Matrix Multiplication using Pthread

OS – LAB 2

Abdullah Mohamed AbdelHakim Mohamed Gomaa

19015953

**Code Summary:**

The program defines three functions that implement three different methods of matrix multiplication. The first method is a normal matrix multiplication that uses nested for-loops to multiply the two matrices. The second method creates a thread for each row of the first matrix, while the third method creates a thread for each element of the resulting matrix.

The main function of the program begins by opening the input files and reading the matrix sizes. It then dynamically allocates memory for the matrices to be multiplied and the resulting matrix. Next, the program uses threads to perform the three multiplication methods and creates output files for each method. These output files contain the resulting matrix, the number of threads used for each method, and the time taken to complete each method.

1. Global Variables

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Global Variables \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int rowsA = 0;

int colsA = 0;

int rowsB = 0;

int colsB = 0;

int rowsC = 0;

int colsC = 0;

int\*\* matrix\_A;

int\*\* matrix\_B;

int\*\* matrix\_C;

int numOfThreads = 0;

struct args

{

    int row;

    int col;

};

* Initialize the variables for storing number of rows and columns for every matrix, the variables for storing the matrices, the variable for storing number of threads, and create struct called args to be sent as an argument to the thread callback function.

1. Threads Functions

* Method 1: Normal multiplication is a simple matrix multiplication algorithm that multiplies two matrices in the traditional way, by taking the dot product of each row of the first matrix with each column of the second matrix.

1. // Method 1: normal multiplication
2. void \*normalMultiply()
3. {
4. for(int i=0; i<rowsA; i++)
5. {
6. for (int j = 0; j < colsB; j++)
7. {
8. matrix\_C[i][j] = 0;
9. for (int k = 0; k < colsA; k++)
10. {
11. matrix\_C[i][j] += matrix\_A[i][k] \* matrix\_B[k][j];
12. }
13. }
14. }
15. pthread\_exit(NULL);
16. }

* Method 2: A thread per row is a matrix multiplication algorithm that uses a separate thread for each row of the first matrix, which speeds up the computation by allowing multiple rows to be processed simultaneously.

// Method 2: a thread per row

void \*rowMultiply(void \*arg)

{

    struct args \*dim = arg;

    int i = dim -> row;

    for (int j = 0; j < colsB; j++)

    {

        matrix\_C[i][j] = 0;

        for (int k = 0; k < colsA; k++)

        {

            matrix\_C[i][j] += matrix\_A[i][k] \* matrix\_B[k][j];

        }

    }

    pthread\_exit(NULL);

}

* Method 3: A thread per element is a matrix multiplication algorithm that uses a separate thread for each element of the resulting matrix. This approach can lead to more overhead due to the creation of a large number of threads.
* // Method 3: a thread per element
* void \*elementMultiply(void \*arg)
* {
* struct args \*dim = arg;
* int i = dim -> row;
* int j = dim -> col;
* matrix\_C[i][j] = 0;
* for (int k = 0; k < colsA; k++)
* {
* matrix\_C[i][j] += matrix\_A[i][k] \* matrix\_B[k][j];
* }
* pthread\_exit(NULL);
* }

3) Main Function

- Prepare Input and output files

This code initializes file names for input and output files. It creates default file names and if four command-line arguments are provided, it modifies the file names based on the third argument. It creates new file names with extensions "\_per\_matrix.txt", "\_per\_row.txt", and "\_per\_element.txt" for the third element and adds a ".txt" extension to the file names of the first two elements. Finally, it updates the variable names to the new file names.

