Assignment 2: OpenMP vs Pthreads
Osama Shamout 201906271
CSC 447 - Parallel Programming for Multicore and Cluster Systems.

Introduction

In this assignment, we explore the performance of matrix multiplication using pthreads and OpenMP, and compare the results to identify the most efficient approach.

To do so, we parallelized the computation of matrix multiplication using both pthreads and OpenMP, and executed a series of experiments with different matrix sizes and number of threads. We measured the execution time and calculated the speedup factor, efficiency, and scalability of each approach. Finally, we compared the performance of the two implementations and drew conclusions based on our results.

The GitHub link for the project is available at:

https://github.com/OsamaShamout/SharedMem

Pthreads

The code is designed to parallelize matrix multiplication using pthreads. It takes input arguments for the dimensions of matrices A and B, and the number of threads to use. The program initializes matrices A and B with all elements set to 1, and matrix C with all elements set to 0.

The program creates multiple threads to perform the matrix multiplication by dividing the computation across threads. Each thread works on a subset of columns of matrix B, with the number of columns per thread calculated based on the number of threads and the number of columns in matrix B. The computation is protected by a mutex to prevent race conditions when updating elements of matrix C.

If the number of threads is greater than the number of columns in matrix B, a warning message is printed suggesting that the maximum number of threads should be equal to the number of columns in matrix B for optimal performance.

After the matrix multiplication is complete, the program measures the execution time and prints the resulting matrix dimensions along with the number of threads and the execution time in seconds.

Pseudocode

```
for each thread t:
    compute start and end column indices for thread t

for each thread t:
    create a new thread that runs the slave function with argument t

for each thread t:
    wait for thread t to finish execution

slave function:
    for each column j assigned to the thread:
        for each row i:
            compute C[i][j] += A[i][k] * B[k][j]
            lock mutex
            update C[i][j]
            unlock mutex
```

OpenMP

The purpose of this code is to perform matrix multiplication using OpenMP for parallelization. It requires inputs for the dimensions of matrices A and B, as well as the number of threads to be used. Both matrices A and B are initialized with every element set to 1, while matrix C is initialized with all elements at 0.

The OpenMP parallel for directive is employed to parallelize the computation, distributing the iterations of the outer loop across the available threads. Inside the loop, each thread operates on a subset of columns of matrix

B, with the number of columns per thread calculated based on the total number of columns and the number of threads used. To guarantee that updates to matrix C are performed atomically, the computation is protected by a reduction clause on the sum variable.

After matrix multiplication is complete, the program measures the execution time and reports the resulting matrix dimensions, along with the number of threads and the time in seconds it took to complete the computation.

No pseudocode is provided as the implementation is very similar to the a written pseudocode.

Implementations Comparison

Comparing the implementations of matrix multiplication using pthreads and OpenMP, we can see that they both aim to parallelize the computation to improve performance. However, they use different approaches to achieve this goal.

In the pthreads implementation, the computation is divided across multiple threads, each working on a subset of columns of matrix B. A mutex is used to protect updates to matrix C and prevent race conditions. This approach requires explicit thread creation, synchronization, and locking and unlocking of mutexes.

On the other hand, the OpenMP implementation uses a parallel for directive to distribute the iterations of the outer loop across multiple threads. Each thread works on a subset of columns of matrix B, and a reduction clause is used to ensure that updates to matrix C are performed atomically. This approach is simpler and more concise than the pthreads implementation, as it requires no explicit thread creation, synchronization, or locking and unlocking of mutexes.

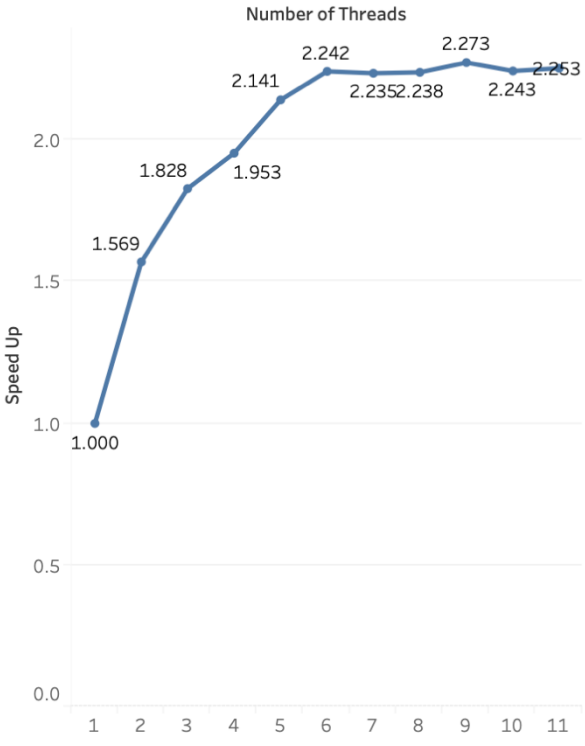
The OpenMP approach offers a more convenient way to parallelize loop-based computations due to its concise and expressive syntax. On the other hand, the pthreads approach requires more low-level details such as thread creation and joining, which can be useful in scenarios that require explicit control over thread synchronization and locking, but can also be more complex and prone to errors.

