Faculty of Engineering & Technology
Electrical & Computer Engineering Department

ENCS3390: Operating System Concepts
First Semester, 2023/2024

Matrix Multiplication Performance Analysis Report

Student Name : Asmaa Abed Al-Rahman Fares

Student Id : 1210084

Instructor : Abed Al-Salam Sayyad

Section.no : 2

Date : 28/11/2023

Table Contents

1. Task Abstract :

The purpose of this task is to analyze the performance of different approaches to matrix multiplication using processes and threads. The implemented program utilizes a naive approach, a process-based approach, and two threaded approaches (joinable threads and detached threads) for matrix multiplication.

2. Methods Used:

A. Naive Approach (Single Thread):

The naive approach involves a straightforward nested loop implementation without utilizing processes or threads. The throughput time for this approach is measured over multiple iterations.

- the implementation :

```
// Naive approach (no child processes or threads)
struct timeval start, end;

gettimeofday(&start, NULL);

// matrix multiplication
for (int i = 0; i < SIZE; ++i)
{
    for (int j = 0; j < SIZE; ++j)
    {
        result[i][j] = 0;
        for (int k = 0; k < SIZE; ++k)
        {
            result[i][j] += Matrix1[i][k] * Matrix2[k][j];
        }
    }
}

gettimeofday(&end, NULL);
```

B. Process-Based Approach:

The process-based approach involves creating multiple child processes to perform matrix multiplication in parallel. Each child process computes a portion of the result, and the parent process combines these partial results.

- the implementation :

```
// Calculate rows per child and thread
int rowsPerChild = SIZE / numProcesses;
int rowsPerThread = SIZE / numThreads;
// Pipe array for IPC
int pipes[numProcesses][2];
// Create a buffer to hold the result for each child
int (*childResults)[SIZE] = malloc(sizeof(int[SIZE][SIZE]));

if (!childResults)
{
    perror("Memory allocation failed");
    exit(EXIT_FAILURE);
}


// Get the start time for processes
gettimeofday(&start, NULL);

// Create child processes and pipes for process-based approach
for (int i = 0; i < numProcesses; ++i)
{
    if (pipe(pipes[i]) == -1)
    {
        perror("Pipe creation failed");
        exit(EXIT_FAILURE);
    }

    pid_t childPid = fork();

    if (childPid == -1)
    {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    }
```

```
        if (childPid == 0) // Child process
        {
            close(pipes[i][0]); // Close unused read end of the pipe
            multiplyRows(i * rowsPerChild, (i + 1) * rowsPerChild, DynamicMatrix1, DynamicMatrix2, childResults);
            write(pipes[i][1], childResults, sizeof(int[SIZE][SIZE])); // Write the result to the pipe
            close(pipes[i][1]);                                        // Close the write end of the pipe

            exit(EXIT_SUCCESS);
        }
        else // Parent process
        {
            close(pipes[i][1]); // Close unused write end of the pipe
        }
    }

    // Wait for all child processes to complete
    for (int i = 0; i < numProcesses; ++i)
    {
        wait(NULL);
    }

    // Read results from pipes and accumulate them in the result matrix for process-based approach
    for (int i = 0; i < numProcesses; ++i)
    {
        close(pipes[i][1]); // Close write end of the pipe in the parent

        // Read the result from the pipe and accumulate it in the result matrix
        read(pipes[i][0], childResults, sizeof(int[SIZE][SIZE]));
        close(pipes[i][0]); // Close read end of the pipe

        // Copy the child result into the main result matrix
        for (int row = 0; row < rowsPerChild; ++row)
        {
            memcpy(DynamicResult[i * rowsPerChild + row], childResults[row], sizeof(int[SIZE]));
        }
    }
```

C. Joinable Threads Approach:

The joinable threads approach utilizes pthreads to create multiple threads that perform matrix multiplication concurrently. Each thread works on a specific range of rows in the matrices.

