Performance issues

Table of Contents

[Issues presented in original solution 1](#_Toc150274485)

[Issue: *BigIntegerIterator / myFiller* are completely useless and introduce an excessive memory consumption as well as additional computation 1](#_Toc150274486)

[Issue: Utilizing *std::exception()* to signify number divisibility 3](#_Toc150274487)

[Issue: iterating over all numbers to check divisibility 4](#_Toc150274488)

[Issue: erasing numbers from vector instead of pushing – find required 4](#_Toc150274489)

[Issue: utilizing vector<int> - memory footprint 7](#_Toc150274490)

[Issue: inefficient way to identify primality of a number 8](#_Toc150274491)

[Other issues 8](#_Toc150274492)

# Issues presented in original solution

## Issue: *BigIntegerIterator / myFiller* are completely useless and introduce an excessive memory consumption as well as additional computation

Detailed description:

Additional `myFiller` vector is used containing `BigIntegerIterator` objects:

std::vector<BigIntegerIterator> myFiller;

// BigIntegerIterator is not needed, we can push\_back() right into primeNumbers

int i = 2;  
for (int j = 0; j < maxPrime; ++j) {  
 myFiller.push\_back(BigIntegerIterator(i++));  
}

Each `BigIntegerIterator` object utilizes two additional vectors instead of storing a single int value:

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

Besides, a size equal to 500 items is reserved for each vector (e.g. for each element in `myFiller` vector):

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

We can skip `myFiller` completely and push numbers right into `primeNumbers` vector instead of doing it following way:

for (int j = 0; j < maxPrime; ++j) {  
 myFiller.push\_back(BigIntegerIterator(i++));  
}  
// primeNumbers memory should be preallocated using .reserve() as we know N preemptively  
for (auto integer: myFiller) {  
 primeNumbers.push\_back(integer.getContain());  
}

Additionally, it introduces unnecessary `max()` function invocation:

int getContain() {  
 return std::max(std::stoi(contain.at(0)), reference.at(0));  
}

and as we’re pushing numbers to `primeNumbers` vector it may resize occasionally so it makes sense to do `.[reserve](https://en.cppreference.com/w/cpp/container/vector/reserve)()`.

## Issue: Utilizing *std::exception()* to signify number divisibility

Detailed description:

Whenever a check for `isPrime()` happens in

try {  
 isPrime(primeNumbers, candidate);  
}

`isPrime()` func needs to throw a new exception:

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

It appeared to be insignificant on a benchmark scale of high max prime numbers (see v1 vs v2 comparison in Optimization.docx report), however it’s a bad approach by design – see section 5.4.2 of a [Technical Report on C++ Performance](https://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf).

Instead, simple Boolean conditions should be used:

`isPrime()`:

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

check:

if (!isPrime(primeNumbers, candidate))

## Issue: iterating over all numbers to check divisibility

Note: We’ll omit mathematical proof of why a candidate number should be only checked for divisibility only up to its square root.

Instead of iterating over all numbers up to a `candidate` within `.isPrime()` func:

for (int j = 0; j < candidate - 2; ++j)

we can do following:

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

## Issue: erasing numbers from vector instead of pushing – find required

Kind of inverted logic is implemented in algorithm as it’s finding non-prime numbers and is adding them to `primeNumbersToRemove` vector:

primeNumbersToRemove.push\_back(candidate);

because of that, later it needs to find matching number in initial `primeNumbers` vector and erase it:

for (int toRemove: primeNumbersToRemove) {  
 primeNumbers.erase(std::find(primeNumbers.begin(), primeNumbers.end(), toRemove));  
}

A screenshot of a computer

Description automatically generated`.find()` bottleneck:

A screenshot of a computer

Description automatically generated`.erase()`:

Instead, we can push numbers directly to some new resulting vector that will store prime numbers only.

Initial vector to store all numbers:

std::vector<int> numbers;

All threads are pushing identified prime numbers to an additional vector (which obviously increases memory footprint):

std::mutex primeNumbersMutex;  
std::vector<int> primeNumbers;  
std::vector<std::future<void>> futures;  
for (int candidate: numbers) {  
 auto future = std::async([candidate, &numbers, &primeNumbers, &primeNumbersMutex]() {  
 if (isPrime(primeNumbers, candidate)) {  
 std::unique\_lock lock(primeNumbersMutex);  
 primeNumbers.push\_back(candidate);  
 }  
 });  
 futures.push\_back(std::move(future));  
}

Additional sorting is also required, but it’s complexity is neglectable compared to the algorithms itself:

std::sort(primeNumbers.begin(), primeNumbers.end());

A screenshot of a computer

Description automatically generatedThe bottleneck here seems to be inefficient threads work division and spawning too many threads:

Lock is not yet playing a significant role:

A screenshot of a computer

Description automatically generated

At that point we can either try to implement efficient single-threaded algorithm (Sieve of Eratosthenes) or try to divide threads work division with more efficient algorithm implementation right away (segmented Sieve of Eratosthenes).  
The idea behind more efficient ss to increase a granularity of work as each thread was given a very small unit of work (checking the primality of a single number).

Both options are described further in Optimization.docx report.

## Issue: utilizing vector<int> - memory footprint

There’re more optimal structures that can be used instead of *vector<int>:*

std::vector<int> primeNumbers;

[*vector<bool>*](https://en.cppreference.com/w/cpp/container/vector_bool)is optimized in a way to use 1 bit per value and we can use it:

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

index of an element will represent the number while bool value will indicate whether number is a prime.

## Issue: inefficient way to identify primality of a number

Instead of doing a check that number is not divisible by any number lower than its square root, we can check it against only already known prime numbers. What is moreover, it turns out that if numbers wasn’t marked as non-prime while checking previously identified prime numbers, it means it is a prime. Thus, we can simply iterate over *vector<bool>* and cross-out multiples of known primes starting from 3 (skipping even numbers as they are non-primes by default):

for (int candidate = 3; candidate <= sqrtMaxPrime; candidate += 2) {  
 if (isPrime[candidate]) {  
 for (long primeMultiple = candidate \* candidate; primeMultiple <= maxPrime; primeMultiple += candidate) {  
 isPrime[primeMultiple] = false;  
 }  
 }  
}

which is basically an implementation of Sieve of Eratosthenes.

# Other issues

Apart from issues already identified in original solution, there’re multiple minor optimizations that should be considered:

1. Utilizing bitwise operators instead of arithmetic for int [`/`, `%`, `\*`];
2. Comprehensive segmentation with custom structures to avoid race conditions and a use of lock mechanism.
3. Efficient work division (segmentation) between threads;

All these optimizations are applied in further versions implementations (single-threaded v5, multi-threaded v6, v7, v8).