CS 131 Homework 3 Report
Java Synchronization Performance Testing

Jon Paino
University of California, Los Angeles

Abstract

In modern software development, efficient multithreading is crucial for improving application performance, particularly in computationally intensive and I/O-bound environments. This report presents a comprehensive analysis of Java multithreading performance, focusing on various levels of synchronization. This investigation centers on a specific multithreaded operation: swapping values in an array of long integers. In each thread, two array locations are incremented and decremented simultaneously, effectively swapping values to maintain the overall integrity of data. The core of this analysis evaluates how different synchronization strategies affect both the execution speed and the accuracy of these operations.

1 Introduction

The Java Memory Model (JMM) serves as the cornerstone for understanding and designing synchronized behaviors in multithreaded Java applications. It outlines the rules that Java uses to read and write to shared memory safely and without data races. This project aims to explore and analyze the performance metrics of Java multithreading, with a focus on the implications of enforced synchronization levels in the context of both platform and virtual threads. Platform threads in Java have traditionally been bound to corresponding operating system (OS) threads, with each thread being a heavy-weight process. In contrast, virtual threads, introduced in more recent Java versions, are not tied to specific OS threads and are designed to be lightweight. They behave similar to how virtual memory works: by allowing the Java runtime to map a large number of virtual threads to a smaller number of OS threads, thereby optimizing resource utilization, especially during blocking I/O operations. This mapping frees up OS threads to perform other operations while the virtual thread is suspended, waiting for I/O to complete. By measuring execution times and evaluating the correctness of the multithreaded operations under various configurations, this study aims to provide insights into optimizing Java applications for different threading models and synchronization levels. These findings will contribute to a better understanding of the practical impacts of the Java Memory Model on application performance in real-world scenarios.

2 System Specifications

In this section of the report, we detail the hardware and software specifications of the system used to conduct the performance tests. Understanding these specifications is crucial as they can significantly influence the outcomes of multithreading performance tests. These details allow for a more accurate comparison of results across different systems with varying hardware and software configurations.

2.1 Software Environment

- **Java Version:** The tests were conducted using OpenJDK version 22.0.1, released on April 16, 2024. The Java environment details are as follows:
  - **Runtime Environment:** OpenJDK Runtime Environment (build 22.0.1+8-16)
  - **JVM Details:** OpenJDK 64-Bit Server VM (build 22.0.1+8-16, mixed mode, sharing)

This setup ensures that Java's latest features and optimizations are utilized, particularly those relevant to multithreading and concurrency.

2.2 Processor Information

- **Processor Type:** Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz
- **Number of Processors:** 4
- **Cores per Processor:** 4
- **Total Cores:** 16

- **Base Clock Speed:** 2095.079 MHz
- **Cache Size:** 16.5 MB (16896 KB)
- **Features:** The processors support advanced features like Hyper-Threading, AVX-512, and Intel's Turbo Boost technology, providing robust support for multitasking and parallel processing tasks.

2.3 Memory Details

- **Total Memory:** 64 GB (65638088 kB)
- **Free Memory:** Approximately 43 GB (43216056 kB) available at test time
- **Memory Type:** DDR4 (assumed based on processor compatibility)
- **Swap Total:** 8 GB (8388604 kB)
- **Swap Free:** 2.3 GB (2312844 kB)

3 Problems Faced

The main problem in analyzing these performance metrics was purely logistical. As there are 36 different combinations of thread number, thread type, array size, and synchronization level, doing this manually one by one is very tedious and difficult to organize. Instead, I wrote a shell script in order to automate this process and for greater reproducability in the future. [1].

4 Performance Analysis

All of this data was extracted from the csv file generated from the shell script referenced earlier. [2]

Among the synchronization types tested (Null, Unsynchronized, and Synchronized), there was a consistent pattern observed. Null synchronization was invariably the fastest, followed by Unsynchronized, with Synchronized proving to be the slowest. This pattern can be attributed to the overhead involved in acquiring and releasing locks in Synchronized threads. Each swap operation within this synchronization type requires locks to be managed, which adds significant overhead and delays to the process.

For Unsynchronized threads, an interesting observation was made: the real time difference between executing with any number of threads greater than one was essentially constant. This suggests that once the initial overhead of initializing multiple threads is overcome, the threads operate independently without any further significant delay due to synchronization. The minimal increase in real time from one to more than one thread is primarily due to the costs associated with setting up each thread.

