Multi-Threaded Matrix Multiplication: Report

Om Dave (CO22BTECH11006)

January 28, 2024

1 Introduction

This report details a program designed to perform matrix multiplication using multi-threading in C++. The program aims to compute the square of a matrix using different strategies: Chunk, Mixed, and Mixed-Chunks methods.

2 Program Design and Implementation

2.1 Initial Setup

- **Matrix Initialization:** The program initializes an $n \times n$ matrix A.
- **Thread Creation:** It creates k threads for parallel computation.

2.2 Computational Methods

2.2.1 Chunk Method

Objective The Chunk Method aims to parallelize matrix operations by dividing the matrix into k equal-sized chunks, assigning each chunk to a different thread for computation. This method is particularly straightforward in its approach, focusing on dividing the workload based on the number of available threads.

Example Consider a 4x4 matrix and 2 threads. The matrix is divided into two chunks: the first chunk comprises rows 1 and 2, and the second chunk comprises rows 3 and 4. Thread 1 is responsible for computing the operations in the first chunk, while thread 2 handles the second chunk.

Advantages

- Simplicity of implementation.
- Suitable for processing contiguous rows.

Disadvantages

- Potential for uneven workload distribution.
- Possible cache inefficiency.

Code Explanation The function `Compute_chunk` is the core of the Chunk Method implementation:

Listing 1: Chunk Method Implementation

```
// Computes square of matrix using chunk method
void* Compute_chunk(void* arg) {
    ComputeArgs* args = (ComputeArgs*)arg;
    int thread_id = args->thread_id;
    int start = args->start;
    int end = min(start + chunk, n);

    // Adjust end for the last thread
    if (thread_id == k) end = n;

    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    // Compute rows assigned to this thread
    for (int i = start; i < end; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                prod_chunk[i][j] += A[i][k] * A[k][j];
            }
        }
    }

    clock_gettime(CLOCK_MONOTONIC, &end_time);
    double time_taken = (end_time.tv_sec - start_time.tv_sec) * 1e9
                        + (end_time.tv_nsec - start_time.tv_nsec);
    exec_time_chunks[thread_id] = time_taken;

    free(args);
    pthread_exit(NULL);
}
```

Code Implementation In the provided code, the Chunk Method is implemented as follows:

- Each thread receives a starting row index and computes a specific chunk of the matrix.
- The range of rows for each thread is determined, ensuring an approximately equal distribution of workload.
- Special handling is included for the last thread to cover any remaining rows.
- Matrix squaring computation is performed using nested loops.
- Execution time for each thread is measured for performance analysis.
- Threads free their resources and exit after completing their task.

2.2.2 Mixed Method

Objective The Mixed Method enhances the distribution of workload among threads by assigning rows in a round-robin fashion, rather than dividing the matrix into chunks. This approach aims to ensure a more balanced workload across threads, especially in cases where the matrix operations vary in complexity across rows.

Example n a 4x4 matrix with 2 threads, the assignment of rows would be as follows: thread 1 computes rows 1 and 3, while thread 2 computes rows 2 and 4. This interleaved assignment ensures that each thread has an equal number of rows to process.

Advantages

- Balanced workload distribution.
- Improved cache efficiency.

Disadvantages

- Increased complexity.
- Potential overhead from each thread accessing all columns for assigned rows.

Code Explanation The function `Compute_mixed` is used for this method.

Listing 2: Chunk Method Implementation

```
void* Compute_mixed(void* arg) {
    ComputeArgs* args = (ComputeArgs*)arg;
    int thread_id = args->thread_id;
    int start = args->start;
    int end = n;

    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    for (int i = start; i < end; i += k) {
        //computing row i of product matrix
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                prod_mixed[i][j] += A[i][k] * A[k][j];
            }
        }
    }

    clock_gettime(CLOCK_MONOTONIC, &end_time);
    double time_taken = (end_time.tv_sec - start_time.tv_sec) * 1e9 + (end
    exec_time_mixed[thread_id] = time_taken;

    free(args);
    pthread_exit(NULL);
}
```

Code Implementation In the Mixed Method:

- Each thread starts computing from a unique row, defined by its thread ID.
- Threads iterate over the matrix with a step size equal to the total number of threads.
- This ensures that each thread processes an evenly distributed set of rows.
- The nested loops calculate the square of the matrix for the rows assigned to each thread.

2.2.3 Mixed-Chunks Method

Objective The mixed-chunk method seeks to balance the workload among threads while maintaining a degree of spatial locality to optimize cache usage. It does this by dividing the matrix into larger chunks and then distributing rows within each chunk in a round-robin fashion among a subset of threads. This way, each thread still works on rows that are relatively close in memory, but the overall workload is more evenly distributed than in the pure chunk method

Example Divide the Matrix: Divide the C matrix into blocks of rows, where each block contains b rows. The size of b is determined based on the total number of rows N and the number of threads K, but unlike the Chunk method, b is chosen to be smaller than $\frac{N}{K}$ to ensure more frequent distribution among threads. For example, if $N = 1000$ and $K = 10$, instead of a chunk size of 100 (as in the Chunk method), we might choose a chunk size of 20.