// Init file names

    char\* fileMatA = "a.txt";

    char\* fileMatB = "b.txt";

    char\* fileMatC\_PM = "c\_per\_matrix.txt";

    char\* fileMatC\_PR = "c\_per\_row.txt";

    char\* fileMatC\_PE = "c\_per\_element.txt";

    char fileNames[5][50]; // 2D array to store the resulting file names

    char \*temp[4];

    struct timeval stop, start;

    // Store args in temp array

    for(int i=0; i<4; i++)

        temp[i] = argv[i];

// If entered args = 4 change the default file names

    if(argc == 4)

    {

        for (int i = 0; i < 3; i++)

    {

        // create filenames with extension \_per\_matrix.txt, \_per\_row.txt, and \_per\_element.txt for element 3

        if (i == 2)

        {

            strcpy(fileNames[i], temp[i+1]);

            strcpy(fileNames[i+1], temp[i+1]);

            strcpy(fileNames[i+2], temp[i+1]);

            strcat(fileNames[i], "\_per\_matrix.txt");

            strcat(fileNames[i+1], "\_per\_row.txt");

            strcat(fileNames[i+2], "\_per\_element.txt");

        }

        else

        {

            // create filename with extension .txt for elements 1 and 2

            strcpy(fileNames[i], temp[i+1]);

            strcat(fileNames[i], ".txt");

        }

    }

    fileMatA = fileNames[0];

    fileMatB = fileNames[1];

    fileMatC\_PM = fileNames[2];

    fileMatC\_PR = fileNames[3];

    fileMatC\_PE = fileNames[4];

    }

* Read Matrix A and B

This code opens a file containing a matrix, reads its dimensions and numbers, allocates memory for the matrix, reads the numbers and stores them in the matrix, and then prints the matrix.

The provided approach for matrix A is done also to matrix B just with different file names and dimensions.

// Open file to read matrix

    FILE \*fa = fopen(fileMatA, "r");

    if (fa == NULL)

    {

        printf("Error opening file %s.\n", fileMatA);

        exit(1);

    }

    // Read row and col from file

    fscanf(fa, "row=%d col=%d\n", &rowsA, &colsA);

    // Allocate memory for the matrixA

    matrix\_A = (int\*\*) malloc(rowsA \* sizeof(int\*));

    if(matrix\_A == NULL)

    {

        printf("Memory Allocation FAILED!");

        return -1;

    }

    for (int i = 0; i < rowsA; i++) {

        matrix\_A[i] = (int\*) malloc(colsA \* sizeof(int));

        if(matrix\_A[i] == NULL)

        {

            printf("Memory Allocation FAILED!");

            free(matrix\_A);

            return -1;

        }

    }

    // Read remaining numbers from file and store in matrix

    for (int i = 0; i < rowsA; i++) {

        for (int j = 0; j < colsA; j++) {

            fscanf(fa, "%d", &matrix\_A[i][j]);

        }

    }

    fclose(fa);

    // Print the matrix

    printf("Matrix A:\n");

    printMatrix(matrix\_A, rowsA, colsA);

* Prepare Matrix C for output result

This code segment is responsible for allocating memory for the resulting matrix C that will store the result of the multiplication of matrices A and B. It checks if the dimensions of A and B matrices are suitable for multiplication, and then it allocates memory for matrix C. It initializes all the elements of the matrix C to zero. If the memory allocation fails, the function returns an error.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Matrix C \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    // Check dimension for suitable matrix multiplication

    if(colsA == rowsB) {

        rowsC = rowsA;

        colsC = colsB;

    }

    else{

        rowsC = 0;

        colsC = 0;

        printf("Dimension Error\n");

        free(matrix\_A);

        free(matrix\_B);

        return -2;

    }

    // Allocate memory for the matrixC

    matrix\_C = (int\*\*) malloc(rowsC \* sizeof(int\*));

    if(matrix\_C == NULL){

        printf("Memory Allocation FAILED!");

        free(matrix\_A);

        free(matrix\_B);

        return -1;

    }

    for (int i = 0; i < rowsC; i++){

        matrix\_C[i] = (int\*) malloc(colsC \* sizeof(int));

        if(matrix\_C[i] == NULL){

            printf("Memory Allocation FAILED!");

            free(matrix\_A);

            free(matrix\_B);

            free(matrix\_C);

            return -1;

        }

        for (int j = 0; j < colsC; j++) {

            matrix\_C[i][j] = 0; // set all elements to zero

        }

    }

* Matrix multiplication per matrix (Normal)