Performance Comparison

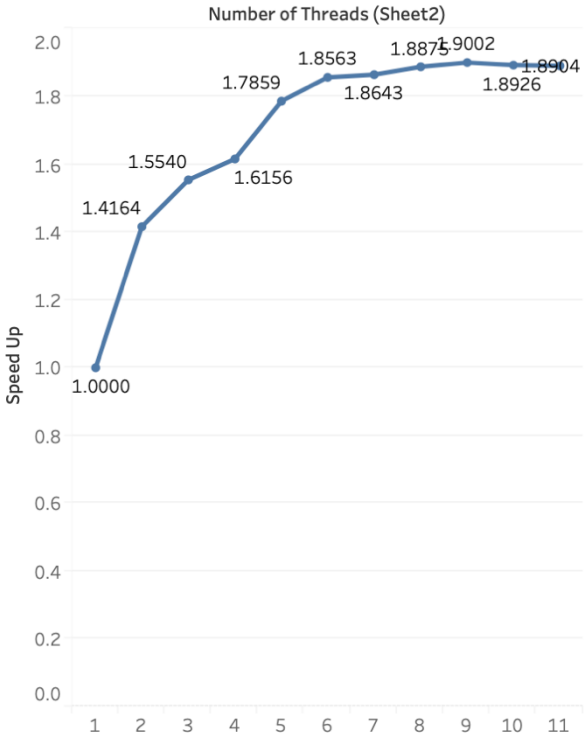
Speed-up Factor = time taken for 1 thread / time taken for N threads

Efficiency = Speedup Factor / N. N = the number of threads.

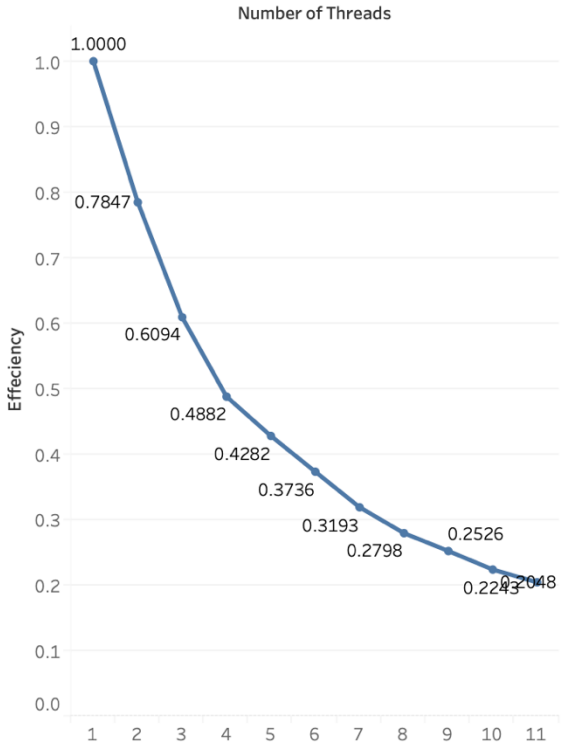
OpenMP Speed-Up Factor for a 3000x3000 Matrix