Advantages

- Effective load balancing.
- Potentially optimal cache performance.

Disadvantages

- Complex implementation.
- Requires precise chunk size tuning.

Listing 3: Mixed-Chunks Method Implementation

```
void* Compute_mixed_chunks(void* arg) {
    ComputeArgs* args = (ComputeArgs*)arg;
    int thread_id = args->thread_id;
    int start = args->start;
    int end = n;

    struct timespec start_time, end_time;
    clock_gettime(CLOCK_MONOTONIC, &start_time);

    for (int p = start; p < end; p += k * chunk_for_mixed_chunks) {
```

```
        for (int i = p; i < min(p + chunk_for_mixed_chunks, n); i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    prod_mixed_chunks[i][j] += A[i][k] * A[k][j];
                }
            }
        }
    }

    clock_gettime(CLOCK_MONOTONIC, &end_time);
    double time_taken = (end_time.tv_sec - start_time.tv_sec) * 1e9 + (end
    exec_time_mixed_chunks[thread_id] = time_taken;

    free(args);
    pthread_exit(NULL);
}
```

Code Implementation The Mixed-Chunks Method is implemented as follows:

- **Chunk Size Calculation:** The size of each chunk (or block) in the mixed-chunks method is determined by the formula: max(((n / k) / k) * 2, 2), where n is the dimension of the square matrix and k is the number of threads. This formula aims to balance the workload among threads by considering the total size of the matrix and the number of threads, adjusting the chunk size accordingly to ensure that each thread has a substantial amount of work. The use of max ensures that the chunk size is at least 2, preventing scenarios where the chunk size could be too small to be practical for parallel processing.
- **Thread Assignment:** Each thread is assigned a starting position based on its thread ID.
- **Matrix Computation:** Nested four-loop structure is used for computation. Each thread works on distinct chunks spread across the matrix, computing blocks of rows within these chunks.
- **Performance Measurement:** Execution time for each thread's computation is measured for analysis.

3 Performance Analysis

3.1 Time vs Size (N)

The performance of the matrix multiplication algorithms was evaluated by varying the size of the input matrix N, with sizes ranging from 16×16 to 2048×2048 matrices. The time taken to compute the square of the matrix using Chunk, Mixed, and Mixed-Chunk methods was recorded. The number of threads (K) was kept constant at 8 for all experiments.

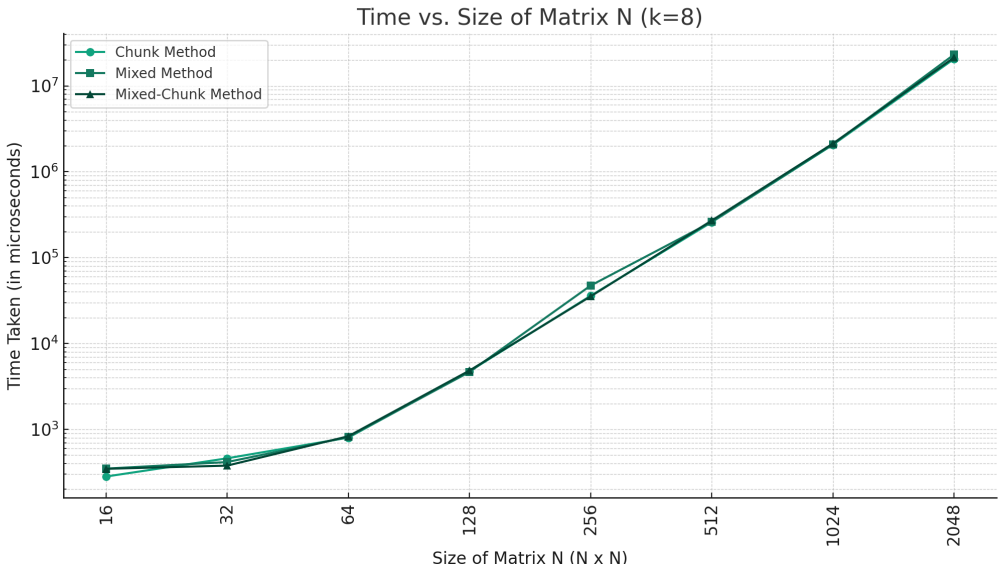

Figure 1: Time taken to compute the square of the matrix vs. the size of the matrix N.

3.1.1 Observations

From the plot depicted in Figure 2, we observe the following trends:

- The time taken for computation by all methods increases with the size of the matrix N. This behavior aligns with the expected computational complexity of matrix multiplication, which is $O(N^3)$ for the straightforward algorithm.
- The Chunk method shows a relatively uniform increase in computation time as the size of the matrix grows, suggesting possible inefficiencies like cache misses or workload imbalance for larger matrix sizes.
- The Mixed method reveals a variable rate of increase in computation time, indicating potential overhead less efficient cache utilization as the matrix size increases.
- The Mixed-Chunk method initially demonstrates an advantage in performance over the other methods, suggesting an effective balance of workload distribution and cache utilization. However, as the matrix size increases, the distinction in performance diminishes.

