Performance analysis & optimization

Table of Contents

[Original solution analysis 2](#_Toc150183870)

[Original solution (v0) 2](#_Toc150183871)

[No BigIntegerIterator (v1) 4](#_Toc150183872)

[No exception thrown (v2) 7](#_Toc150183873)

[Single-threaded solution 13](#_Toc150183874)

[Naïve solution (v3) 13](#_Toc150183875)

[Basic Eratosthenes Sieve (v4) 16](#_Toc150183876)

[Eratosthenes Sieve odd numbers (v5) 18](#_Toc150183877)

[Multi-threaded solution 21](#_Toc150183878)

**Note**:

\* since for numbers close to INT\_MAX output to the console can take significant time (proof in profiling), we will initially eliminate this factor by printing only a maximum prime number found by the algorithm

A screen shot of a computer

Description automatically generated

**Profiler setup**:

A screenshot of a black screen

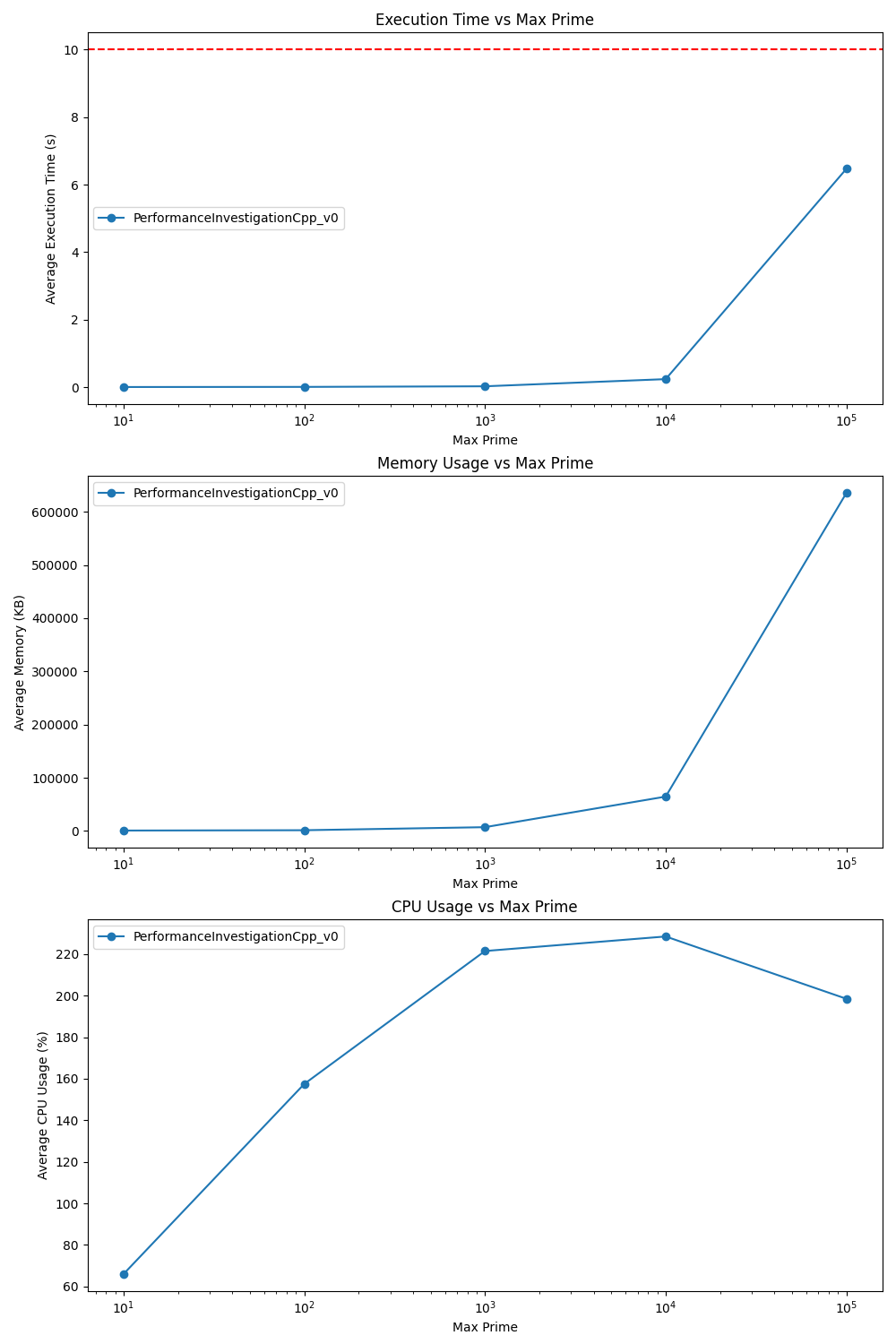
Description automatically generatedDefault CLion profiler (based on Dtrace) is used:

# Original solution analysis

## Original solution (v0)

Original implementation is labeled as **PerformanceInvestigationCpp\_v0**.