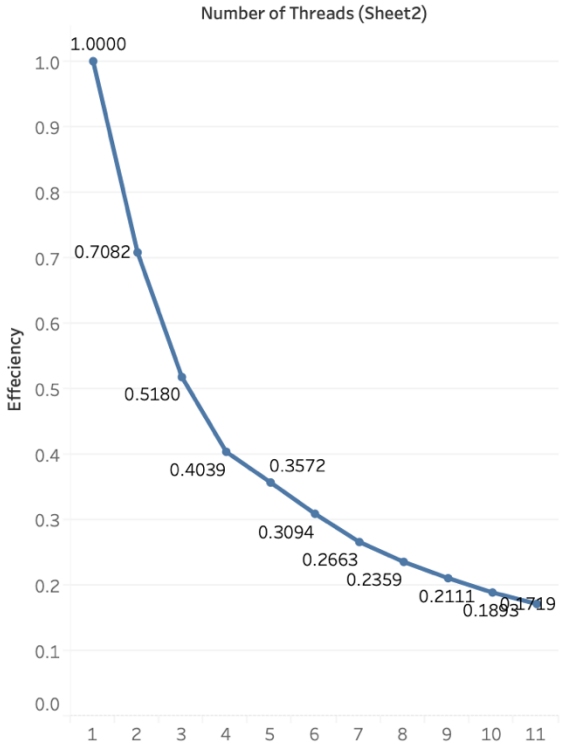
Pthreads Speed-Up Factor for a 3000 x 3000 Matrix



