Comparison Between a C++ and Java Application Implementation

Stephan Peters

Colorado State University Global

CSC450-1 23WD: Programming III

Reginald Haseltine

7 April 2024

Comparison Between a C++ and Java Application Implementation

This document compares performance and security considerations between a C++ and Java implementations of a program that uses threads to count up to 20, then count down from 20. To perform the performance, I will use `std::chrono` to benchmark the performance in the C++ program, and `edu.princeton.cs.algs4.Stopwatch` (Sedgewick & Wayne, 2014 § 1.4, pp 172-175) to benchmark the performance of the Java application. There are other stopwatch-like methods available in Java, such as `org.apache.commons.lang.time.StopWatch`, `org.springframework.util.StopWatch`, or `com.google.common.base.Stopwatch` (Guava), but I chose to use the method from algs4 because it can be included easily in the project by merely adding one class from the package, rather than relying on a build system such as Mavin or Gradle to download and install external library dependencies. The process of counting to 20 and back down is a fairly small task, a true benchmarking comparison might be better done counting to a much larger number.

Differences in security considerations will be primarily derived from my previous analyses of these two implementations, as well as other sources that define security concerns for the two languages.

Initial Benchmarking Tools Setup

For the C++ implementation, I added some benchmarking code to measure both the static and non-static methods used to perform the task. See Source Code 1. For the Java implementation, I directly added a Stopwatch object to a Count object which starts timing at the creation of the Count object, and a public method to retrieve the current count to the Count class. See source code 2.

Source Code 1

std::chrono benchmarking setup for Count.cpp.

```
#include <chrono>

. . .

int main() {
    using chrono::high_resolution_clock;
    using chrono::duration_cast;
    using chrono::duration;
    using chrono::milliseconds;

    // Initialize the minimum, maximum, and increment.
    int min = 0, max = 20, increment = 1;

    auto c1 = high_resolution_clock::now();
    // Using a non-static object through a worker function.
    cout << "Demonstrates using a non-static class object for the threads."
         << endl;
    thread t1(worker, min, max, increment, false, false);
    t1.join();
    thread t2(worker, min, max, increment, true, false);
    t2.join();
    auto c2 = high_resolution_clock::now();

    auto c3 = high_resolution_clock::now();
    // Using static functions.
    cout << endl << "Demonstrates using a static functions for the threads."
         << endl;
    thread t3(staticCountUp, min, max, increment);
    t3.join();
    thread t4(staticCountDown, min, max, increment);
    t4.join();
    auto c4 = high_resolution_clock::now();

    // Display benchmark data:
    duration<double, std::milli> nonstatic_duration = c2 - c1;
    duration<double, std::milli> static_duration = c4 - c3;
    fprintf(stdout, "\nDuration using the non-static method: %.3f ms",
        nonstatic_duration.count());
    fprintf(stdout, "\nDuration using the static method: %.3f ms",
        static_duration.count());

    return 0;
}
```

Source Code 2

edu.princeton.cs.algs4.Stopwatch benchmarking setup for Count.java.

```
import edu.princeton.cs.algs4.Stopwatch;

public class Count {
    Stopwatch stopwatch;

    . . .

    public Count(int min, int max) {
        this.stopwatch = new Stopwatch();
        this.min = min;
        this.MAX = max;
    }

    . . .

    public void getElapsedTime() {
        synchronized (this) {
            try {
                latch.await();
            } catch (InterruptedException ie) {
                ie.printStackTrace();
            }
            System.out.printf("\nDuration using the Java method: %.1f ms\n",
                stopwatch.elapsedTime() * 1000);
        }
    }

    . . .

}
```

Benchmarking Results

I ran and recorded the benchmarking results of both implementations 10 times. The results can be seen in Table 1. All readings are in milliseconds. The average time to run the C++ implementations is 10.774 milliseconds, while the average time to run the Java implementation is 13.1 milliseconds, around 22% higher than the C++ implementations. There seems to be no measurable difference between the static and non-static C++ methods with this small sample size.

Table 1

Benchmark results for the C++ and Java implementations. Measurements are in milliseconds.

	C++ Non-Static	C++ Static	Java
Reading 1	10.321	8.881	13
Reading 2	11.643	12.182	14
Reading 3	9.724	11.28	12
Reading 4	12.138	11.347	14
Reading 5	10.732	8.96	13
Reading 6	11.07	9.877	14
Reading 7	10.37	10.486	14
Reading 8	9.76	10.916	12
Reading 9	13.137	11.737	12
Reading 10	11.391	9.522	13
Average	11.029	10.519	13.1

Retesting the results while counting to a higher number would be an interesting follow-up, to see if the 22% increase for the Java implementation holds with larger numbers.

Security Vulnerability Comparison

While the C++ implementation benchmarks with better performance than the Java implementation, the Java implementation provides much better security. In my former analyses of these implementations, I found the main security concern with the Java implementation was there is no input validation or sanitization for the integers passed to the Count class (Peters, 2024b). This would easily be rectified in a production release, though it would increase the benchmark time even more, and would require testing to see how much more.

The C++ implementation was found to be at risk for at least one Carnegie Mellon CERT C++ coding standard (CON50-CPP) (Carnegie Mellon University Software Engineering Institute, 2023a) and fully non-compliant with another (CON55-CPP) (Carnegie Mellon University Software Engineering Institute, 2023b) (Peters, 2024a). This non-compliant code certainly leads to security concerns in a production model.

Conclusion

This small test, with a simple program and a small sample size certainly reinforces the common conception that as a rule C++ is faster than Java, but Java has less security concerns than C++. This is really an over-simplification, as any C / C++ program written and tested with security in mind will almost certainly have less security vulnerabilities than an equivalent Java program that is written with performance in mind rather than security. C / C++ vulnerabilities are likely to be caused by memory access, while Java vulnerabilities are likely to be caused by privilege escalation (Svoboda, 2015). So, in reality, while C++ has more exploitable vulnerabilities, either language can be vulnerable if the code is not written with security in mind.

References

Carnegie Mellon University Software Engineering Institute. (2023a, April 20). *CON50-CPP. Do not destroy a mutex while it is locked - SEI CERT C++ Coding Standard - Confluence*. Wiki.sei.cmu.edu. https://wiki.sei.cmu.edu/confluence/display/cplusplus/CON50-CPP.+Do+not+destroy+a+mutex+while+it+is+locked

Carnegie Mellon University Software Engineering Institute. (2023b, December 21). *CON55-CPP. Preserve thread safety and liveness when using condition variables - SEI CERT C++ Coding Standard - Confluence*. Wiki.sei.cmu.edu. https://wiki.sei.cmu.edu/confluence/display/cplusplus/CON55-CPP.+Preserve+thread+safety+and+liveness+when+using+condition+variables

Peters, S. (2024a, March 31). *Security and Concurrency in C++*. GitHub. https://github.com/speters33w/CSUGlobal_CSC450/blob/main/PortfolioProjectMilestone/doc/PortfolioProjectMilestone_Concurrency_in_CPP.pdf

Peters, S. (2024b, April 5). *Security and Concurrency in Java*. GitHub. https://github.com/speters33w/CSUGlobal_CSC450/blob/main/PortfolioProject/doc/detailed_analysis.pdf

Sedgewick, R., & Wayne, K. (2014). *Algorithms*. Addison-Wesley Professional.

Svoboda, D. (2015, October 5). *Is Java More Secure than C?* SEI Blog. https://insights.sei.cmu.edu/blog/is-java-more-secure-than-c/