3.1.2 Analysis and Interpretation

The increasing computation time with matrix size is anticipated due to the nature of matrix multiplication. The rate of increase, however, varies among the methods and provides insight into their relative efficiency concerning workload distribution and cache usage. Specifically:

- The Chunk method's performance suggests potential cache inefficiencies or imbalance in workload distribution as the matrix size scales.
- The Mixed method's performance curve suggests that while it may offer a more evenly distributed approach, it might also suffer from overhead due to less efficient cache usage for larger matrices.
- The Mixed-Chunk method's initial performance improvement suggests that the strategy of balancing workload distribution while minimizing context switching and considering cache utilization is effective. However, for larger matrices, the benefits may be constrained by hardware and algorithmic scalability limits.

In conclusion, while all methods exhibit longer computation times with larger matrices, the Mixed-Chunk method initially provides a performance benefit by combining the advantages of both Chunk and Mixed methods. Nonetheless, as matrix sizes become larger, the advantages of different methods converge, potentially hitting scalability limits.

3.2 Time vs Number of Threads (K)

The graph visualizes the time taken for matrix squaring using three different parallel processing methods as the number of threads k increases.

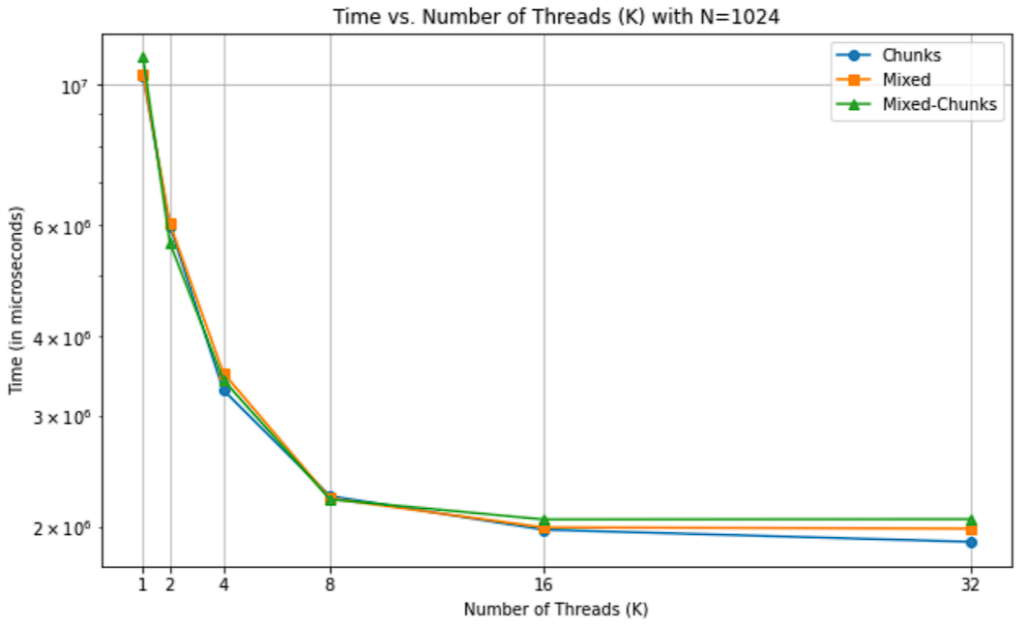

Figure 2: Time taken to compute the square of the matrix vs. the size of the matrix N.

The key observations from the graph are:

3.2.1 Observations

- For all methods, there is a significant decrease in computation time as the number of threads increases from 1 to 16.
- Beyond a certain number of threads (around 16), the decrease in computation time becomes less pronounced, and for some methods, the time remains relatively constant or even slightly increases.

3.2.2 Analysis and Interpretation

1. *Initial Decrease in Computation Time*: The initial decrease in computation time with an increase in the number of threads is expected and can be attributed to the parallel processing capabilities of the hardware. As more threads are utilized, the workload is distributed more evenly across the CPU cores, leading to faster computation times.
2. *Diminishing Returns Beyond a Certain Point*: The observation that the decrease in computation time becomes less significant or halts beyond a certain number of threads can be explained by hardware limitations and the overhead associated with managing a large number of threads. Specifically, my laptop is equipped with a hexa-core processor, which, due to hyper-threading technology, supports up to 12 threads simultaneously. Once the number of threads exceeds this limit, additional threads do not contribute to performance improvements and may even lead to increased overhead due to context switching and resource contention. This is because the physical core count of the CPU limits the actual parallel processing capability.

In conclusion, the performance of parallel processing methods for matrix squaring is significantly influenced by the hardware capabilities and the method's efficiency in distributing the workload among threads. While parallel processing can drastically reduce computation times, there is an optimal number of threads for each hardware setup beyond which the benefits diminish due to hardware limitations and increased overhead.