile partition

    //using result will divide into half even number process left odd number process right

    int row\_start\_point, row\_end\_point;

    int col\_start\_point, col\_end\_point;

    int threadColDiv = 1;

    int threadColDivRemain = 0;

    int threadRowDiv = 1;

    int threadRowDivRemain = 0;

    threadColDiv = colC / 2;

    threadColDivRemain = colC % 2;

    threadRowDiv = rowC / (NUM\_THREADS / 2);

    threadRowDivRemain = rowC % (NUM\_THREADS / 2);

    if(my\_rank % 2 == 0){

        // Even thread

        if(my\_rank == (NUM\_THREADS - 2)){

            // Last even thread

            row\_start\_point = (my\_rank / 2) \* threadRowDiv;

            row\_end\_point = row\_start\_point + threadRowDiv + threadRowDivRemain;

        }else{

            // Not last even thread

            row\_start\_point = (my\_rank / 2) \* threadRowDiv;

            row\_end\_point = row\_start\_point + threadRowDiv;

        }

        col\_start\_point = 0;

        col\_end\_point = threadColDiv;

    }else{

        // Odd thread

        if(my\_rank == (NUM\_THREADS - 1)){

            // Last odd thread

            row\_start\_point = (my\_rank / 2) \* threadRowDiv;

            row\_end\_point = row\_start\_point + threadRowDiv + threadRowDivRemain;

        }else{

            // Not last odd thread

            row\_start\_point = (my\_rank / 2) \* threadRowDiv;

            row\_end\_point = row\_start\_point + threadRowDiv;

        }

        col\_start\_point = threadColDiv;

        col\_end\_point = col\_start\_point + threadColDiv + threadColDivRemain;

    }

    // Matrix multiplication

    for(i = row\_start\_point; i < row\_end\_point; i++){

        for(j = col\_start\_point; j < col\_end\_point; j++){

            for(k = 0; k < commonPoint; k++){

                pMatrixC[(i\*colC)+j] += (pMatrixA[(i\*colA)+k] \* pMatrixB[(k\*colB)+j]);

            }

        }

    }

    return NULL;

}

int main()

{

    // Variables

    int i = 0, j = 0, k = 0;

    /\* Clock information \*/

    struct timespec start, end;

    double time\_taken;

    pthread\_mutex\_t mutex;

    pthread\_mutex\_init(&mutex, NULL);

    pthread\_t threads[NUM\_THREADS];

    int thread\_ids[NUM\_THREADS];

    clock\_gettime(CLOCK\_MONOTONIC, &start);

    // 1. Read Matrix A

    rowA = 0;

    colA = 0;

    printf("Matrix Multiplication using 1-Dimension Arrays - Start\n\n");

    printf("Reading Matrix A - Start\n");

    FILE \*pFileA = fopen("MA\_500x500.bin", "rb");

    fread(&rowA, sizeof(int), 1, pFileA);

    fread(&colA, sizeof(int), 1, pFileA);

    pMatrixA = (int\*)malloc((rowA\*colA) \* sizeof(int));

    for(i = 0; i < rowA; i++){

        fread(&pMatrixA[i\*colA], sizeof(int), colA, pFileA);

    }

    fclose(pFileA);

    printf("Reading Matrix A - Done\n");

    // 2. Read Matrix B

    rowB = 0;

    colB = 0;

    printf("Reading Matrix B - Start\n");

    FILE \*pFileB = fopen("MB\_500x500.bin", "rb");

    fread(&rowB, sizeof(int), 1, pFileB);

    fread(&colB, sizeof(int), 1, pFileB);

    pMatrixB = (int\*)malloc((rowB\*colB) \* sizeof(int));

    for(i = 0; i < rowB; i++){

        fread(&pMatrixB[i\*colB], sizeof(int), colB, pFileB);

    }

    fclose(pFileB);

    printf("Reading Matrix B - Done\n");

    // 3. Perform matrix multiplication

    printf("Matrix Multiplication - Start\n");

    rowC = rowA;

    colC = colB;

    pMatrixC = (unsigned long long\*)calloc((rowC\*colC), sizeof(unsigned long long));

    commonPoint = colC;

    int rows\_per\_thread = rowC/NUM\_THREADS;

    int cols\_per\_thread = colC/NUM\_THREADS;

    for (i = 0; i < NUM\_THREADS; i++){

        //creating threads to compute

        thread\_ids[i] = i;

        pthread\_create(&threads[i], NULL, ThreadFunc2, &thread\_ids[i]);

    }

    for (i = 0; i< NUM\_THREADS; i++){

        //join them tgt after computing

        pthread\_join(threads[i], NULL);

    }

    printf("Matrix Multiplication - Done\n");

    // 4. Write resuls to a new file

    printf("Write Resultant Matrix C to File - Start\n");

    FILE \*pFileC = fopen("MC\_500x500.bin", "wb");

    fwrite(&rowC, sizeof(int), 1, pFileC);

    fwrite(&colC, sizeof(int), 1, pFileC);

    for(i = 0; i < rowC; i++){

        fwrite(&pMatrixC[i\*colC], sizeof(unsigned long long), colC, pFileC); // use fwrite to write one row of columns to the file

    }

    fclose(pFileC);

    printf("Write Resultant Matrix C to File - Done\n");

    //time taken for them to complete overall program

    clock\_gettime(CLOCK\_MONOTONIC, &end);

    time\_taken = (end.tv\_sec - start.tv\_sec) \* 1e9;

    time\_taken = (time\_taken + (end.tv\_nsec - start.tv\_nsec)) \* 1e-9;

    printf("Overall time (Including read, multiplication and write)(s): %lf\n", time\_taken);    // ts

    // Clean up

    free(pMatrixA);

    free(pMatrixB);

    free(pMatrixC);

    printf("Matrix Multiplication using 1-Dimension Arrays - Done\n");

    return 0;

}

**Task 1**

|  |  |
| --- | --- |
| **Overall Time** | 0.873422 |
| **Computational Time (parallelizable)** | 0.779986 |
| **Read write time (serial)** | 0.093436 |
| **Parallelizable code** | 0.779986 / 0.873422 = 0.893 |
| **Serial** | 0.107 |
| **Theoretical Speed Up** | 1/ 0.107 + (0.893/8) = 4.573 |

A screen shot of a computer program

Description automatically generated

**Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Number of CPU cores or logical processes: 8**  **Number of threads used for POSIX/OMP: 8** | | | | | |
| **Matrix size** | **500x500** | **1000x1000** | **2000x2000** | **3000x3000** | **4000x4000** |
| **Serial time (s)** | 1.036289 | 7.205710 | 52.871768 | 180.674973 | 424.491226 |
| **Parallel time POSIX/OMP** | 0.329884 | 2.313118 | 12.446038 | 36.649904 | 83.631660 |
| **Speed up (Ts/Tp)** | 3.141374 | 3.11515 | 4.24808 | 4.929753 | 5.075724 |