Early signs of performance problems: both execution time and memory consumption (6s+ and 600Mb+ with maxPrime=100k)



Apparently, ‘myFiller’ vector is not needed as it’s playing a temporary role to store int numbers that are pushed into ‘primeNumbers’ vector later:

std::vector<BigIntegerIterator> myFiller;

for (auto integer: myFiller) {  
 primeNumbers.push\_back(integer.getContain());  
}

The problem is that it not only store them in additional vector, but uses two additional vectors of ‘int’ and ‘string’ types reserved for 500 elements to store a single value:

private:  
 std::vector<std::string> contain;  
 std::vector<int> reference;  
};

contain.reserve(500);  
reference.reserve(500);

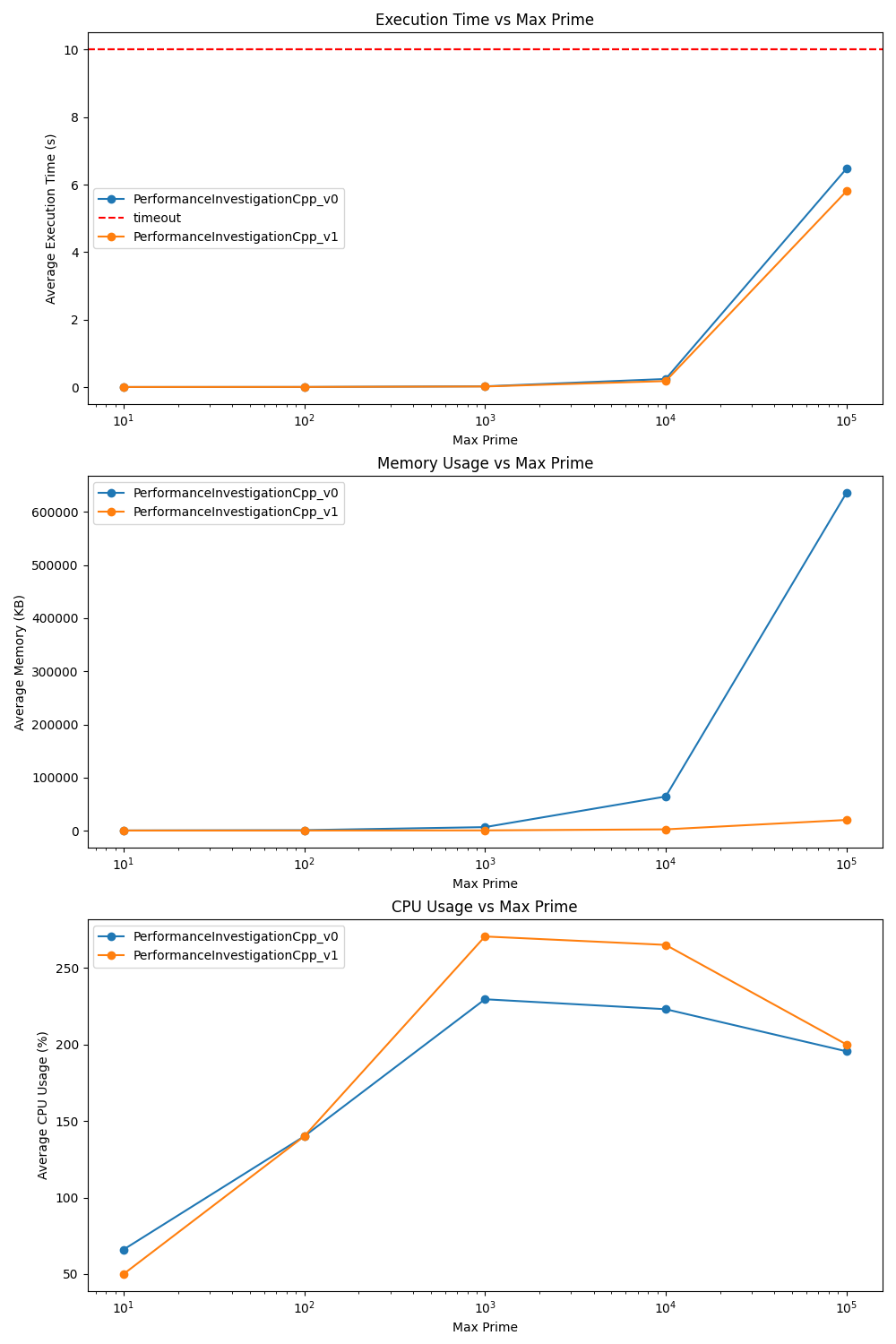
## No BigIntegerIterator (v1)