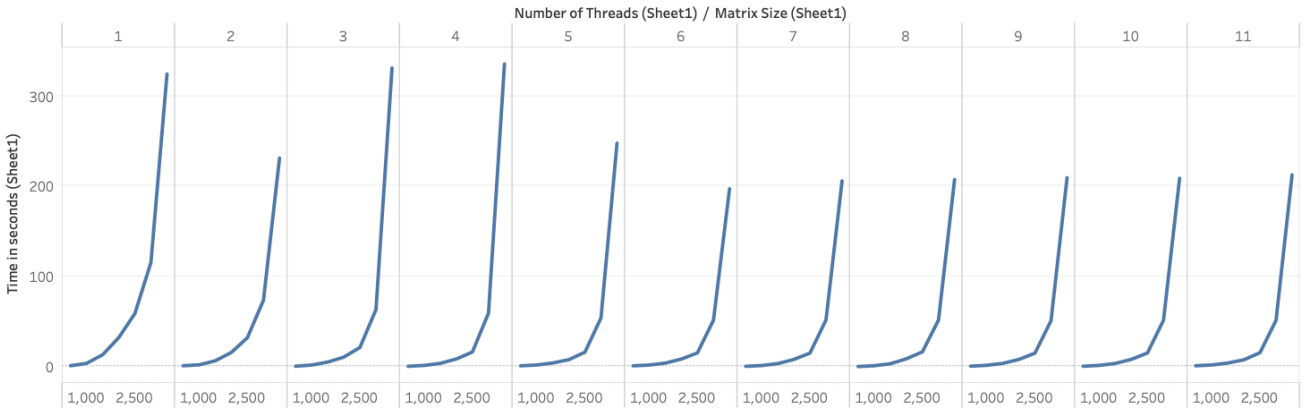
OpenMP Implementation Effeciency



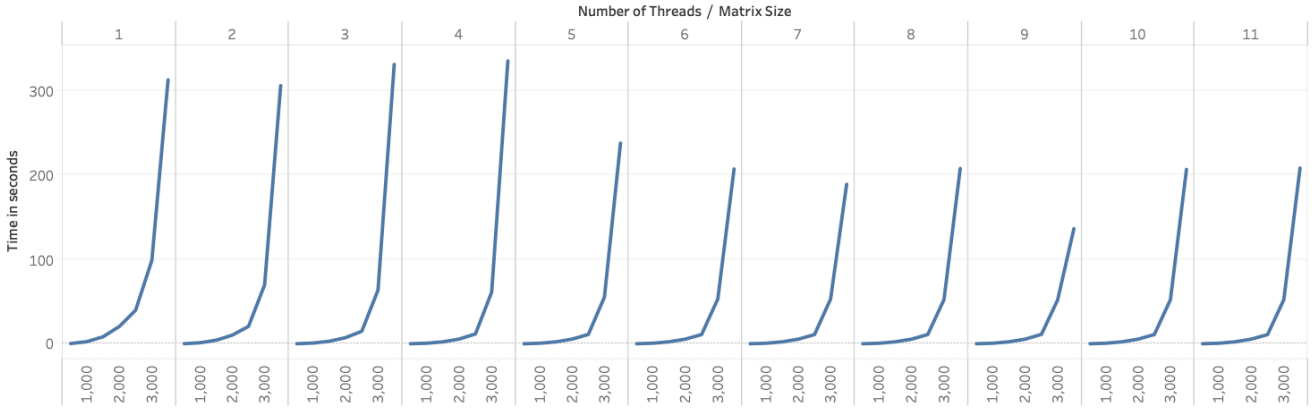
Pthreads Implementation Effeciency



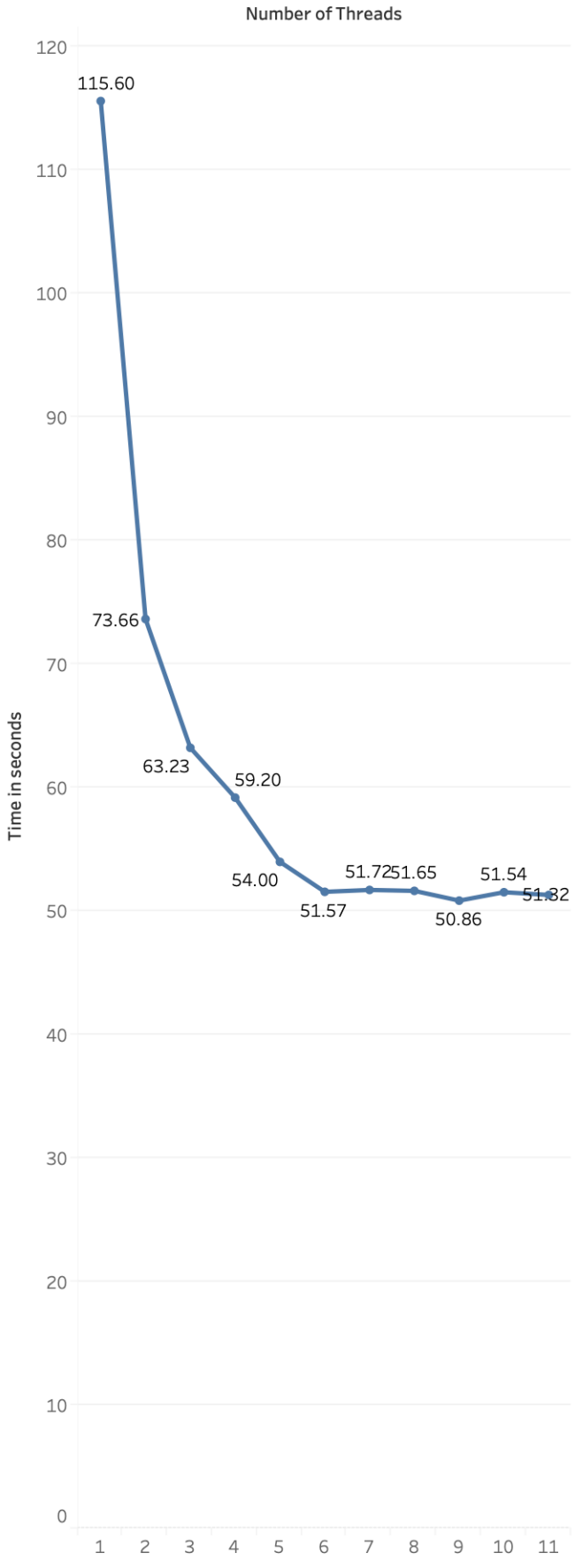
OpenMP Execution Time across Different Matrices