**Analysis**

The theoretical speed up is 4.574 which is calculated by the amdahl’s law

Speed up = 1/rs + rp/n

Rs = 0.093436/0.873422 = 0.107

Rp = 1 - 0.107 = 0.893

Speed up = 1/ 0.107 + 0.893/8 = 4.574

**500x500**

**Serial code**

A screen shot of a computer program

Description automatically generated

**Parallel code**

A black screen with white text

Description automatically generated

**1000x1000**

**Serial code**

A screen shot of a computer

Description automatically generated

**Parallel code**

A screen shot of a computer program

Description automatically generated

**2000x2000**

**Serial code**

A screen shot of a computer program

Description automatically generated

**Parallel code**

A screen shot of a computer

Description automatically generated

**3000x3000**

**Serial code**

A screen shot of a computer program

Description automatically generated

**Parallel code**

A black screen with white text

Description automatically generated

**4000x4000**

**Serial code**

A screen shot of a computer

Description automatically generated

**Parallel code**

A screen shot of a computer

Description automatically generated

**Check Correctness with 500 x 500 Matrix Mul**

**Serial code provided**

A close-up of a grid

Description automatically generated

**Parallel code with POSIX**

A grid of squares with black and white lines

Description automatically generated with medium confidence

**Task 2 – Code**

#include <stdio.h>

#include <stdlib.h>

#include <memory.h>

#include <time.h>

#include <mpi.h>

//combine local matrices to the global matrix

void combineLocalMatrices(unsigned long long \*localMatrix, unsigned long long \*globalMatrix, int row\_start, int row\_end, int col\_start, int col\_end, int colC) {

    //only loop through the necessary row and column for the local matrix

    //+= is because each of the elem in global matrix initially is 0, so x+0 = x

    for(int row = row\_start; row < row\_end; row++){

        for (int col = col\_start; col < col\_end; col++){

            // Add localMatrix to globalMatrix element by element

            globalMatrix[row \* colC + col] += localMatrix[row \* colC + col];

        }

    }

}

int main(int argc, char \*argv[])

{

    // Variables

    int i = 0, j = 0, k = 0;

    int my\_rank, size;

    int position;

    int pack\_size;

    int row\_start\_point, row\_end\_point;

    int col\_start\_point, col\_end\_point;

    unsigned long long \*pMatrixC;

    int rowA, rowB, rowC, colA, colB, colC, commonPoint;

    int \*pMatrixA = NULL, \*pMatrixB = NULL;

    /\* Clock information \*/

    struct timespec start, end, startComp, endComp;

    double time\_taken;

    clock\_gettime(CLOCK\_MONOTONIC, &start);

    //initialise mpi

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);

    // 1. Read Matrix A

    rowA = colA = rowB = colB = 0;

    if (my\_rank == 0) {

        printf("Matrix Multiplication using 1-Dimension Arrays - Start\n\n");

        // Reading Matrix A

        printf("Reading Matrix A - Start\n");

        FILE \*pFileA = fopen("MA\_500x500.bin", "rb");

        if (!pFileA) {

            perror("File A opening failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        fread(&rowA, sizeof(int), 1, pFileA);

        fread(&colA, sizeof(int), 1, pFileA);

        pMatrixA = (int \*)malloc((rowA \* colA) \* sizeof(int));

        if (pMatrixA == NULL) {

            perror("Memory allocation for Matrix A failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        for (i = 0; i < rowA; i++) {

            //reading row by row using fread

            fread(&pMatrixA[i \* colA], sizeof(int), colA, pFileA);

        }

        fclose(pFileA);

        printf("Reading Matrix A - Done\n");

        // Reading Matrix B

        printf("Reading Matrix B - Start\n");

        FILE \*pFileB = fopen("MB\_500x500.bin", "rb");

        if (!pFileB) {

            perror("File B opening failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        fread(&rowB, sizeof(int), 1, pFileB);

        fread(&colB, sizeof(int), 1, pFileB);

        pMatrixB = (int \*)malloc((rowB \* colB) \* sizeof(int));

        if (pMatrixB == NULL) {

            perror("Memory allocation for Matrix B failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        for (i = 0; i < rowB; i++) {

            //reading row by row

            fread(&pMatrixB[i \* colB], sizeof(int), colB, pFileB);

        }

        fclose(pFileB);

        printf("Reading Matrix B - Done\n");

        // Initialize Matrix C

        rowC = rowA;

        colC = colB;

        commonPoint = colA; // Initialize commonPoint

    }

    //broadcast rows to all the processses

    MPI\_Bcast(&rowA, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&colA, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&rowB, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&colB, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&rowC, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&colC, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&commonPoint, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); // Broadcast commonPoint

    //assign local matrix result

    pMatrixC = (unsigned long long \*)calloc((rowC \* colC), sizeof(unsigned long long));

    //calculate the tile partition

    //using result will divide into half even number process left odd number process right

    int threadColDiv = colC / 2;

    int threadColDivRemain = colC % 2;

    int threadRowDiv = rowC / (size / 2);

    int threadRowDivRemain = rowC % (size / 2);

    if (my\_rank == 0) {

        for (int temprank = 1; temprank < size; temprank++) {

            position = 0;

            if (temprank % 2 == 0) {

                // Even thread

                if (temprank == (size - 2)) {

                    // Last even thread

                    row\_start\_point = (temprank / 2) \* threadRowDiv;

                    row\_end\_point = row\_start\_point + threadRowDiv + threadRowDivRemain;

                } else {

                    // Not last even thread

                    row\_start\_point = (temprank / 2) \* threadRowDiv;

                    row\_end\_point = row\_start\_point + threadRowDiv;

                }

                col\_start\_point = 0;

                col\_end\_point = threadColDiv;

            } else {

                // Odd thread

                if (temprank == (size - 1)) {

                    // Last odd thread

                    row\_start\_point = (temprank / 2) \* threadRowDiv;

                    row\_end\_point = row\_start\_point + threadRowDiv + threadRowDivRemain;

                } else {

                    // Not last odd thread

                    row\_start\_point = (temprank / 2) \* threadRowDiv;

                    row\_end\_point = row\_start\_point + threadRowDiv;

                }

                col\_start\_point = threadColDiv;

                col\_end\_point = col\_start\_point + threadColDiv + threadColDivRemain;

            }

            //send to all processes

            //after getting their own row\_start\_point, row end  point, col start and col end

            MPI\_Send(&row\_start\_point, 1, MPI\_INT, temprank, 0, MPI\_COMM\_WORLD);

