Comparison of Java Virtual and Non-Virtual Threads in Parallel Programming

Zlatko Sirotić[1,2], Siniša Sovilj[2], Matija Oršulić[2], Krešimir Pripužić[3]
[1]ISTRA TECH d.o.o., Pula, Croatia
[2]Juraj Dobrila University of Pula, Faculty of Informatics, Pula, Croatia
[3]University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia

Abstract
The Java programming language, in its long-term support version 21 (released in September 2023), introduced Java virtual threads (prior to version 21, this was a prototype solution). While Java non-virtual (or platform) threads are mapped one-to-one to operating system threads, multiple Java virtual threads can be executed on a single platform thread, i.e. on a single operating system thread, which makes the number of Java virtual threads practically unlimited. Java virtual threads enable writing code that is as efficient as asynchronous code but is more readable, much easier to debug, and provides clearer error messaging (exceptions) compared to asynchronous code. It is often stated that virtual threads are primarily intended for concurrent programming (e.g. each active user on an application server can now be assigned a separate virtual thread) and are less suited for parallel programming (where a task is divided into subtasks to utilize the processing power of multiprocessor systems).

In this paper, we analyze the efficiency of virtual threads specifically in parallel programming (counting prime numbers that are less than a given number and the goal was not to create a particularly efficient program, but to compare different methods). First, we demonstrate the use of non-virtual Java threads in Java versions 5/6 (Executors), Java 7 (Fork Join framework), and Java 8 (Parallel Streams), and then we modify the Java 5/6 program (with very simple changes) to an equivalent program using Java 21 virtual threads. It was found that Java 21 virtual threads were slightly slower in our case than the solution using non-virtual threads.

Keywords – concurrent and parallel programming in Java; Java non-virtual (platform) threads; Java virtual threads;

I. Introduction

Java threads have been part of the Java language from the start and are built on OS threads. Every Java program has at least one thread, the one that executes the `main()` method. Working with multiple Java threads can serve at least two different purposes:

- Parallelization of a single program, so the program can utilize all hardware processors (hardware threads or cores);
- Concurrent programming, where each user is assigned a Java thread (usually from a Java thread pool) to handle their request.

Working with Java threads is not trivial [1] (parallel and concurrent programming is generally not simple), but it is also not terribly complicated either (at least since Java 5 and onwards). However, since Java threads are built on OS threads, they have two significant drawbacks:

- time overhead due to context switching;
- number of OS threads is limited (today, several thousand is manageable, but several million is usually too many).

Java asynchronous programming (including reactive programming) addresses these issues, but asynchronous programs have their own major drawbacks [2]:

- programs are challenging to write, read, and test;
- debugging is hardly feasible;
- profiling is practically impossible.

Java 21 (released in September 2023) introduced virtual threads. As stated in the Java 21 documentation [3]: *"Virtual threads are cheap to create, to a point where you can have as many as you need"*. In addition to the term Virtual Thread, a new term also introduced is Platform Thread: *"A platform thread is implemented as a thin wrapper around an operating system (OS) thread... consequently, the number of available platform threads is limited to the number of OS threads"*. In contrast to Platform Thread: *"... a virtual thread is an instance of java.lang.Thread. However, a virtual thread isn't tied to a specific OS thread. A virtual thread still runs code on an OS thread. However, when code running in a virtual thread calls a blocking I/O operation, the Java runtime suspends the virtual thread until it can be resumed. The OS thread associated with the suspended virtual thread is now free to perform operations for other virtual threads"*.

So, to run code in a virtual thread, the JDK's scheduler assigns the virtual thread for execution on a platform thread by mounting the virtual thread on the platform thread. This makes the platform thread become the carrier of the virtual thread. After running some code, the virtual thread can unmount from its carrier. At that point the platform thread is released so that the JDK's scheduler can mount a different virtual thread on it, thereby making it a carrier again.

