Multi-Threaded Matrix Multiplication with Thread Affinity: Report

Om Dave (CO22BTECH11006)

February 23, 2024

1 Introduction

This report investigates the performance of parallelizing the matrix squaring operation, a specialized form of matrix multiplication, using multi-core processors. Specifically, this study examines the impact of thread affinity, which involves assigning threads to specific CPU cores, on the efficiency of two parallel implementation strategies: the chunk method and the mixed method.

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.
- **Thread Affinity:** It binds first bt threads to specific cores.

2.2 System specifications

Table 1: System Specifications

Detail	Specification
CPU(s)	12
Model name	AMD Ryzen 5 5600H
Thread(s) per core:	2
Core(s) per socket:	6

2.3 Computational Methods

2.3.1 Chunk Method

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.

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.

2.3.2 Mixed Method

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.

Example Consider 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.

2.4 Thread affinity

Thread affinity constrains the execution of threads to specific CPU cores, which can potentially enhance performance by improving cache utilization and minimizing the overhead associated with thread migration between cores.

2.4.1 Implementation

Listing 1: Thread Affinity Implementation

```
//Set affinity of thread if it is to be bounded
if (thread_number < bt) {

    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);

    int cpu_id = thread_number % bc;

    CPU_SET(cpu_id, &cpuset);
    bool unsuccessful = pthread_setaffinity_np(pthread_self(), sizeof(
        ↪ cpu_set_t), &cpuset);
    if (!unsuccessful) {
        cout << "Thread " << thread_number << " set to CPU " << cpu_id <<
            ↪ "" << endl;
        cpu_id_for_bounded_thread[thread_number] = cpu_id;
    }
    else {
        cout << "Thread " << thread_number << " failed to set to CPU " <<
            ↪ cpu_id << "" << endl;
    }
}
```

The implementation of thread affinity in the provided code is as follows. The code checks if the thread number is less than a predefined threshold ‘bt‘. If so, it initializes a ‘cpu set t‘ structure to represent the set of CPUs on which the thread will be allowed to

run. Then, it calculates the CPU ID by taking the modulo of the thread number with 'bc', which is then number of cores which are to be bounded. The thread's affinity is set to this CPU ID using the 'pthread setaffinity np' function. If the operation is successful, it prints a message indicating the thread is set to the specific CPU; otherwise, it prints an error message.

3 Performance Analysis

3.1 Experiment 1: Total Time vs Number of Bounded Threads

In this experiment, we evaluate the performance impact of thread affinity on matrix multiplication. We consider a square matrix of size $N = 1024$ and a thread count $K = 48$ (Thread count is modified so that K/C remains divisible. The logical processor count C is 12 (fixed) and b is computed as $b = K/C$. The graph shows four curves representing the Chunk and Mixed algorithms, with and without thread bounding.

3.1.1 Methodology

The Chunk and Mixed algorithms are executed with a varying number of threads bound to specific CPU cores. The bound thread counts are $b, 2b, 3b, \ldots$, up to K. Performance is measured as the time taken to compute the square of the matrix.