            MPI\_Send(&row\_end\_point, 1, MPI\_INT, temprank, 1, MPI\_COMM\_WORLD);

            MPI\_Send(&col\_start\_point, 1, MPI\_INT, temprank, 2, MPI\_COMM\_WORLD);

            MPI\_Send(&col\_end\_point, 1, MPI\_INT, temprank, 3, MPI\_COMM\_WORLD);

            //create subarray to store all the necessary rows and column

            //needed for multiplication for that process

            int subarray\_size = rowC\*colC;

            int \*subarrayA = (int \*)malloc(subarray\_size \* sizeof(int));

            int \*subarrayB = (int \*)malloc(subarray\_size \* sizeof(int));

            if (!subarrayA || !subarrayB) {

                perror("Subarray allocation failed");

                MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

            }

            //loop through matrix A to copy all the necessary cells

            for (i = row\_start\_point; i < row\_end\_point; i++) {

                for (k = 0; k < commonPoint; k++){

                    subarrayA[((i - row\_start\_point) \* colA) + k] = pMatrixA[(i \* colA) + k];

                }

            }

            //loop through matrix B to copy all the necessary cells

            for(j = col\_start\_point; j < col\_end\_point; j++){

                for (k = 0; k < commonPoint; k++) {

                    subarrayB[((j - col\_start\_point) \* rowA) + k] = pMatrixB[(k \* colB) + j];

                }

            }

            //pack size according to the subarray size

            MPI\_Pack\_size(subarray\_size + (sizeof(unsigned long long) \* 2), MPI\_INT, MPI\_COMM\_WORLD, &pack\_size);

            int buffer\_size = pack\_size;

            //initialise buffer for subarray to send to process

            char \*bufferA = (char \*)malloc(buffer\_size);

            char \*bufferB = (char \*)malloc(buffer\_size);

            clock\_gettime(CLOCK\_MONOTONIC, &startComp);

            //pack the time and subarrays

            MPI\_Pack(&startComp.tv\_sec, 1, MPI\_UINT64\_T, bufferA, buffer\_size, &position, MPI\_COMM\_WORLD);

            MPI\_Pack(&startComp.tv\_nsec, 1, MPI\_UINT64\_T, bufferA, buffer\_size, &position, MPI\_COMM\_WORLD);

            MPI\_Pack(subarrayA, subarray\_size, MPI\_INT, bufferA, buffer\_size, &position, MPI\_COMM\_WORLD);

            //send the subarrays and time together to process

            MPI\_Send(bufferA, position, MPI\_PACKED, temprank, 4, MPI\_COMM\_WORLD);

            position = 0;

            MPI\_Pack(subarrayB, subarray\_size, MPI\_INT, bufferB, buffer\_size, &position, MPI\_COMM\_WORLD);

            MPI\_Send(bufferB, position, MPI\_PACKED, temprank, 5, MPI\_COMM\_WORLD);

        }

    } else {

        //for other processes receive from root rank

        //receive their own row/col start and end

        MPI\_Recv(&row\_start\_point, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        MPI\_Recv(&row\_end\_point, 1, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        MPI\_Recv(&col\_start\_point, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        MPI\_Recv(&col\_end\_point, 1, MPI\_INT, 0, 3, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        //preparation to receive the packed subarrays

        int totalElem = rowC\*colC + (sizeof(unsigned long long) \* 2);

        MPI\_Pack\_size(totalElem, MPI\_INT, MPI\_COMM\_WORLD, &pack\_size);

        int buffer\_size = pack\_size;

        char \*recv\_bufferA = (char \*)malloc(buffer\_size);

        char \*recv\_bufferB = (char \*)malloc(buffer\_size);

        if (!recv\_bufferA || !recv\_bufferB) {

            perror("Buffer allocation failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        //receive the subarrays using buffer

        MPI\_Recv(recv\_bufferA, buffer\_size, MPI\_PACKED, 0, 4, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        MPI\_Recv(recv\_bufferB, buffer\_size, MPI\_PACKED, 0, 5, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        clock\_gettime(CLOCK\_MONOTONIC, &endComp);

        pMatrixA = (int \*)malloc(totalElem \* sizeof(int)); // Allocate memory for pMatrixA

        pMatrixB = (int \*)malloc(totalElem \* sizeof(int)); // Allocate memory for pMatrixB

        if (!pMatrixA || !pMatrixB) {

            perror("Memory allocation for pMatrixA failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        position = 0;

        // Unpack timestamp and subarrays

        MPI\_Unpack(recv\_bufferA, buffer\_size, &position, &startComp.tv\_sec, 1, MPI\_UINT64\_T, MPI\_COMM\_WORLD);

        MPI\_Unpack(recv\_bufferA, buffer\_size, &position, &startComp.tv\_nsec, 1, MPI\_UINT64\_T, MPI\_COMM\_WORLD);

        MPI\_Unpack(recv\_bufferA, buffer\_size, &position, pMatrixA, (rowC\*colC), MPI\_INT, MPI\_COMM\_WORLD);

        position = 0;

        MPI\_Unpack(recv\_bufferB, buffer\_size, &position, pMatrixB, totalElem, MPI\_INT, MPI\_COMM\_WORLD);

        time\_taken = (endComp.tv\_sec - startComp.tv\_sec) \* 1e9;

        time\_taken = (time\_taken + (endComp.tv\_nsec - startComp.tv\_nsec)) \* 1e-9;

        //print out the timetaken

        printf("Rank %d took %lf (s) to receive matrix from Root process\n", my\_rank, time\_taken);

    }

    if(my\_rank == 0) { //rank 0 follow the normal way as the matrix arrange same as before

        row\_start\_point = 0;

        row\_end\_point = row\_start\_point + threadRowDiv;

        col\_start\_point = 0;

        col\_end\_point = threadColDiv;

        for (i = row\_start\_point; i < row\_end\_point; i++) {

            for (j = col\_start\_point; j < col\_end\_point; j++) {

                    pMatrixC[i \* colC + j] = 0; // Initialize the result element

                for (k = 0; k < commonPoint; k++) {

                    pMatrixC[(i\*colC)+j] += (pMatrixA[(i\*colA)+k] \* pMatrixB[(k\*colB)+j]);

                }

            }

        }

    } else{ //the others the other way as it is transposed

        for (i = row\_start\_point; i < row\_end\_point; i++) {

            for (j = col\_start\_point; j < col\_end\_point; j++) {

                pMatrixC[i \* colC + j] = 0; // Initialize the result element

                for (k = 0; k < commonPoint; k++) {

                    // Matrix A row remains the same

                    int elemA = pMatrixA[((i - row\_start\_point) \* commonPoint) + k];

