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.

2) 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.

```
3) // Method 1: normal multiplication
4) void *normalMultiply()
5) {
6)     for(int i=0; i<rowsA; i++)
7)     {
8)         for (int j = 0; j < colsB; j++)
9)         {
10)            matrix_C[i][j] = 0;
11)            for (int k = 0; k < colsA; k++)
12)            {
13)                matrix_C[i][j] += matrix_A[i][k] * matrix_B[k][j];
14)            }
15)        }
16)    }
17)    pthread_exit(NULL);
18)}
```

- 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
```

```c
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);
}
```

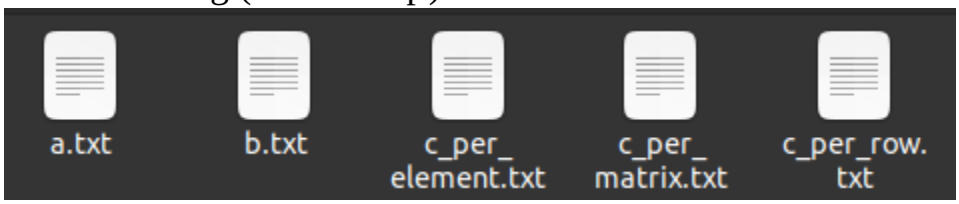
```
void printMatrix(int **matrix, int rows, int cols)
{
    for (int i = 0; i < rows; i++){
        for (int j = 0; j < cols; j++){
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
```

Sample runs:

Using make command for Makefile:

```
acer@acer-Aspire-A315-53:~/TERM 8/OS/Lab2 matMult$ ls
main.c  Makefile  test1  test1a.txt  test1b.txt  test2  test2a.txt  test2b.txt
  test3  test3a.txt  test3b.txt
acer@acer-Aspire-A315-53:~/TERM 8/OS/Lab2 matMult$ make
gcc -pthread main.c -o matMultp
acer@acer-Aspire-A315-53:~/TERM 8/OS/Lab2 matMult$ ls
main.c  Makefile  matMultp  test1  test1a.txt  test1b.txt  test2  test2a.txt
test2b.txt  test3  test3a.txt  test3b.txt
```

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



Files contents:

a.txt

```
row=5 col=5
1   2   3   4   5
6   7   8   9   10
11  12  13  14  15
16  17  18  19  20
21  22  23  24  25
```

b.txt

```
row=5 col=4
1   2   3   4
5   6   7   8
9   10  11  12
13  14  15  16
17  18  19  20
```

c_per_element.txt

```
Method: A thread per element
row=5 col=4
175 190 205 220
400 440 480 520
625 690 755 820
850 940 1030 1120
1075 1190 1305 1420

Microseconds taken: 7742
Number of threads: 20
```

c_per_matrix.txt

```
Method: A thread per matrix
row=5 col=4
175 190 205 220
400 440 480 520
625 690 755 820
850 940 1030 1120
1075 1190 1305 1420

Microseconds taken: 737
Number of threads: 1
```

c_per_row.txt

```
Method: A thread per row
row=5 col=4
175 190 205 220
400 440 480 520
625 690 755 820
850 940 1030 1120
1075 1190 1305 1420

Microseconds taken: 1080
Number of threads: 5
```

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

test1a.txt

```
row=10 col=5
1   2   3   4   5
6   7   8   9   10
11  12  13  14  15
16  17  18  19  20
21  22  23  24  25
26  27  28  29  30
31  32  33  34  35
36  37  38  39  40
41  42  43  44  45
46  47  48  49  50
```

test1b.txt

```
row=5 col=10
1   2   3   4   5   6   7   8   9   10
11  12  13  14  15  16  17  18  19  20
21  22  23  24  25  26  27  28  29  30
31  32  33  34  35  36  37  38  39  40
41  42  43  44  45  46  47  48  49  50
```

res1_per_element.txt

```
Method: A thread per element
row=10 col=10
415 430 445 460 475 490 505 520 535 550
940 980 1020 1060 1100 1140 1180 1220 1260 1300
1465 1530 1595 1660 1725 1790 1855 1920 1985 2050
1990 2080 2170 2260 2350 2440 2530 2620 2710 2800
2515 2630 2745 2860 2975 3090 3205 3320 3435 3550
3040 3180 3320 3460 3600 3740 3880 4020 4160 4300
3565 3730 3895 4060 4225 4390 4555 4720 4885 5050
4090 4280 4470 4660 4850 5040 5230 5420 5610 5800
4615 4830 5045 5260 5475 5690 5905 6120 6335 6550
5140 5380 5620 5860 6100 6340 6580 6820 7060 7300

Microseconds taken: 4687
Number of threads: 100
```

res1_per_matrix.txt

```
Method: A thread per matrix
row=10 col=10
415 430 445 460 475 490 505 520 535 550
940 980 1020 1060 1100 1140 1180 1220 1260 1300
1465 1530 1595 1660 1725 1790 1855 1920 1985 2050
1990 2080 2170 2260 2350 2440 2530 2620 2710 2800
2515 2630 2745 2860 2975 3090 3205 3320 3435 3550
3040 3180 3320 3460 3600 3740 3880 4020 4160 4300
3565 3730 3895 4060 4225 4390 4555 4720 4885 5050
4090 4280 4470 4660 4850 5040 5230 5420 5610 5800
4615 4830 5045 5260 5475 5690 5905 6120 6335 6550
5140 5380 5620 5860 6100 6340 6580 6820 7060 7300

Microseconds taken: 432
Number of threads: 1
```

res1_per_row.txt

```
Method: A thread per row
row=10 col=10
415 430 445 460 475 490 505 520 535 550
940 980 1020 1060 1100 1140 1180 1220 1260 1300
1465 1530 1595 1660 1725 1790 1855 1920 1985 2050
1990 2080 2170 2260 2350 2440 2530 2620 2710 2800
2515 2630 2745 2860 2975 3090 3205 3320 3435 3550
3040 3180 3320 3460 3600 3740 3880 4020 4160 4300
3565 3730 3895 4060 4225 4390 4555 4720 4885 5050
4090 4280 4470 4660 4850 5040 5230 5420 5610 5800
4615 4830 5045 5260 5475 5690 5905 6120 6335 6550
5140 5380 5620 5860 6100 6340 6580 6820 7060 7300

Microseconds taken: 604
Number of threads: 10
```