The Java documentation explains the purpose of virtual threads: "Use virtual threads in high-throughput concurrent applications, especially those that consist of a large number of concurrent tasks that spend much of their time waiting... Virtual threads are not faster threads; they do not run code any faster than platform threads. They exist to provide scale (higher throughput), not speed (lower latency)."

The Java documentation explains what virtual threads are (and what they are not) intended for: *"Use virtual threads in high-throughput concurrent applications, especially those that consist of a great number of concurrent tasks that spend much of their time waiting; Virtual threads are not faster threads; they do not run code any faster than platform threads. They exist to provide scale (higher throughput), not speed (lower latency)".*

We analyzed whether virtual threads are significantly slower than traditional (non-virtual) threads in parallel programs. We created three parallel programs using non-virtual threads in three different ways (`Executors`, `ForkJoin` framework, `Streams`) and one program using virtual threads (`Executors`) and compared their execution speed on two computers (one with an Intel i7 7th-generation chip, the other with an Intel i7 13th-generation chip). The programs count how many prime numbers exist between 1 and a user-input number.

II. METHODS

Here, we briefly describe four Java programs that solve the task (counting prime numbers). These four solutions were made in: Java 5 (Executors), 7 (ForkJoin), 8 (Streams), and 21 (virtual Executors). The first three code variants (Executors, ForkJoin, and Streams) were partially analyzed in our paper written in Croatian [7]; however, virtual threads were not available back then. Code details are provided in Appendix A. Note that the goal was not to find the fastest algorithm but to compare the speed of different Java methods.

A. *Executors (Java 5)*

The first program uses Java Executors and has been available since Java 5. In Java 5, the `java.util.concurrent` package introduced executors (i.e., interfaces like `Executor`, `ExecutorService`, `Callable`, `Future`, and classes like `Executors`, `ThreadPoolExecutor`, `FutureTask`, etc.) [4]. Executors help programmers focus on creating tasks to be executed, while the executors handle optimization using a thread pool. Tasks are instances of classes (often anonymous) that implement the `Runnable` or `Callable` interface.

The main question is: How to divide the task into subtasks? A related question is: How many Java threads to create? Perhaps as many as there are subtasks? This is usually not ideal! A simple formula can be used to (approximately) determine the number of Java threads:

```
number_of_Java_threads = number_of_HW_threads /
(1 - blocking_coefficient)
```

The blocking coefficient is a number between 0 and 1 and is not easy to determine. Computationally intensive problems have a coefficient close to zero, so the recommended number of Java threads is equal to or slightly higher than the number of hardware threads.

We need to determine how to divide the problem into subtasks, with at least as many subtasks as Java threads to ensure concurrency. A simple approach is to set the number of subtasks high enough to fully utilize the threads, preventing idleness. While the exact number is hard to determine and may require experimentation, increasing the subtasks usually boosts performance initially, but the gains diminish with further increases, and performance can eventually decrease.

The Java 5/6 program is based on the `PrimesExecutor` class and has the following input parameters:

- `number`: the upper limit for finding prime numbers;
- `poolSize`: the number of Java threads;
- `numberOfParts`: the number of subtasks.

B. *ForkJoin Framework (Java 7)*

The second program uses the Java 7 ForkJoin framework, which follows the old Roman motto: "*Divide et impera*" ("Divide and Conquer"). The `ForkJoinPool` class implements the `ExecutorService` interface. Typically, only one task is sent to a `ForkJoinPool` instance, and the task and pool instance together apply the Divide and Conquer technique. The number of Java threads in the pool can be specified:

- Explicitly (e.g., 8):
  `ForkJoinPool fjPool = new ForkJoinPool(8);`
- Implicitly, using `Runtime.availableProcessors()`:
  `ForkJoinPool fjPool = new ForkJoinPool();`

The task should be an instance of a subclass of the abstract class `ForkJoinTask`, typically `RecursiveTask` or `RecursiveAction`. Key methods include `compute()`, `fork()`, and `join()`.