In order to fix **Memory Inefficiencies**, we can completely **get rid of ‘BigIntegerIterator’ class and ‘myFiller’ vector** by pushing numbers directly to ‘primeNumbers’ vector:

std::vector<int> primeNumbers;  
  
for (int j = 2; j <= maxPrime; ++j) {  
 primeNumbers.push\_back(j);  
}

That version is marked as **PerformanceInvestigationCpp\_v1**.

Benchmark comparison (**v0 vs v1**):



Memory consumption has dropped from **~650Mb to ~20Mb** for maxPrime = 100k.

A screenshot of a computer

Description automatically generatedHowever, we haven’t dealt with slow execution time yet. Let’s analyze performance profile:

**Throwing an exception appears to be inefficient way** to indicate that a number is not prime:

if (candidate % primeNumbers[j] == 0) {  
 throw std::exception();  
}

## No exception thrown (v2)

**Boolean value should be returned** instead:

if (candidate % prime == 0) {  
 return false;  
}

‘isPrime’ also can iterate over numbers up to a candidate’s square root to check for a divisibility:

int sqrtCandidate = static\_cast<int>(std::sqrt(candidate));  
 for (int prime : primeNumbers) {  
 if (prime > sqrtCandidate) {  
 break;  
 }  
 if (candidate % prime == 0) {  
 return false;  
 }  
 }  
 return true;  
}

Solution is marked as **PerformanceInvestigationCpp\_v2**.

Benchmark comparison (**v1 vs v2**):

Surprisingly, there’s no significant benefit compared to a version with exceptions thrown.

A graph of a number of data

Description automatically generated with medium confidence

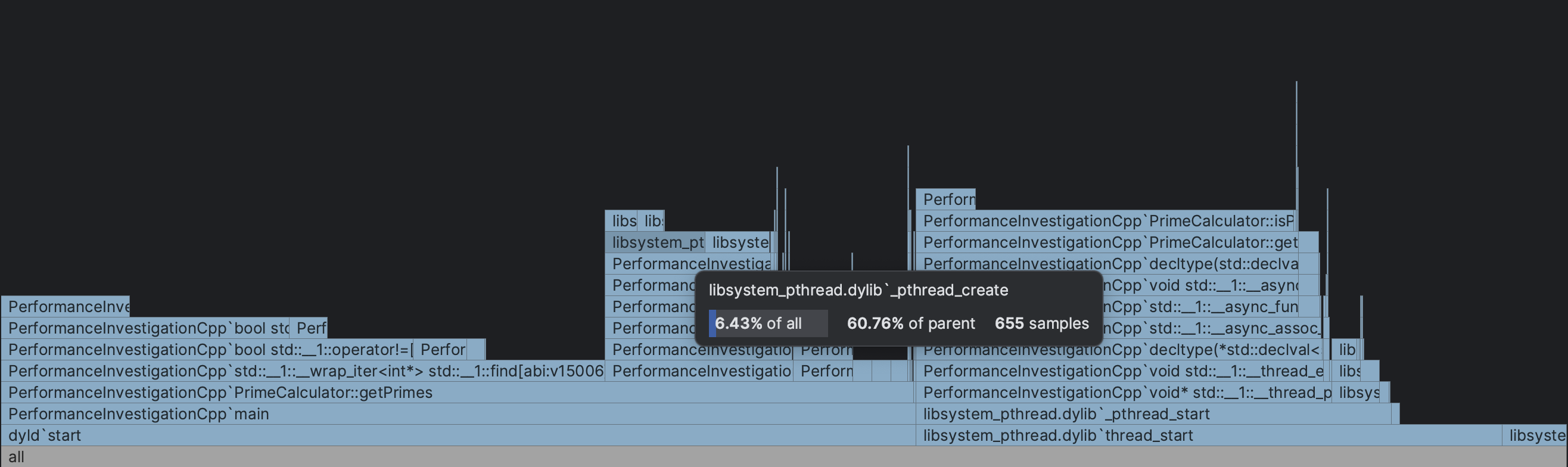
Let’s profile our new version to see a bottleneck:

maxPrime = 1000

A screen shot of a computer

Description automatically generated

maxPrime.= 100000



Seems like **threads creation overhead** is significant on lower maxPrime and **inefficient algorithms** is taking over on higher maxPrime.