                    // Matrix B column is treated as a row

                    int elemB = pMatrixB[((j - col\_start\_point) \* commonPoint) + k];

                    // Perform the multiplication

                    pMatrixC[(i \* colC) + j] += elemA \* elemB;

                }

            }

        }

    }

    unsigned long long \*globalMatrixC = NULL;

    if(my\_rank != 0){

        //send local pMatrixC back to root

        MPI\_Pack\_size(rowC\*colC, MPI\_UNSIGNED\_LONG\_LONG, MPI\_COMM\_WORLD, &pack\_size);

        int buffer\_size = pack\_size;

        char \*bufferC = (char \*)malloc(buffer\_size);

        position = 0;

        MPI\_Pack(pMatrixC, rowC \* colC, MPI\_UNSIGNED\_LONG\_LONG, bufferC, buffer\_size, &position, MPI\_COMM\_WORLD);

        MPI\_Send(bufferC, position, MPI\_PACKED, 0, my\_rank, MPI\_COMM\_WORLD);

        MPI\_Send(&row\_start\_point, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

        MPI\_Send(&row\_end\_point, 1, MPI\_INT, 0, 1, MPI\_COMM\_WORLD);

        MPI\_Send(&col\_start\_point, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);

        MPI\_Send(&col\_end\_point, 1, MPI\_INT, 0, 3, MPI\_COMM\_WORLD);

    }

    if (my\_rank == 0){

        globalMatrixC = (unsigned long long \*)calloc((rowC \* colC), sizeof(unsigned long long));

        if (!globalMatrixC) {

            perror("Memory allocation for globalMatrixC failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        //set all values to 0

        for(i = 0; i < rowC\* colC; i++){

            globalMatrixC[i] = 0;

        }

        //combine the rank 0 matrixC to globalmatrixC

        combineLocalMatrices(pMatrixC, globalMatrixC, row\_start\_point, row\_end\_point, col\_start\_point, col\_end\_point, colC);

        //loop through each rank and receive their local matrixC from each rank

        //then combine it to globalmatrixC

        for (i = 1; i < size; i++){

            MPI\_Pack\_size(rowC\*colC, MPI\_UNSIGNED\_LONG\_LONG, MPI\_COMM\_WORLD, &pack\_size);

            int buffer\_size = pack\_size;

            char \*recv\_bufferC = (char \*)malloc(buffer\_size);

            //receive bufferC from each rank

            MPI\_Recv(recv\_bufferC, buffer\_size, MPI\_PACKED, i, i, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            position = 0;

            MPI\_Unpack(recv\_bufferC, buffer\_size, &position, pMatrixC, rowC\*colC, MPI\_UNSIGNED\_LONG\_LONG, MPI\_COMM\_WORLD);

            MPI\_Recv(&row\_start\_point, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            MPI\_Recv(&row\_end\_point, 1, MPI\_INT, i, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            MPI\_Recv(&col\_start\_point, 1, MPI\_INT, i, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            MPI\_Recv(&col\_end\_point, 1, MPI\_INT, i, 3, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            combineLocalMatrices(pMatrixC, globalMatrixC, row\_start\_point, row\_end\_point, col\_start\_point, col\_end\_point, colC);

        }

    }

    //checking each rank's row/col start and end

    // printf("rank: %d, row\_start: %d, row\_end: %d, col\_start: %d, col\_end: %d", my\_rank, row\_start\_point, row\_end\_point, col\_start\_point, col\_end\_point);

    // fflush(stdout);

    //waiting for each rank to finish

    MPI\_Barrier(MPI\_COMM\_WORLD);

    MPI\_Finalize();

    if (my\_rank == 0) {

        // Write result to file

        printf("Writing Matrix C - Start\n");

        FILE \*pFileC = fopen("MC\_500x500.bin", "wb");

        fwrite(&rowC, sizeof(int), 1, pFileC);

        fwrite(&colC, sizeof(int), 1, pFileC);

        //write row by row

        for (i = 0; i < rowC; i++) {

            fwrite(&globalMatrixC[i \* colC], sizeof(unsigned long long), colC, pFileC);

        }

        fclose(pFileC);

        printf("Writing Matrix C - Done\n");

        // Time calculation of overall program

        clock\_gettime(CLOCK\_MONOTONIC, &end);

        time\_taken = (end.tv\_sec - start.tv\_sec) \* 1e9;

        time\_taken = (time\_taken + (end.tv\_nsec - start.tv\_nsec)) \* 1e-9;

        printf("Elapsed Time: %f seconds\n", time\_taken);

    }

    return 0;

}

**Check Correctness with 500x500 Matrix Mul**

**Serial code provided**

A close-up of a grid

Description automatically generated

**Parallel code with MPI**

A group of colorful squares

Description automatically generated with medium confidence

**Task 2 – Table & Analysis**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Number of nodes (Specify 1 if only using your local computer): 1**  **Number of CPU cores or logical processes per node: 1**  **Number of MPI processes: 8**  **Theoretical speed up: 8.466** | | | | | |
| **Matrix Size** | **500x500** | **1000x1000** | **2000x2000** | **3000x3000** | **4000x4000** |
| **Serial Time** | 1.036289 | 7.205710 | 52.871768 | 180.674973 | 424.491226 |
| **Parallel Time** | 0.576 | 2.419 | 13.170 | 39.405 | 89.506 |
| **MPI Communication time** | 0.017 | 0.272 | 0.260 | 0.478 | 0.808 |
| **Speed Up** | 1.799 | 2.979 | 4.015 | 4.585 | 4.743 |

**Compare and analysis**

As the matrix size increases the speed up improves from 1.799 for 500x500 to 4.743 for 4000x4000. Larger matrix sizes allow more computational work to be distributed across the threads. The actual speed up is more than the theoretical speed up for the matrix that are 3000x3000 and above.

**Why it achieves lower speed up compared to posix**

Mpi processes it achieve lower speed up compared to posix. This is due to mpi has distributed memory, so it needs to send the matrix across the processes. Hence, the communication time is more than using posix as posix using shared memory which is faster.

**When problem size increases**

Beyond a certain point, increasing the matrix size doesn’t result in increase in speed up. For example, the jump from 3000x3000 to 4000x4000 shows only a slight increase in speed up due to the factors like increased communication time according to the table 4000x4000 communication time is 2 times of 3000x3000.

**Why actual value is close to theoretical value**

For larger matrices, the actual speed up close to the theoretical value because the parallel workload increases, reducing the impact of communication overhead.