Unlike most `ExecutorService` implementations, in the ForkJoin framework, each Java thread in the `ForkJoinPool` has its own queue of subtasks. The `fork()` method places a `ForkJoinTask` in the current thread's queue. Initially, only one thread is occupied - the one to which the entire task is sent. The thread then divides the task into two subtasks, placing the first (left) subtask in the queue and attempting to execute the second (right) subtask (recursively).

A key feature of the ForkJoin framework is Work Stealing. Java threads steal tasks (subtasks) from other threads' queues and add them to their own queue.

The order of subtasks in the queue is critical. Larger tasks should be placed first. For instance, an initial task covering 100% of the work is split into two 50% subtasks. The first goes into the queue, and the second is processed and divided further. Java threads without work will steal tasks from others, taking the oldest and largest task in the queue. Subtask division is implicit and based on when a task becomes small enough, with the size typically determined experimentally. When calling `join()`, a subtask may have already been stolen and completed by another thread, or may still be in progress. Calling the `join()` method in the `compute()` method should be one of the last actions, after the `fork()` method and the recursive `compute()` method call. Since this is very important, there is also the `invokeAll(a2, a1)` method which can replace the sequence of `a1.fork()`; `a2.compute()`; `a1.join()`;

The Java 7 program is based on the PrimesForkJoin class and has the following input parameters:

- `number`: the upper limit for finding prime numbers;
- `poolSize`: the number of Java threads;
- `threshold`: the number that defines when a subtask is small enough (and should no longer be divided).

C. *Streams (Java 8)*

The third program uses the Java 8 Stream API, which introduced key features like lambda expressions and default methods in interfaces [5], [6]. To add new methods to interfaces without breaking existing code, Java 8 introduced default methods, allowing both method declarations and implementations.

Streams enable existing collections to be wrapped in (parallel) Streams, allowing for indirect parallelization. Unlike traditional collections, which store all their elements in memory, streams can be thought of as “temporal collections”, where elements (which can theoretically be infinite sequences) are generated as needed. Parallel streams internally rely on the ForkJoin framework.

The Java 8 program is based on the `PrimesStream` class and has the following input parameters:

- `number`: the upper limit for finding prime numbers;
- `isParallel`: 0 for serial execution, 1 for parallel execution.

In the last variant, the number of Java threads in the pool is either 1 (non-parallel case) or equal to the number of available hardware threads minus 1 (this could be set differently, but we wanted to keep the default value).

D. *Virtual Threads (Java 21)*

The fourth program (also using Executors, like the first) uses Java 21 virtual threads. In Appendix A, the Java 5 and Java 21 programs are shown side by side, making it easy to see that the difference between the two is minimal.

Virtual threads are internally based on the ForkJoin framework, which can have a default number of Java threads, so in this program, the user only inputs the `number` and `numberOfParts` parameters. Note that it would not make sense to use virtual threads with the ForkJoin and Streams variants (since Streams are internally based on ForkJoin), as this would result in “ForkJoin on top of ForkJoin”.

III. Results and discussion

We executed the four programs using Java 21 JRE on two different computers:

- An older computer with an Intel i7-7700HQ processor at 2.8 GHz, with 24 GB of RAM; it has 4 Performance cores supporting Hyper-threading, meaning it has 8 hardware threads;
- A newer computer with an Intel i7-13700H processor at 2.4 GHz, with 16 GB of RAM; it has 6 Performance cores and 8 Efficient cores, totaling 20 hardware threads (2 * 6 + 8).

For the two programs using non-virtual threads (Executors and ForkJoin), we tested with different numbers of Java threads (parameter `poolSize`): 1, 2, 4, 6, 8, 14, 20, 32. On the older processor (8 hardware threads), it was sufficient to test with 1, 2, 4, 8 Java threads, but for the newer processor, we also tested with 6 and 14 Java threads (since it has 6 + 8 cores) and 20 Java threads (since it has 20 hardware threads).

For the Streams program and the program with virtual threads, the number of Java threads is the default (number of hardware threads: 8 for the older processor, 20 for the newer one).