Further analysis showed a notable distinction between Platform and Virtual threads. With a shared state size of 5, Platform threads significantly outperformed Virtual threads. However, as the number of threads increased to 40, the performance gap closed, with both types of threads showing similar times. This convergence in performance can be explained by the behavior of Platform threads, which are bound by the physical cores of the system. As the number of Platform threads exceeds the number of cores, the advantage diminishes due to context switching and other multitasking overheads, thereby leveling the playing field with Virtual threads.

Interestingly, Virtual threads demonstrated a slight reduction in real time when the thread count was increased from 5 to 40. This can be attributed to the lightweight nature of Virtual threads, which are designed to handle large numbers of concurrent tasks efficiently without heavy reliance on physical thread resources. As the number of Virtual threads increases, they can manage idle times (like waiting for I/O operations) more effectively, thus slightly improving throughput.

It is also important to note that with average swap time, platform threads performed much better than virtual threads. Platform threads can make better use of CPU core capabilities, especially under high synchronization, because each thread can run independently on separate cores. In contrast, Virtual threads are more susceptible to delays from lock contention because they might not be scheduled as efficiently by the JVM compared to how the OS handles native threads.

Lastly, incorrect swap results only occured within the Unsynchronized class with a thread count greater than one. This can be attributed to the fact that These findings underscore the critical impact of thread management and synchronization techniques on performance, particularly highlighting how different types of synchronization and threading can optimize or hinder Java application efficiency.

References

[1] Available online at https://github.com/jon-paino/java_synchronization/blob/main/jmm/testing.sh

[2] Available online at https://github.com/jon-paino/java_synchronization/blob/main/jmm/performance_data.csv

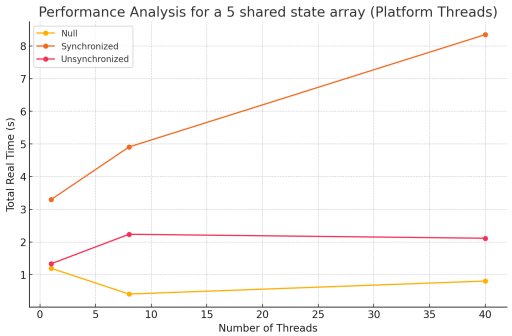

Figure 1: Platform threads on shared state of size 5

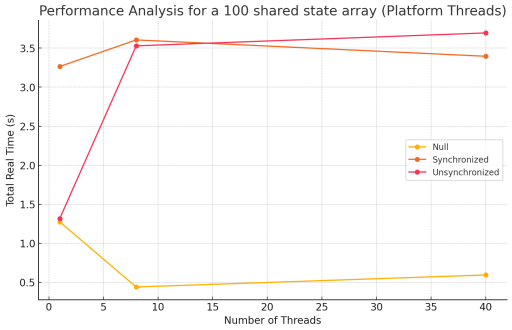

Figure 2: Platform threads on shared state of size 100

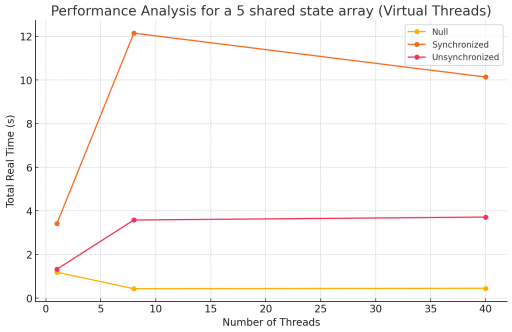

Figure 3: Virtual threads on shared state of size 5

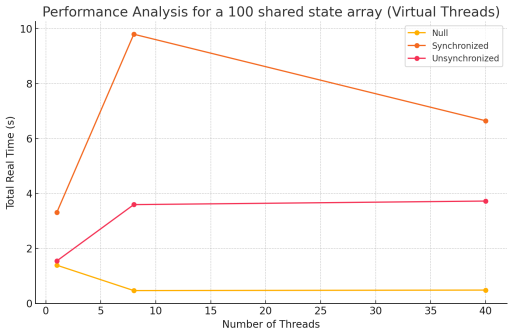

Figure 4: Virtual threads on shared state of size 100