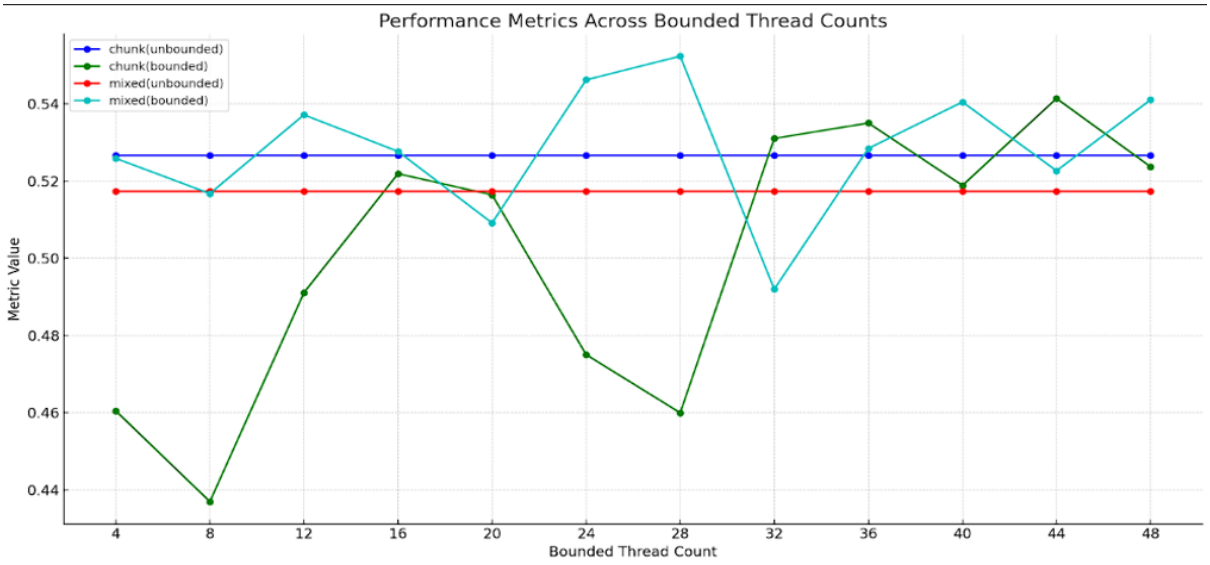

Figure 1: Total time vs bt (Number of bounded threads) N=1024 C=12

Table 2: Performance Metrics for Bounded Threads in Chunk and Mixed Methods

(BT)	Chunk	Mixed
0	0.5265966	0.5173028
4	0.4604406	0.5258612
8	0.436954	0.51674
12	0.4910696	0.5371448
16	0.5218812	0.5276614
20	0.5164098	0.509129
24	0.4750128	0.5462068
28	0.4599466	0.552329
32	0.5310136	0.4919752
36	0.5350102	0.52844
40	0.5188414	0.5404604
44	0.5413826	0.5226514
48	0.5236858	0.541028

3.1.2 Chunk Method Analysis

The Chunk method's performance improves initially with an increase in bounded threads (lower times compared to unbounded chunk), indicating better cache utilization and reduced cross-core data management overhead. However, a decline in performance is observed as more threads are bound, attributable to factors such as:

- Cache saturation leading to more frequent cache evictions.
- Increased resource contention within CPU cores.
- Inefficiencies introduced by scheduling overheads.
- Adverse effects due to Non-Uniform Memory Access (NUMA) on systems where this applies.

These observations suggest that there is an optimal number of threads for binding that maximizes cache efficiency without incurring additional overheads.

3.1.3 Mixed Method Analysis

The performance of the Mixed method exhibits relative insensitivity to the number of bounded threads. The interleaved access pattern inherent in the Mixed method results in a performance that is less dependent on cache locality. The consistent performance across varying numbers of bounded threads may be due to:

- A natural predisposition to cache misses regardless of thread affinity.
- Efficient utilization of cache due to uniform workload distribution.
- Effective load balancing by the operating system's scheduler when threads are not bound.

3.1.4 Observations and Reasoning

From the plot, we deduce that thread affinity significantly influences the Chunk method, enhancing performance through spatial locality. However, this improvement has limits, and performance can decrease when core saturation occurs. Conversely, the Mixed method's performance appears to be influenced more by algorithmic efficiency and hardware parallelism handling than by cache locality.

3.1.5 Conclusion

The experiment suggests that while thread affinity can be beneficial for computational tasks that exploit spatial locality, its advantages are not universal. The efficacy of thread affinity is contingent upon the computational workload, memory access patterns, and the hardware characteristics of the system. Identifying the optimal number of bounded threads requires balancing these factors to harness the full potential of thread affinity.

3.2 Experiment 2: Time vs Number of Threads

This experiment investigates the performance differences between CPU-bound threads and normal threads in matrix multiplication algorithms. The matrix size N is fixed at 1024, while the number of threads K varies from 6 to 96, increasing by multiples of 2. The average execution time for each thread type is measured and plotted against the number of threads.

3.2.1 Methodology

With $C = 12$ logical cores on the test laptop, the number of bounded threads BT is set to half the number of threads $K/2$, and these are assigned equally among $C/2$ cores. The remaining threads are managed by the operating system's scheduler. For instance, with $K = 48$, BT would be 24, and these would be distributed across 6 cores, assigning 4 threads per core.