**500x500**

A screenshot of a computer program

Description automatically generated

**1000x1000**

**A screenshot of a computer program

Description automatically generated**

**2000x2000**

A screenshot of a computer program

Description automatically generated

**3000x3000**

A screenshot of a computer program

Description automatically generated

**4000x4000**

A screenshot of a computer program

Description automatically generated

**Task 3 – Code**

#include <stdio.h>

#include <stdlib.h>

#include <memory.h>

#include <time.h>

#include <mpi.h>

#include <pthread.h>

#define NUM\_THREADS 2

int row\_start\_point, row\_end\_point;

int col\_start\_point, col\_end\_point;

unsigned long long \*pMatrixC;

int rowA, rowB, rowC, colA, colB, colC, commonPoint;

int \*pMatrixA = NULL, \*pMatrixB = NULL;

int my\_rank, size;

pthread\_mutex\_t mutex;

unsigned long long \*globalMatrixC = NULL;

//combine local matrices to the global matrix

void combineLocalMatrices(unsigned long long \*localMatrix, unsigned long long \*globalMatrix, int row\_start, int row\_end, int col\_start, int col\_end, int colC) {

    //only loop through the necessary row and column for the local matrix

    //+= is because each of the elem in global matrix initially is 0, so x+0 = x

    for(int row = row\_start; row < row\_end; row++){

        for (int col = col\_start; col < col\_end; col++){

            // Add localMatrix to globalMatrix element by element

            globalMatrix[row \* colC + col] += localMatrix[row \* colC + col];

        }

    }

}

void \*ThreadReadMatrixA(void \*arg) {

    //assign row/col start end for each thread to read

    int thread\_id = \*((int \*)arg);

    int rows\_per\_thread = rowA / NUM\_THREADS;

    int start\_row = thread\_id \* rows\_per\_thread;

    int end\_row = (thread\_id == NUM\_THREADS - 1) ? rowA : start\_row + rows\_per\_thread;

    FILE \*pFileA = fopen("MA\_500x500.bin", "rb");

    if (!pFileA) {

        perror("File A opening failed");

        pthread\_exit(NULL);

    }

    //loop through and copy to pmatrixA

    fseek(pFileA, sizeof(int) \* 2 + sizeof(int) \* start\_row \* colA, SEEK\_SET); // Seek to the appropriate row

    for (int i = start\_row; i < end\_row; i++) {

        fread(&pMatrixA[i \* colA], sizeof(int), colA, pFileA);

    }

    fclose(pFileA);

    pthread\_exit(NULL);

}

void \*ThreadReadMatrixB(void \*arg) {

    //assign each threads row/col to start and end to read

    int thread\_id = \*((int \*)arg);

    int cols\_per\_thread = colB / NUM\_THREADS;

    int start\_col = thread\_id \* cols\_per\_thread;

    int end\_col = (thread\_id == NUM\_THREADS - 1) ? colB : start\_col + cols\_per\_thread;

    FILE \*pFileB = fopen("MB\_500x500.bin", "rb");

    if (!pFileB) {

        perror("File B opening failed");

        pthread\_exit(NULL);

    }

    fseek(pFileB, sizeof(int) \* 2, SEEK\_SET); // Skip the matrix dimensions

    for (int i = 0; i < rowB; i++) {

        //loop through to read and put in matrix B

        int \*row\_buffer = (int \*)malloc(sizeof(int) \* colB); // Temporary buffer for the row

        fread(row\_buffer, sizeof(int), colB, pFileB);        // Read the entire row into buffer

        memcpy(&pMatrixB[i \* colB + start\_col], &row\_buffer[start\_col], sizeof(int) \* (end\_col - start\_col));  // Copy only the required columns

        free(row\_buffer);

    }

    fclose(pFileB);

    pthread\_exit(NULL);

}

void \*ThreadWriteMatrixC(void \*arg) {

    //use multiple thread to write to matrixC file

    int thread\_id = \*((int \*)arg);

    int rows\_per\_thread = rowC / NUM\_THREADS;

    int start\_row = thread\_id \* rows\_per\_thread;

    int end\_row = (thread\_id == NUM\_THREADS - 1) ? rowC : start\_row + rows\_per\_thread;

    // Use write mode for a fresh file

    FILE \*pFileC = fopen("MC\_500x500.bin", "rb+");

    if (!pFileC) {

        perror("File C opening failed");

        pthread\_exit(NULL);

    }

    // Synchronize file writing to prevent race conditions

    pthread\_mutex\_lock(&mutex);

    //write to globalmatrixC

    fseek(pFileC, sizeof(int) \* 2 + sizeof(unsigned long long) \* start\_row \* colC, SEEK\_SET); // Seek to the appropriate row

    for (int i = start\_row; i < end\_row; i++) {

        fwrite(&globalMatrixC[i \* colC], sizeof(unsigned long long), colC, pFileC);

    }

    pthread\_mutex\_unlock(&mutex);//unlock after done for other threads to write

    fclose(pFileC);

    pthread\_exit(NULL);

}

void \*ThreadComp(void \*pArg){

    int i, j, k;

    int threads\_num = \*((int\*)pArg);

    //distribute the rows and cols to each thread

    int row\_per\_thread = (row\_end\_point - row\_start\_point) / NUM\_THREADS;

    int rptr = (row\_end\_point - row\_start\_point) % NUM\_THREADS;

    int start\_point = row\_start\_point + threads\_num \* row\_per\_thread;

    int end\_point = start\_point + row\_per\_thread;

    if(threads\_num == NUM\_THREADS - 1)

        end\_point += rptr;

    if (my\_rank == 0){//rank 0 follow the normal way as the matrix arrange same as before

        for (i = start\_point; i < end\_point ; i++) {

                for (j = col\_start\_point; j < col\_end\_point; j++) {

                        pMatrixC[i \* colC + j] = 0; // Initialize the result element

                    for (k = 0; k < commonPoint; k++) {

                        pMatrixC[(i\*colC)+j] += (pMatrixA[(i\*colA)+k] \* pMatrixB[(k\*colB)+j]);

                    }

                }

            }

    } else{//the other ranks diff as matrix b is transposed

        for (i = start\_point; i < end\_point; i++) {

            for (j = col\_start\_point; j < col\_end\_point; j++) {

                pMatrixC[i \* colC + j] = 0; // Initialize the result element

                for (k = 0; k < commonPoint; k++) {

                    // Matrix A row remains the same

                    int elemA = pMatrixA[((i - row\_start\_point) \* commonPoint) + k];

                    // Matrix B column is treated as a row

                    int elemB = pMatrixB[((j - col\_start\_point) \* commonPoint) + k];

                    // Perform the multiplication

                    pMatrixC[(i \* colC) + j] += elemA \* elemB;

                }

            }

        }

    }

    pthread\_exit(NULL);

}

int main(int argc, char \*argv[])