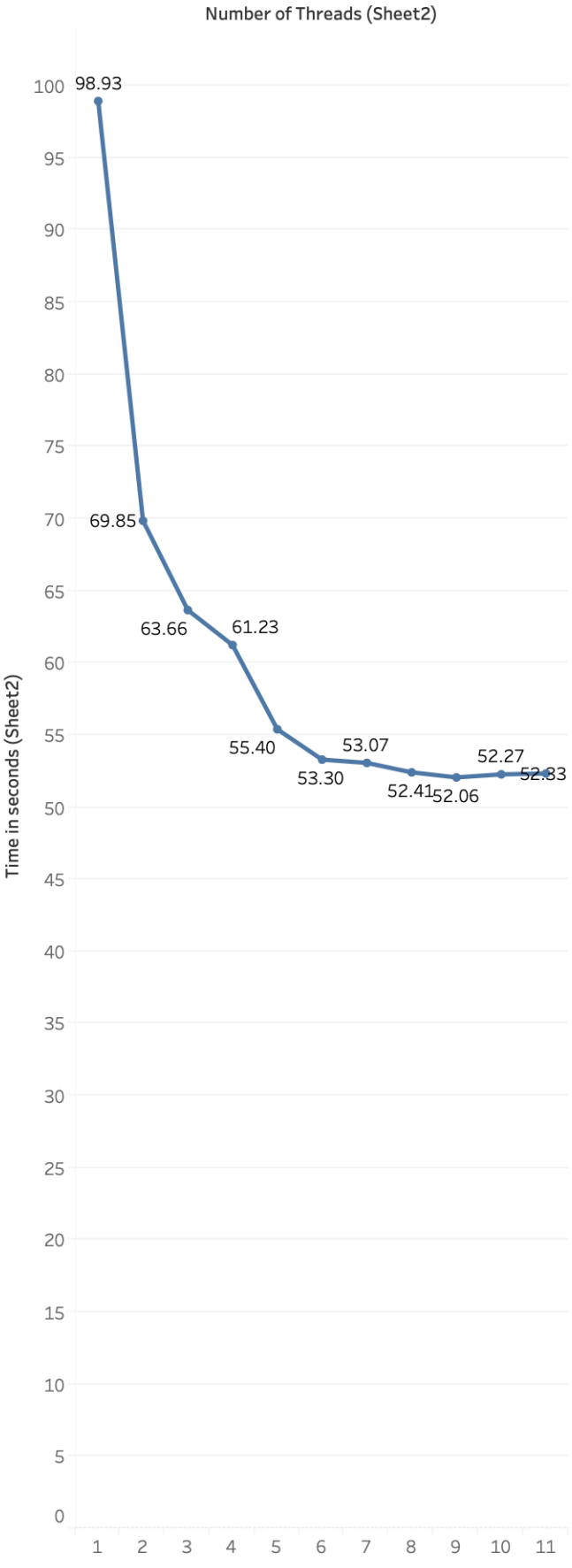
Pthreads Execution Time across Different Matrices



OpenMP Execution Speed for a 3000x3000 Matrix



Pthreads Execution Speed for a 3000 x 3000 Matrix



Based on the provided charts, it is apparent that both implementations achieve a speed-up factor by utilizing parallelization techniques. However, the OpenMP implementation demonstrates a higher speed-up factor and efficiency compared to the Pthreads implementation. This may be attributed to the effectiveness of the pre-written OpenMP library, which may result in less overhead compared to the Pthreads implementation, which involves creating and joining threads. Additionally, the Pthreads implementation performs better with a lower number of threads, while the OpenMP implementation is better in terms of scalability.

However, both implementations exhibit limited scalability or poor scalability, as observed by the non-linear improvement in execution time with increasing matrix size and thread count. The diminishing returns observed from the number of threads equal to 5, is also noteworthy. It is essential to note that threads 9-11 were hypothetical, as the system only has 8 cores (thus 8 threads according to the Macbook M1 architecture). The execution time in these cases was almost identical, indicating the system's limit in terms of parallel processing capability.

Finally, both implementations exhibit limitations in scalability, with the OpenMP implementation providing a higher speed-up factor and efficiency. However, further optimization may be necessary to achieve improved scalability for both implementations (discussed below).

Issues & Future Recommendation

During the implementation phase, I encountered some challenges while attempting to implement the OpenMP parallelization method on the MacBook M1. Specifically, the machine was unable to correctly read the <omp.h> header file and the OMP library, which led to unexpected errors. To ensure that the

results were accurate and reliable, I had to rerun the tests on a virtual machine that would allow us to create an environment that was consistent with our original tests.

To create this environment, I used UTM to create a virtual machine running Kali Linux v2022.2 (with the environment having 4GB RAM and all 8 cores). This enabled me to run both the OpenMP and pthreads implementations with the same set of tools and libraries, ensuring that the results were comparable and consistent. In addition, I also ran a native pthreads implementation on the MacBook M1, and included these results in the analysis.

While the native environment on the MacBook M1 produced significantly faster execution times, it was necessary to use the virtual machine to ensure that we are comparing the same values across all our tests. This allowed the removal of any confounding factors that might skew the results, and ensured that the conclusions were based on accurate and reliable data. Still though, confounding factors such as using the device while in execution or the battery percentage could have played role in the noise that is presented in the data.

In the future, I plan to optimize the implementation of this problem by converting the 2D matrix into a 1D format while also executing it in a more controlled environment to further reduce noise in the data obtained. With this optimization, I expect a reduction in memory that ultimately leads to even faster execution speeds.