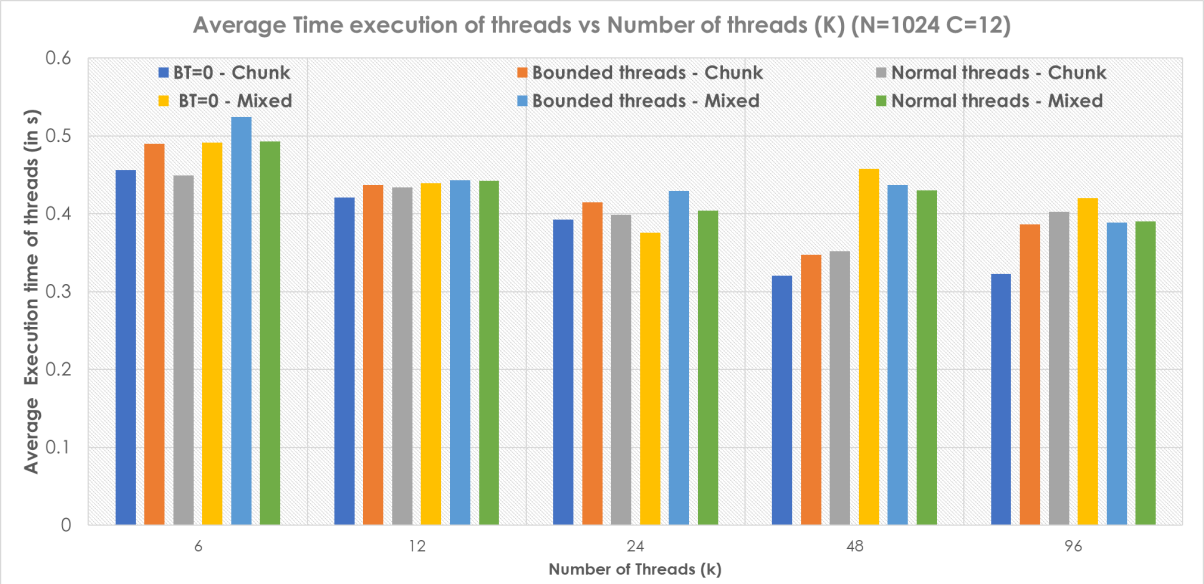

Figure 2: Avg Thread Execution time vs number of threads

Table 3: Chunk Method Performance Metrics

Threads (K)	BT=0	Bounded threads	Normal threads
6.000	0.456	0.490	0.449
12.000	0.421	0.437	0.434
24.000	0.392	0.414	0.399
48.000	0.320	0.348	0.352
96.000	0.323	0.387	0.402

Table 4: Mixed Method Performance Metrics

Threads(K)	BT=0	Bounded threads	Normal threads
6.000	0.491	0.525	0.493
12.000	0.439	0.443	0.442
24.000	0.376	0.429	0.404
48.000	0.458	0.437	0.430
96.000	0.420	0.389	0.390

Analysis of the Results

The bar chart illustrates the average execution times for both the Chunk and Mixed algorithms, comparing scenarios with and without thread affinity, as well as normal thread execution:

- For both algorithms, the average execution times without thread binding generally show a decrease as the number of threads increases, up to a certain point—specifically, $k = 48$ in this dataset. Beyond this point, the benefits of additional threads plateau, likely due to the overhead associated with managing a larger number of threads and the saturation of CPU resources.

- The average execution time for $BT = 0$ for both methods at all values of k is consistently lower than when $BT = \frac{k}{2}$. This observation suggests that the inherent CPU scheduling yields better average times for each thread compared to when threads are bound to specific cores.

- In the Chunk algorithm, CPU-bound threads exhibit a significant improvement in execution time with a smaller number of threads. However, as the number of threads increases—from $k = 48$ to $k = 96$—the performance benefit from thread affinity decreases and ultimately becomes negligible compared to that of normal threads.

- The Mixed algorithm's execution times appear to be less sensitive to the number of CPU-bound threads. The improvement in execution time is minimal across different thread counts, indicating that the Mixed algorithm may not benefit as much from thread affinity due to its inherent characteristics or workload distribution.

- In both algorithms, the performance of normal threads remains better than that of bounded threads. across all thread counts. This highlights the efficiency of the operating system's scheduling algorithm, which can dynamically allocate threads to cores based on real-time system load and other factors.

Reasoning Behind Observations

- The initial decrease in execution time with increasing threads can be attributed to the parallel nature of the algorithms, which allows them to utilize multiple cores effectively. As the number of threads exceeds the number of cores, the benefit of additional threads diminishes.

- The performance of CPU-bound threads in the Chunk algorithm suggests that when the number of threads matches the number of cores, there is a sweet spot where the data locality and reduced context switching significantly improve performance. As the number of threads exceeds this sweet spot, the overhead of synchronization and potential cache contention may offset the benefits of thread affinity.
- The consistent performance of normal threads could be due to the operating system's ability to balance the workload across all available cores, preventing any single core from becoming a bottleneck.

Conclusion

The experiment demonstrates that thread affinity can offer performance benefits in certain conditions, particularly when the number of threads is aligned with the number of available cores. However, these benefits are not uniform across different algorithms and diminish as the number of threads increases beyond the number of cores. It also showcases the robustness of the operating system's thread scheduler, which provides competitive performance without the need for manual thread-core binding.