All programs were tasked with finding prime numbers less than or equal to 10,000,000. The `numberOfParts` parameter was set to 1000, and the `threshold` parameter was set to 100. The following table shows the average execution times (in seconds, rounded to two decimal places) over 10 runs for each of the four programs on both processors:

Java threads	Executors time (seconds) on Intel i7 7700HQ / 13700H	Executors on Java virtual threads	ForkJoin	Streams
1	7.52 / 2.01	-	7.72 / 2.20	7.20 / 2.01
2	4.03 / 1.01	-	4.09 / 1.12	-
4	2.20 / 0.51	-	2.25 / 0.58	-
6	1.77 / 0.35	-	1.81 / 0.40	-
8	**1.54** / 0.30	**1.56** / -	1.57 / 0.35	1.63 / -
14	1.54 / 0.23	-	1.57 / 0.28	-
20	1.54 / **0.22**	- / **0.23**	1.57 / 0.28	- / 0.22
32	1.54 / 0.22	-	1.59 / 0.29	-

The table shows the following:

- As the number of Java threads increases (for Executors and ForkJoin), the programs run faster until the number of hardware threads (8 or 20) is reached;
- The Executors solution is slightly faster than ForkJoin and Streams, and Streams is generally faster than ForkJoin;
- Most importantly (the goal of this analysis), the solution with non-virtual threads is not significantly faster than the solution with virtual threads: 1.54 vs. 1.56 seconds (i7-7700HQ, 8 Java threads), or 0.22 vs. 0.23 seconds (i7-13700H, 20 Java threads).

IV. Conclusion

Virtual Java threads (introduced in Java 21) are comparable in speed to asynchronous programming for concurrent applications, but without the drawbacks of asynchronous programs, which are difficult to write, read, and test, and practically impossible to debug and profile.

Although virtual Java threads are not primarily intended for parallel programming, our example shows that a parallel program using Java Executors, after very minor modifications (highlighted in bold in Appendix A), runs slightly slower on virtual threads.

Acknowledgement

This research is partly supported by 'European Digital Innovation Hub Adriatic Croatia (EDIH Adria) (project no. 101083838)' under the European Commission's Digital Europe Programme, SPIN project 'INFOBIP Konverzacijski Order Management (IP.1.1.03.0120)', SPIN project 'Projektiranje i razvoj nove generacije laboratorijskog informacijskog sustava (iLIS)' (IP.1.1.03.0158)'.

References

[1] B. Göetz, *Java Concurrency in Practice*, Addison-Wesley, 2006.

[2] J. Paumard, *Are Virtual Threads Going to Make Reactive Programming Irrelevant?*, https://2024.javazone.no/program/, 2024.

[3] Oracle: *Java Platform SE Core Libraries (Release 21)*, https://docs.oracle.com/en/java/javase/21/core/, 2024.

[4] K. Sierra and B. Bates, *OCA/OCP Java SE 7 Programmer I & II Study Guide*, McGraw-Hill Education, 2015.

[5] P-Y. Saumont, *Functional Programming in Java*, Manning, 2017.

[6] J. Boyarsky and S. Selikoff, *OCP: Oracle Certified Professional Java SE 8 Programmer II Study Guide*, Sybex, 2015.

[7] Z. Sirotić, *Paralelno programiranje u Javi.* Zbornik radova konferencije CASE 29, Zagreb, Hrvatska, 2017.