- the implementation :

```
// Thread routine for joinable threads
void *multiplyRowsJoinable(void *args)
{
    struct ThreadArgs *threadArgs = (struct ThreadArgs *)args;
    multiplyRows(threadArgs->startRow, threadArgs->endRow, Matrix1, Matrix2, result);
    pthread_exit(NULL);
}
```

```
// Create thread array and thread argument array for threaded approaches
pthread_t threads[MAX_THREADS];
struct ThreadArgs threadArgs[MAX_THREADS];

// Get the start time for joinable threads
gettimeofday(&start, NULL);

// Create and run threads with joinable for threaded approach
for (int i = 0; i < numThreads; ++i)
{
    threadArgs[i].startRow = i * rowsPerThread;
    threadArgs[i].endRow = (i + 1) * rowsPerThread;

    if (pthread_create(&threads[i], NULL, multiplyRowsJoinable, (void *)&threadArgs[i]) != 0)
    {
        fprintf(stderr, "Error: Failed to create thread.\n");
        exit(EXIT_FAILURE);
    }
}

// Join threads for threaded approach
for (int i = 0; i < numThreads; ++i)
{
    if (pthread_join(threads[i], NULL) != 0)
    {
        fprintf(stderr, "Error: Failed to join thread.\n");
        exit(EXIT_FAILURE);
    }
}

// Get the end time for joinable threads
gettimeofday(&end, NULL);
```

D. Detached Threads Approach:

The detached threads approach also uses pthreads, but in this case, threads are detached after creation. The throughput time for this approach is measured over multiple iterations

- the implementation :

```
pthread_t detachedThreads[MAX_THREADS];
// Get the start time for detached threads
gettimeofday(&start, NULL);


for (int i = 0; i < numThreads; ++i)
{
    threadArgs[i].startRow = i * rowsPerThread;
    threadArgs[i].endRow = (i + 1) * rowsPerThread;

    if (pthread_create(&detachedThreads[i], NULL, multiplyRows, (void *)&threadArgs[i]) != 0)
    {
        fprintf(stderr, "Error: Failed to create thread.\n");
        exit(EXIT_FAILURE);
    }
}
```

3. Results in a Table:

Approach	Time Taken (microseconds)
Naive (Single Thread)	7273
Process-Based (2 processes)	2697
Process-Based (4 processes)	5474
Process-Based (8 processes)	5466
Joinable Threads (2 threads)	2245
Joinable Threads (4 threads)	4762
Joinable Threads (8 threads)	4854
Detached Threads (2 threads)	42
Detached Threads (4 threads)	71
Detached Threads (8 threads)	148

Note that for detach this is the time to create it because it continues running in the background

Approach	Throughput Taken
Naive (Single Thread)	0.00013149
Process-Based (2 processes)	0.00037078
Process-Based (4 processes)	0.00018268
Process-Based (8 processes)	0.00017718
Joinable Threads (2 threads)	0.00044543
Joinable Threads (4 threads)	0.00021
Joinable Threads (8 threads)	0.00020602
Detached Threads (2 threads)	0.02380952
Detached Threads (4 threads)	0.01408451
Detached Threads (8 threads)	0.00675676

4. What We Learned:

- Depending on our system, different methods and numbers of processes/threads might be faster.
- More isn't always better. Sometimes too many processes or threads can slow things down.
- The impact of parallelization on matrix multiplication performance.
- The trade-offs between process-based and threaded approaches.
- The importance of synchronization in threaded implementations.

5. Conclusion:

What I concluded from my obtained output is that using 4 number of processes and threads is the best choice because it gave me less execution time and throughput for the task , and the benchmarking results provide insights into the efficiency of different parallelization approaches for matrix multiplication. Depending on the system and workload, choosing the right approach can significantly impact overall performance. Further optimizations, such as workload balancing and memory management, could be explored for enhanced efficiency.