This section of code implements matrix multiplication using a single thread in a normal method. The result matrix C is printed, and the matrix is written to a new text file along with the execution time and number of threads.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Multiplication per matrix \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    // Normal method

    pthread\_t tid[rowsC][colsC];

    numOfThreads = 0;

    printf("\nMatrix multiplication:\n");

    gettimeofday(&start, NULL); //start checking time

    //your code goes here

    pthread\_create(&tid[0][0], NULL, normalMultiply, NULL);

    numOfThreads++;

    // Join all threads

    pthread\_join(tid[0][0], NULL);

    gettimeofday(&stop, NULL); //end checking time

    printf("Seconds taken %lu\n", stop.tv\_sec - start.tv\_sec);

    printf("Microseconds taken: %lu\n", stop.tv\_usec - start.tv\_usec);

    // Print the result matrix

    printf("\nMatrix C (Normal):\n");

    printMatrix(matrix\_C, rowsC, colsC);

    // Write matrixC to new text file

    FILE \*fnorm = fopen(fileMatC\_PM, "w");

    if (fnorm == NULL) {

        printf("Error opening file %s.\n", fileMatC\_PM);

        exit(1);

    }

    // Write matrixC to file

    fprintf(fnorm, "Method: A thread per matrix\n");

    fprintf(fnorm, "row=%d col=%d\n", rowsC, colsC);

    for (int i = 0; i < rowsC; i++) {

        for (int j = 0; j < colsC; j++){

            fprintf(fnorm, "%d ", matrix\_C[i][j]);

        }

        fprintf(fnorm, "\n");

    }

    fprintf(fnorm, "\nMicroseconds taken: %lu\n", stop.tv\_usec - start.tv\_usec);

    fprintf(fnorm, "Number of threads: %d\n", numOfThreads);

    fclose(fnorm);

* Matrix multiplication per row

The code creates a struct args that contains the row number of matrix A that each thread will use to calculate the corresponding rows of matrix C. It then creates a thread for each row of matrix A and passes the corresponding struct args as an argument to each thread. The rowMultiply function is the function that each thread executes, which calculates the corresponding rows of matrix C.

After all threads have finished executing, the code prints the time taken for the matrix multiplication and the resulting matrix C. It then writes the resulting matrix C to a file, along with the time taken and the number of threads used. The file name is specified by the fileMatC\_PR variable.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Multiplication per row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    struct args \*dims;

    numOfThreads = 0;

    // Multiply matrices using row-wise method

    printf("\nRow-wise multiplication:\n");

    gettimeofday(&start, NULL); //start checking time

    for (int i = 0; i < rowsA; i++)

    {

        dims = malloc(sizeof(struct args));

        dims->row = i;

        pthread\_create(&tid[i][0], NULL, rowMultiply, dims);

        numOfThreads++;

    }

    // Join all threads

    for (int i = 0; i < rowsA; i++) {

        pthread\_join(tid[i][0], NULL);

    }

    free(dims);

    gettimeofday(&stop, NULL); //end checking time

    printf("Seconds taken %lu\n", stop.tv\_sec - start.tv\_sec);

    printf("Microseconds taken: %lu\n", stop.tv\_usec - start.tv\_usec);

    // Print the result matrix

    printf("\nMatrix C (Row-wise):\n");

    printMatrix(matrix\_C, rowsC, colsC);

    // Write matrixC to new text file

    FILE \*frow = fopen(fileMatC\_PR, "w");

    if (frow == NULL)

    {

        printf("Error opening file %s.\n", fileMatC\_PR);

        exit(1);

    }

    // Write matrixC to file

    fprintf(frow, "Method: A thread per row\n");

    fprintf(frow, "row=%d col=%d\n", rowsC, colsC);

    for (int i = 0; i < rowsC; i++)

    {

        for (int j = 0; j < colsC; j++)

        {

            fprintf(frow, "%d ", matrix\_C[i][j]);

        }

        fprintf(frow, "\n");