APPENDIX A

```java
// Java 5/6 Executors
import java.util.List;
import java.util.ArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Callable;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class PrimesExecutor {
  public static void main(final String[] args) {
    if (args.length != 3) {
      System.out.println("Usage: number poolSize numberOfParts");
      return;
    }
    final long number = Long.parseLong(args[0]);
    final int poolSize = Integer.parseInt(args[1]);
    long numberOfParts = Long.parseLong(args[2]);
    if (numberOfParts > number) numberOfParts = number / 10;
    final long numberOfPrimes;
    final long startTime = System.nanoTime();
    if (number <= 1)
      numberOfPrimes = 0;
    else {
      PrimesExecutor task = new PrimesExecutor();
      numberOfPrimes = task.countPrimes(number, poolSize, numberOfParts);
    }
    final long endTime = System.nanoTime();
    System.out.printf("Primes under %d is %d\n", number, numberOfPrimes);
    System.out.println("Time (seconds): " + (endTime - startTime) / 1.0e9);
  }

  private long countPrimes
  (final long number, final int poolSize, final long numberOfParts) {
    long count = 0;
    try {
      final List<Callable<Long>> partitions = new ArrayList<>();
      final long chunksPerPartition = (number + numberOfParts - 1) /
                                      numberOfParts;
      for(long i = 0; i < numberOfParts; i++) {
        final long lower = i * chunksPerPartition + 1;
        long upperTemp = (i + 1) * chunksPerPartition;

        if (upperTemp > number) upperTemp = number;
        final long upper = upperTemp;

        partitions.add(new Callable<Long>() {
          public Long call() {
            return countPrimesInRange(lower, upper);
          }
        });
      }
```

```java
// Java 21 Virtual Executors
import java.util.List;
import java.util.ArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Callable;
import java.util.concurrent.Future;
import java.util.concurrent.TimeUnit;

public class PrimesVirtualExecutor {
  public static void main(final String[] args) {
    if (args.length != 2) {
      System.out.println("Usage: number numberOfParts");
      return;
    }
    final long number = Long.parseLong(args[0]);

    long numberOfParts = Long.parseLong(args[1]);
    if (numberOfParts > number) numberOfParts = number / 10;
    final long numberOfPrimes;
    final long startTime = System.nanoTime();
    if (number <= 1)
      numberOfPrimes = 0;
    else {
      PrimesVirtualExecutor task = new PrimesVirtualExecutor();
      numberOfPrimes = task.countPrimes(number, numberOfParts);
    }
    final long endTime = System.nanoTime();
    System.out.printf("Primes under %d is %d\n", number, numberOfPrimes);
    System.out.println("Time (seconds): " + (endTime - startTime) / 1.0e9);
  }

  private long countPrimes
  (final long number, final long numberOfParts) {
    long count = 0;
    try {
      final List<Callable<Long>> partitions = new ArrayList<>();
      final long chunksPerPartition = (number + numberOfParts - 1) /
                                      numberOfParts;
      for(long i = 0; i < numberOfParts; i++) {
        final long lower = i * chunksPerPartition + 1;
        long upperTemp = (i + 1) * chunksPerPartition;

        if (upperTemp > number) upperTemp = number;
        final long upper = upperTemp;

        partitions.add(new Callable<Long>() {
          public Long call() {
            return countPrimesInRange(lower, upper);
          }
        });
      }
      final ExecutorService executorPool =
        Executors.newFixedThreadPool(poolSize);
      final List<Future<Long>> resultFromParts =
        executorPool.invokeAll(partitions, 10000, TimeUnit.SECONDS);
      executorPool.shutdown();
      for(final Future<Long> result : resultFromParts) {
        count += result.get();
      }
    } catch(Exception ex) { throw new RuntimeException(ex); }
    return count;
  }

  private static long countPrimesInRange(long lower, long upper) {
    long total = 0;

    // even numbers are also checked, for speed comparison
    // with the Streams implementation (LongStream.rangeClosed step = 1)
    for(long i = lower; i <= upper; i++) {
      if (isPrime(i)) total++;
    }

    // 1 is not prime, isPrime for 2 returns false, total number is correct
    return total;
  }