{

    // Variables

    int i = 0, j = 0, k = 0;

    int position;

    int pack\_size;

    //initialise mutex

    pthread\_mutex\_init(&mutex, NULL);

    pthread\_t threads[NUM\_THREADS];

    int thread\_ids[NUM\_THREADS];

    /\* Clock information \*/

    struct timespec start, end, startComp, endComp;

    double time\_taken;

    clock\_gettime(CLOCK\_MONOTONIC, &start);

    MPI\_Init(&argc, &argv);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);

    // 1. Read Matrix A

    rowA = colA = rowB = colB = 0;

    if (my\_rank == 0) {

        printf("Matrix Multiplication using 1-Dimension Arrays - Start\n\n");

        // Reading Matrix A

        printf("Reading Matrix A - Start\n");

        FILE \*pFileA = fopen("MA\_500x500.bin", "rb");

        if (!pFileA) {

            perror("File A opening failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        fread(&rowA, sizeof(int), 1, pFileA);

        fread(&colA, sizeof(int), 1, pFileA);

        pMatrixA = (int \*)malloc((rowA \* colA) \* sizeof(int));

        if (pMatrixA == NULL) {

            perror("Memory allocation for Matrix A failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

       // Create threads to read Matrix A

        for (int i = 0; i < NUM\_THREADS; i++) {

            thread\_ids[i] = i;

            pthread\_create(&threads[i], NULL, ThreadReadMatrixA, &thread\_ids[i]);

        }

        // Join threads after reading Matrix A

        for (int i = 0; i < NUM\_THREADS; i++) {

            pthread\_join(threads[i], NULL);

        }

        printf("Reading Matrix A - Done\n");

        // Reading Matrix B

        printf("Reading Matrix B - Start\n");

        FILE \*pFileB = fopen("MB\_500x500.bin", "rb");

        if (!pFileB) {

            perror("File B opening failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        fread(&rowB, sizeof(int), 1, pFileB);

        fread(&colB, sizeof(int), 1, pFileB);

        pMatrixB = (int \*)malloc((rowB \* colB) \* sizeof(int));

        if (pMatrixB == NULL) {

            perror("Memory allocation for Matrix B failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        // Create threads to read Matrix B

        for (int i = 0; i < NUM\_THREADS; i++) {

            thread\_ids[i] = i;

            pthread\_create(&threads[i], NULL, ThreadReadMatrixB, &thread\_ids[i]);

        }

        // Join threads after reading Matrix B

        for (int i = 0; i < NUM\_THREADS; i++) {

            pthread\_join(threads[i], NULL);

        }

        printf("Reading Matrix B - Done\n");

        // Initialize Matrix C

        rowC = rowA;

        colC = colB;

        commonPoint = colA; // Initialize commonPoint

    }

    //broadcast rows to all the processses

    MPI\_Bcast(&rowA, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&colA, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&rowB, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&colB, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&rowC, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&colC, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(&commonPoint, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); // Broadcast commonPoint

    //assign local matrix result

    pMatrixC = (unsigned long long \*)calloc((rowC \* colC), sizeof(unsigned long long));

    //calculate the tile partition

    //using result will divide into half even number process left odd number process right

    int threadColDiv = colC / 2;

    int threadColDivRemain = colC % 2;

    int threadRowDiv = rowC / (size / 2);

    int threadRowDivRemain = rowC % (size / 2);

    //root rank calculate the subarray for matrixA and matrixB

    //send it to the corresponding rank

    if (my\_rank == 0) {

        for (int temprank = 1; temprank < size; temprank++) {

            position = 0;

            if (temprank % 2 == 0) {

                // Even thread

                if (temprank == (size - 2)) {

                    // Last even thread

                    row\_start\_point = (temprank / 2) \* threadRowDiv;

                    row\_end\_point = row\_start\_point + threadRowDiv + threadRowDivRemain;

                } else {

                    // Not last even thread

                    row\_start\_point = (temprank / 2) \* threadRowDiv;

                    row\_end\_point = row\_start\_point + threadRowDiv;

                }

                col\_start\_point = 0;

                col\_end\_point = threadColDiv;

            } else {

                // Odd thread

                if (temprank == (size - 1)) {

                    // Last odd thread

                    row\_start\_point = (temprank / 2) \* threadRowDiv;

                    row\_end\_point = row\_start\_point + threadRowDiv + threadRowDivRemain;

                } else {

                    // Not last odd thread

                    row\_start\_point = (temprank / 2) \* threadRowDiv;

                    row\_end\_point = row\_start\_point + threadRowDiv;

                }

                col\_start\_point = threadColDiv;

                col\_end\_point = col\_start\_point + threadColDiv + threadColDivRemain;

            }

            //send row/col start and end point to each rank

            MPI\_Send(&row\_start\_point, 1, MPI\_INT, temprank, 0, MPI\_COMM\_WORLD);

            MPI\_Send(&row\_end\_point, 1, MPI\_INT, temprank, 1, MPI\_COMM\_WORLD);

            MPI\_Send(&col\_start\_point, 1, MPI\_INT, temprank, 2, MPI\_COMM\_WORLD);

            MPI\_Send(&col\_end\_point, 1, MPI\_INT, temprank, 3, MPI\_COMM\_WORLD);

            int subarray\_size = rowC\*colC;

            int \*subarrayA = (int \*)malloc(subarray\_size \* sizeof(int));

            int \*subarrayB = (int \*)malloc(subarray\_size \* sizeof(int));

            if (!subarrayA || !subarrayB) {

                perror("Subarray allocation failed");

                MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

            }

            //copy necessary cells to subarrayA

            for (i = row\_start\_point; i < row\_end\_point; i++) {

                for (k = 0; k < commonPoint; k++){

                    subarrayA[((i - row\_start\_point) \* colA) + k] = pMatrixA[(i \* colA) + k];

                }

            }

            //copy ncessary cells to subarrayB for matrix multiplication

            for(j = col\_start\_point; j < col\_end\_point; j++){

                for (k = 0; k < commonPoint; k++) {

                    subarrayB[((j - col\_start\_point) \* rowA) + k] = pMatrixB[(k \* colB) + j];

                }

            }

            //calculate the packsize of subarrays

            MPI\_Pack\_size(subarray\_size + (sizeof(unsigned long long) \* 2), MPI\_INT, MPI\_COMM\_WORLD, &pack\_size);

            int buffer\_size = pack\_size;

            char \*bufferA = (char \*)malloc(buffer\_size);

            char \*bufferB = (char \*)malloc(buffer\_size);

            clock\_gettime(CLOCK\_MONOTONIC, &startComp);

            //pack the time and subarrays to send to the corresponding rank

            MPI\_Pack(&startComp.tv\_sec, 1, MPI\_UINT64\_T, bufferA, buffer\_size, &position, MPI\_COMM\_WORLD);