Interestingly, **lock (mutex) impact is not visible yet** as the entire process takes much more time on iterating rather than injecting found numbers:

A screenshot of a computer

Description automatically generated

To summarize, seems like these are the main inefficiencies that are impacting overall performance at that point:

1. **Spawning too many threads / lack of work division**: a new thread is created to check multiples of a single numbers instead of dividing work more efficiently (for vector segments);
2. **Iterating over non-primes**: the code checked each candidate number against all numbers less than itself, rather than only known primes;

Instead of trying to optimize algorithms and threading approach at once, let’s first try to implement more efficient single-threaded solution.

# Single-threaded solution

## Naïve solution (v3)

Let’s start from implementing naive single-threaded solution without significant changes to the algorithm itself.

Identifying numbers that are non-primes is done withing single thread (implementation **PerformanceInvestigationCpp\_v3**):

for (int candidate: primeNumbers) {  
 if (!isPrime(primeNumbers, candidate)) {  
 primeNumbersToRemove.push\_back(candidate);  
 }  
}

Benchmark comparison (**v2 vs v3**):

Single threaded version appears to be **~20% faster** with maxPrime = 100k and consumes significantly less memory (**1.6Mb vs 20Mb**).

A graph of a number of data

Description automatically generated with medium confidence

Let’s now profile naïve single-threaded solution to identify next bottleneck (maxPrime = 100k):

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Seems like **iterating over all numbers (not only already known primes)** is taking too much time.

## Basic Eratosthenes Sieve (v4)

To mitigate that problem let’s use a well-known algorithm called the Sieve of Eratosthenes.

It implies multiple optimizations in comparison to a naïve implementation.

Store numbers in bool vector instead of int, where index represents a number (memory efficiency):

std::vector<bool> isPrime(maxPrime + 1L, true);

It iterates over a bool vector and in case it finds a prime number, it eliminates its multiples so there’s no need to check divisibility of a number explicitly:

for (int candidate = 2; candidate <= sqrtMaxPrime; ++candidate) {  
 if (isPrime[candidate]) {  
 for (long primeMultiple = candidate \* candidate; primeMultiple <= maxPrime; primeMultiple += candidate) {  
 isPrime[primeMultiple] = false;  
 }  
 }  
}

Implementation labeled as **PerformanceInvestigationCpp\_v4.**

Benchmark comparison (**v3 vs v4**):

V4 started to take over from maxPrime = 100k and showed extreme advantage at maxPrime = 500k (**1Mb vs 6Mb** & **0.04s vs 96s**)

A graph of different types of data

Description automatically generated with medium confidence

## Eratosthenes Sieve odd numbers (v5)

However, there’s still a space for improvement. Let’s see a profiling result:

A screenshot of a computer

Description automatically generated

A loop that eliminates multiples is taking major part of a processing time:

for (long primeMultiple = candidate \* candidate; primeMultiple <= maxPrime; primeMultiple += candidate) {  
 isPrime[primeMultiple] = false;  
}

We can iterate over odd numbers only (as all even number are divisible by 2):

// Starting from 3 and incrementing by 2 to check odd numbers only  
for (int candidate = 3; candidate <= sqrtMaxPrime; candidate += 2) {  
 if (isPrime[candidate]) {  
 for (long primeMultiple = candidate \* candidate; primeMultiple <= maxPrime; primeMultiple += candidate) {  
 isPrime[primeMultiple] = false;  
 }  
 }  
}

A screenshot of a computer

Description automatically generated.push\_back() of identified prime numbers to a int vector doesn’t seem to take much time, but let’s also try to reserve a memory to prevent vector resizing:

// could be also maxPrime / log(maxPrime) as per the prime number theorem (PNT)  
primeNumbers.reserve(maxPrime / 2);

Optimized solution is labeled as **PerformanceInvestigationCpp\_v5.**

Benchmark comparison (**v4 vs v5**):

Both execution time and memory consumption decreased: **4.5s vs 3.3s & 67Mb vs 35Mb** (for maxPrime = 100m)

A screenshot of a graph

Description automatically generated

# Multi-threaded solution

// Add the primes in this segment to the shared list  
for (int i = std::max(2, startSegment); i <= endSegment; ++i) {  
 if (isPrime[i - startSegment]) {  
 // Lock the mutex to protect access to the shared vector  
 std::unique\_lock<std::mutex> lock(mtx);  
 primes.push\_back(i);  
 }  
}

Lock is a problem

A screenshot of a computer

Description automatically generated