    }

    fprintf(frow, "\nMicroseconds taken: %lu\n", stop.tv\_usec - start.tv\_usec);

    fprintf(frow, "Number of threads: %d\n", numOfThreads);

    fclose(frow);

* Matrix multiplication per element

This section of the code is for element-by-element multiplication. It creates a thread for each element in the resulting matrix C, and each thread is responsible for computing the value of that element. The code also prints out the time taken to perform the multiplication and the resulting matrix C. It writes the matrix C to a new text file and includes information about the method used, the size of the matrix, the time taken, and the number of threads used. Finally, the code frees the matrices.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Multiplication per element \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

    // Multiply matrices using element-by-element method

    printf("\nelement-by-element multiplication:\n");

    numOfThreads = 0;

    gettimeofday(&start, NULL); //start checking time

    //your code goes here

for (int i = 0; i < rowsC; i++)

    {

        for (int j = 0; j < colsC; j++)

        {

            dims = malloc(sizeof(struct args));

            dims->row = i;

            dims->col = j;

            pthread\_create(&tid[i][j], NULL, elementMultiply, dims);

            numOfThreads++;

        }

    }

    // Join all threads

    for (int i = 0; i < rowsC; i++)

    {

        for (int j = 0; j < colsC; j++)

        {

            pthread\_join(tid[i][j], NULL);

        }

    }

    free(dims);

    gettimeofday(&stop, NULL); //end checking time

    printf("Seconds taken %lu\n", stop.tv\_sec - start.tv\_sec);

    printf("Microseconds taken: %lu\n", stop.tv\_usec - start.tv\_usec);

    // Print the result matrix

    printf("\nMatrix C (element):\n");

    printMatrix(matrix\_C, rowsC, colsC);

    // Write matrixC to new text file

    FILE \*felem = fopen(fileMatC\_PE, "w");

    if (felem == NULL)

    {

        printf("Error opening file %s.\n", fileMatC\_PE);

        exit(1);

    }

// Write matrixC to file

    fprintf(felem, "Method: A thread per element\n");

    fprintf(felem, "row=%d col=%d\n", rowsC, colsC);

    for (int i = 0; i < rowsC; i++) {

        for (int j = 0; j < colsC; j++) {

            fprintf(felem, "%d ", matrix\_C[i][j]);

        }

        fprintf(felem, "\n");

    }

    fprintf(felem, "\nMicroseconds taken: %lu\n", stop.tv\_usec - start.tv\_usec);

    fprintf(felem, "Number of threads: %d\n", numOfThreads);

    fclose(felem);

    freeMatrices(matrix\_A, matrix\_B, matrix\_C);

// Helping functions implementation

void freeMatrices(int\*\* matrix\_A, int\*\* matrix\_B, int\*\* matrix\_C)

{

    // Free memory for matrixA

    for (int i = 0; i < rowsA; i++) {

        free(matrix\_A[i]);

    }

    free(matrix\_A);

    // Free memory for matrixB

    for (int i = 0; i < rowsB; i++) {

        free(matrix\_B[i]);

    }

    free(matrix\_B);

    // Free memory for matrixC

    for (int i = 0; i < rowsC; i++) {

        free(matrix\_C[i]);

    }

    free(matrix\_C);

}

void printMatrix(int \*\*matrix, int rows, int cols)

{

    for (int i = 0; i < rows; i++){

        for (int j = 0; j < cols; j++){

            printf("%d ", matrix[i][j]);

        }

        printf("\n");

    }

}

**Sample runs:**

Using make command for Makefile:

Text

Description automatically generated

When running (./matMultp) in terminal the files:

Graphical user interface, text, application

Description automatically generated

Files contents:

Calendar

Description automatically generated A picture containing text

Description automatically generated

Text

Description automatically generated Text

Description automatically generated

Text

Description automatically generated

When running (./matMultp test1a test1b res1) in terminal:

A screenshot of a computer

Description automatically generated with medium confidence Text

Description automatically generated

Text

Description automatically generated with low confidence

Text

Description automatically generated

Text

Description automatically generated