            MPI\_Pack(&startComp.tv\_nsec, 1, MPI\_UINT64\_T, bufferA, buffer\_size, &position, MPI\_COMM\_WORLD);

            MPI\_Pack(subarrayA, subarray\_size, MPI\_INT, bufferA, buffer\_size, &position, MPI\_COMM\_WORLD);

            MPI\_Send(bufferA, position, MPI\_PACKED, temprank, 4, MPI\_COMM\_WORLD);

            position = 0;

            MPI\_Pack(subarrayB, subarray\_size, MPI\_INT, bufferB, buffer\_size, &position, MPI\_COMM\_WORLD);

            MPI\_Send(bufferB, position, MPI\_PACKED, temprank, 5, MPI\_COMM\_WORLD);

        }

    } else {

        //recv row/col start n end point from root rank

        MPI\_Recv(&row\_start\_point, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        MPI\_Recv(&row\_end\_point, 1, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        MPI\_Recv(&col\_start\_point, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        MPI\_Recv(&col\_end\_point, 1, MPI\_INT, 0, 3, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        //preparation to receive submatrix from root rank

        int totalElem = rowC\*colC + (sizeof(unsigned long long) \* 2);

        MPI\_Pack\_size(totalElem, MPI\_INT, MPI\_COMM\_WORLD, &pack\_size);

        int buffer\_size = pack\_size;

        char \*recv\_bufferA = (char \*)malloc(buffer\_size);

        char \*recv\_bufferB = (char \*)malloc(buffer\_size);

        if (!recv\_bufferA || !recv\_bufferB) {

            perror("Buffer allocation failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        //receive subarray

        MPI\_Recv(recv\_bufferA, buffer\_size, MPI\_PACKED, 0, 4, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        MPI\_Recv(recv\_bufferB, buffer\_size, MPI\_PACKED, 0, 5, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

        clock\_gettime(CLOCK\_MONOTONIC, &endComp);

        pMatrixA = (int \*)malloc(totalElem \* sizeof(int)); // Allocate memory for pMatrixA

        pMatrixB = (int \*)malloc(totalElem \* sizeof(int)); // Allocate memory for pMatrixB

        if (!pMatrixA || !pMatrixB) {

            perror("Memory allocation for pMatrixA failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        position = 0;

        // Unpack timestamp and matrix and store in local pmatrix

        MPI\_Unpack(recv\_bufferA, buffer\_size, &position, &startComp.tv\_sec, 1, MPI\_UINT64\_T, MPI\_COMM\_WORLD);

        MPI\_Unpack(recv\_bufferA, buffer\_size, &position, &startComp.tv\_nsec, 1, MPI\_UINT64\_T, MPI\_COMM\_WORLD);

        MPI\_Unpack(recv\_bufferA, buffer\_size, &position, pMatrixA, (rowC\*colC), MPI\_INT, MPI\_COMM\_WORLD);

        position = 0;

        MPI\_Unpack(recv\_bufferB, buffer\_size, &position, pMatrixB, totalElem, MPI\_INT, MPI\_COMM\_WORLD);

        time\_taken = (endComp.tv\_sec - startComp.tv\_sec) \* 1e9;

        time\_taken = (time\_taken + (endComp.tv\_nsec - startComp.tv\_nsec)) \* 1e-9;

        //time taken for ranks to receive from the root rank

        printf("Rank %d took %lf (s) to receive matrix from Root process\n", my\_rank, time\_taken);

    }

    if(my\_rank == 0) { //rank 0 default setting values

        row\_start\_point = 0;

        row\_end\_point = row\_start\_point + threadRowDiv;

        col\_start\_point = 0;

        col\_end\_point = threadColDiv;

    }

    //create threads to calculate their tile results

    for (i = 0; i < NUM\_THREADS; i++){

        thread\_ids[i] = i;

        pthread\_create(&threads[i], NULL, ThreadComp, &thread\_ids[i]);

    }

    //join after calculating

    for (i = 0; i< NUM\_THREADS; i++){

        pthread\_join(threads[i], NULL);

    }

    if(my\_rank != 0){

        //send local pMatrixC back to root

        MPI\_Pack\_size(rowC\*colC, MPI\_UNSIGNED\_LONG\_LONG, MPI\_COMM\_WORLD, &pack\_size);

        int buffer\_size = pack\_size;

        char \*bufferC = (char \*)malloc(buffer\_size);

        position = 0;

        MPI\_Pack(pMatrixC, rowC \* colC, MPI\_UNSIGNED\_LONG\_LONG, bufferC, buffer\_size, &position, MPI\_COMM\_WORLD);

        MPI\_Send(bufferC, position, MPI\_PACKED, 0, my\_rank, MPI\_COMM\_WORLD);

        MPI\_Send(&row\_start\_point, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD);

        MPI\_Send(&row\_end\_point, 1, MPI\_INT, 0, 1, MPI\_COMM\_WORLD);

        MPI\_Send(&col\_start\_point, 1, MPI\_INT, 0, 2, MPI\_COMM\_WORLD);

        MPI\_Send(&col\_end\_point, 1, MPI\_INT, 0, 3, MPI\_COMM\_WORLD);

    }

    if (my\_rank == 0){

        globalMatrixC = (unsigned long long \*)calloc((rowC \* colC), sizeof(unsigned long long));

        if (!globalMatrixC) {

            perror("Memory allocation for globalMatrixC failed");

            MPI\_Abort(MPI\_COMM\_WORLD, EXIT\_FAILURE);

        }

        //set all values to 0

        for(i = 0; i < rowC\* colC; i++){

            globalMatrixC[i] = 0;

        }

        //combine root rank local matrixC to globalmatrixC

        combineLocalMatrices(pMatrixC, globalMatrixC, row\_start\_point, row\_end\_point, col\_start\_point, col\_end\_point, colC);

       //loop through each rank and receive their local matrixC from each rank

        //then combine it to globalmatrixC

        for (i = 1; i < size; i++){

            MPI\_Pack\_size(rowC\*colC, MPI\_UNSIGNED\_LONG\_LONG, MPI\_COMM\_WORLD, &pack\_size);

            int buffer\_size = pack\_size;

            char \*recv\_bufferC = (char \*)malloc(buffer\_size);

            //receive and unpack local matrix C that receive from other ranks

            MPI\_Recv(recv\_bufferC, buffer\_size, MPI\_PACKED, i, i, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            position = 0;

            MPI\_Unpack(recv\_bufferC, buffer\_size, &position, pMatrixC, rowC\*colC, MPI\_UNSIGNED\_LONG\_LONG, MPI\_COMM\_WORLD);