  private static boolean isPrime(final long number) {
    if (number % 2 == 0) return false;

    for(long i = 3; i * i <= number; i = i + 2) {
      if (number % i == 0) return false;
    }
    return true;
  }
}
```

```java
      final ExecutorService executorPool =
        Executors.newVirtualThreadPerTaskExecutor();
      final List<Future<Long>> resultFromParts =
        executorPool.invokeAll(partitions, 10000, TimeUnit.SECONDS);
      executorPool.shutdown();
      for(final Future<Long> result : resultFromParts) {
        count += result.get();
      }
    } catch(Exception ex) { throw new RuntimeException(ex); }
    return count;
  }

  private static long countPrimesInRange(long lower, long upper) {
    long total = 0;

    // even numbers are also checked, for speed comparison
    // with the Streams implementation (LongStream.rangeClosed step = 1)
    for(long i = lower; i <= upper; i++) {
      if (isPrime(i)) total++;
    }

    // 1 is not prime, isPrime for 2 returns false, total number is correct
    return total;
  }


  private static boolean isPrime(final long number) {
    if (number % 2 == 0) return false;

    for(long i = 3; i * i <= number; i = i + 2) {
      if (number % i == 0) return false;
    }
    return true;
  }
}
```

```java
// Java 7 ForkJoin
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.ExecutionException;

public class PrimesForkJoin extends RecursiveTask<Long> {
  private static int threshold;
  private long start;
  private long end;

  private PrimesForkJoin(final long theStart, final long theEnd) {
    start = theStart;
    end = theEnd;
  }

  public static void main(final String[] args) {
    if (args.length != 3) {
      System.out.println("Usage: number poolSize threshold");
      return;
    }
    final long number = Long.parseLong(args[0]);
    final int poolSize = Integer.parseInt(args[1]);
    long threshold = Long.parseLong(args[2]);
    if (threshold > number) threshold = number;
    final long numberOfPrimes;
    final long startTime = System.nanoTime();
    if (number <= 1)
      numberOfPrimes = 0;
    else {
      PrimesForkJoin task = new PrimesForkJoin(1, number);
      ForkJoinPool fjPool = new ForkJoinPool(poolSize);
      numberOfPrimes = fjPool.invoke(task);
    }
    final long endTime = System.nanoTime();
    System.out.printf("Primes under %d is %d\n", number, numberOfPrimes);
    System.out.println("Time (seconds): " + (endTime - startTime) / 1.0e9);
  }

  protected Long compute() {
    if (end - start <= threshold) {
      return countPrimesInRange(start, end);
    } else {
      long halfWay = ((end - start) / 2) + start;
      PrimesForkJoin t1 = new PrimesForkJoin(start, halfWay);
      PrimesForkJoin t2 = new PrimesForkJoin(halfWay + 1, end);
      t1.fork();
      long count2 = t2.compute();
      long count1 = t1.join();
      return count1 + count2;
    }
  }

  ... countPrimesInRange and isPrime - as before
}
```

```java
// Java 8 Streams
import java.util.stream.*;
import java.util.concurrent.TimeUnit;

public class PrimesStream {
  public static void main(final String[] args) {
    if (args.length != 2) {
      System.out.println("Usage: number 1/0 (parallel/not parallel)");
      return;
    }

    final long number = Long.parseLong(args[0]);
    final boolean isParallel =
      (Integer.parseInt(args[1]) == 1) ? true : false;
    final long numberOfPrimes;
    final long startTime = System.nanoTime();

    if (number <= 1)
      numberOfPrimes = 0;
    else
      numberOfPrimes = countPrimes(number, isParallel);

    final long endTime = System.nanoTime();
    System.out.printf("Primes under %d is %d\n", number, numberOfPrimes);
    System.out.println("Time (seconds): " + (endTime - startTime) / 1.0e9);
  }


  protected static long countPrimes
  (final long number, final boolean isParallel) {
    long count;

    if (isParallel) {
      count = LongStream.rangeClosed(1L, number)
                        .parallel()
                        .filter(n -> isPrime(n)) // lambda expression
                        .count();
    } else {
      count = LongStream.rangeClosed(1L, number)
                        .filter(PrimesStream::isPrime) //method reference
                        .count();
    }

    // 1 is not prime, isPrime for 2 returns false, total number is correct
    return (count);
  }

  ... isPrime - as before
}
```