            MPI\_Recv(&row\_start\_point, 1, MPI\_INT, i, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            MPI\_Recv(&row\_end\_point, 1, MPI\_INT, i, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            MPI\_Recv(&col\_start\_point, 1, MPI\_INT, i, 2, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            MPI\_Recv(&col\_end\_point, 1, MPI\_INT, i, 3, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

            //combine it with globalmatrixC

            combineLocalMatrices(pMatrixC, globalMatrixC, row\_start\_point, row\_end\_point, col\_start\_point, col\_end\_point, colC);

        }

    }

    //wait for all ranks

    MPI\_Barrier(MPI\_COMM\_WORLD);

    MPI\_Finalize();

    if (my\_rank == 0) {

        // Write result to file

        printf("Writing Matrix C - Start\n");

        FILE \*pFileC = fopen("MC\_500x500.bin", "wb");

        fwrite(&rowC, sizeof(int), 1, pFileC);

        fwrite(&colC, sizeof(int), 1, pFileC);

        // Create threads to write Matrix C

        for (int i = 0; i < NUM\_THREADS; i++) {

            thread\_ids[i] = i;

            pthread\_create(&threads[i], NULL, ThreadWriteMatrixC, &thread\_ids[i]);

        }

        // Join threads after writing Matrix C

        for (int i = 0; i < NUM\_THREADS; i++) {

            pthread\_join(threads[i], NULL);

        }

        fclose(pFileC);

        printf("Writing Matrix C - Done\n");

        // Time calculation of overall program

        clock\_gettime(CLOCK\_MONOTONIC, &end);

        time\_taken = (end.tv\_sec - start.tv\_sec) \* 1e9;

        time\_taken = (time\_taken + (end.tv\_nsec - start.tv\_nsec)) \* 1e-9;

        printf("Elapsed Time: %f seconds\n", time\_taken);

    }

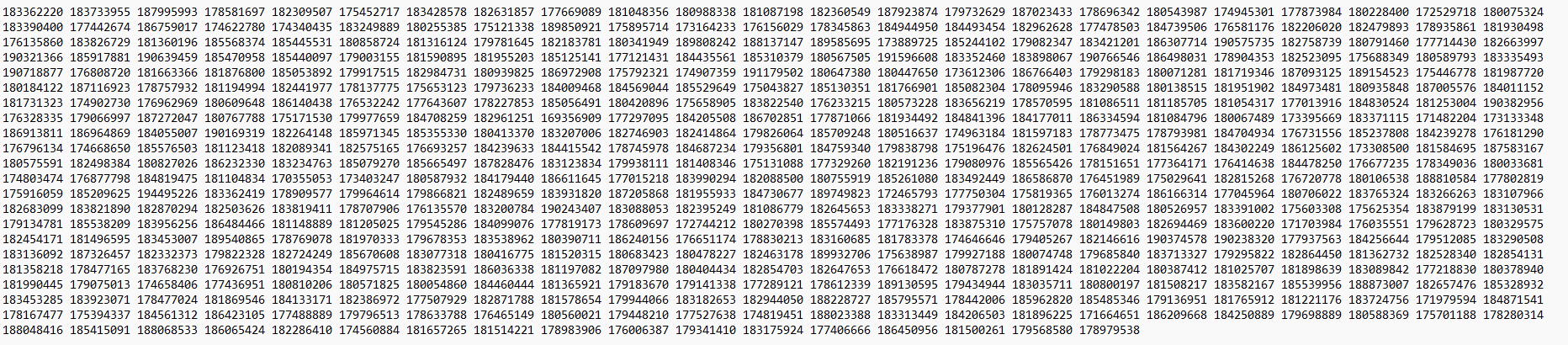
    return 0;

}

**Check correctness with Matrix Mul**

**500x500**

**Provided Matrix Mul**



**Task 3 MPI with threads**

A grid of squares with black and white lines

Description automatically generated with medium confidence

**Task 3 – Table & Analysis**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Number of nodes (Specify 1 if only using your local computer): 1**  **Number of CPU cores or logical processes per node: 8**  **Number of MPI processes: 4**  **Number of threads per MPI process: 2**  **Theoretical speed up:** 4.573 | | | | | |
| **Matrix Size** | **500x500** | **1000x1000** | **2000x2000** | **3000x3000** | **4000x4000** |
| **Serial Time** | 1.036289 | 7.205710 | 52.871768 | 180.674973 | 424.491226 |
| **Task 3 Parallel Time** | 0.557 | 2.146 | 12.208 | 39.505 | 89.257 |
| **Speed Up** | 1.860 | 3.358 | 4.331 | 4.573 | 4.755 |
| **MPI communication time** | 0.006868 | 0.02038 | 0.074234 | 0.1633 | 0.230557 |
| **Task 2 Parallel Time** | 0.576 | 2.419 | 13.170 | 39.405 | 89.506 |

**Compare and analysis**

The communication time in table 3 is significantly lower than table 2. This results in better speed up in table 3 compared to table 2. The reasons of why task 3 achieve a better speed up:

1. **Hybrid approach**

Task 3 used hybrid approach which is 4 processes and 2 threads per processes, this result in smaller communication time in task 3. Thus, reduces the overhead of message passing, allowing for better parallel performance. By distributing task to MPI and each process will have their own threads to distribute the workload again, this lower down the communication time and increase the computational power as threads using shared memory so they can compute faster compared to MPI.

1. **Decrease communication time**

The speed up in task 3 is slightly higher than task 2. This indicates better performance when using 4 MPI processes with 2 threads per process. This is due to lower communication overhead compared to 8 mpi processes according to the table above by using hybrid approach, it significantly lower down the MPI communication time by 2 times. In Task 3, each MPI process is handling more data internally, which reduces the need for frequent interprocess communication whereas in task 2 root rank needs to distribute tasks to 8 processes which increase in communication time. Additionally, since there are 2 threads per MPI process so the work can be distributed within each process, so they don’t need to exchange much data between processes.

**Actual speed up is close to theoretical speed up**

Furthermore, when the problem size increases the speed up increase as well. The speed up of task 3 began to more than the theoretical speed up when the matrix size is 3000x3000 and above. This is due to for larger matrices the parallel workload increases. Hence, increase in the speed up.

**500x500**

A screen shot of a computer program

Description automatically generated

**1000x1000**

A screen shot of a computer program

Description automatically generated

**2000x2000**

A screen shot of a computer program

Description automatically generated

**3000x3000**

A screen shot of a computer program

Description automatically generated

**4000x4000**

A screenshot of a computer program

Description automatically generated

**AI Acknowledgment Statement**

I would like to acknowledge the contributions of artificial intelligence technologies (ChatGPT) that have supported my work.