EXAMENSARBETE INOM TEKNIK,
GRUNDNIVÅ, 15 HP

STOCKHOLM, *SVERIGE 2017*

Trial Division Improvments and Implementations

Trial Division
Förbättringar och Implementationer

FELIX HEDENSTRÖM

KTH
SKOLAN FÖR DATAVETENSKAP OCH KOMMUNIKATION

Abstract

Trial division is possibly the simplest algorithm for factoring numbers. The problem with Trial division is that it is slow and wastes computational time on unnecessary tests of division. How can this simple algorithms be sped up while still being serial? How does this algorithm behave when parallelized? Can a superior serial and a parallel version be combined into an even more powerful algorithm?

To answer these questions the basics of trial divisions where researched and improvements where suggested. These improvements where later implemented and tested by measuring the time it took to factorize a given number.

A version using a list of primes and multiple threads turned out to be the fastest for numbers larger than 10^{10}, but was beaten when factoring lower numbers by its serial counterpart. A problem was detected that caused the parallel versions to have long allocation times which slowed them down, but this did not hinder them much.

Sammandrag

Trial division är en av de enklaste algoritmerna när det kommer till att faktorisera tal. Problemet med trial division är att det är relativt långsamt och att det gör onödiga beräkningar. Hur kan man göra denna algoritm snabbare utan att inte göra den seriell? Hur beter sig algoritmen när den är parallelliserad? Kan en förbättrad seriell sedan bli parallelliserad?

För att besvara dessa frågor studerades trial division och dess möjliga förbättringar. Dessa olika förbättringar implementerades i form av flera funktioner som sedan och testades mot varandra.

Den snabbaste versionen byggde på att använda en lista utav primtal och trådar för att minimera antal 'trials' samt att dela upp arbetet. Den var dock inte alltid snabbast, då den seriella versionen som också använde en lista av primtal var snabbare för siffror under 10^{10}. Sent upptäcktes ett re-allokeringsproblem med de parallella implementationerna, men eftersom de ändå var snabbare fixades inte detta problemet.

Contents

1 Introduction

RSA Laboratories announced a reward of 200000$ for anyone able to factor a number known as RSA-2048, a number with 617 decimal digits. As of February 2017 this number as well as RSA-1536, RSA-1024, RSA-896, and RSA-704 are not factored. The challenge is no longer active since 2007. RSA Laboratories claim that the best algorithm for factoring large numbers is using General Number Field Sieve (GNFS).[1]

SSL encryption is used for TCP/IP connections all over the world is only as secure as RSA encryption. RSA encryption can be broken by factoring integers. This means that a lot of internet security rely on fast integer factorization being impossible. If such an algorithm was to be found, encryption as we know it would need to be rethought.[2]

Trial division is an algorithm that does not require a very advanced understanding of math to comprehend, and has a lot of depth and potential for improvement. By diving into the depth of how trial division really works we might get a better understanding of what ways it can be improved and estimate the impact of these improvements.

1.1 Problem statement

Trial division is a simple algorithm and by its nature wasteful of computational time. But by its design it is also easy to run in parallel, since its future calculations are not based on past ones. Is there also a way to minimize the amount of numbers tested? If it is known that the number is not divisible by 2, it is foolish to test divisibility by 4 and so on. Is it possible to stop these unnecessary division tests and will doing so always speed up the factoring time? How does parallelizing the algorithms affect speedup? Is it worth the time to create a parallel version?

1.2 Scope

This study will explore the possible improvements of trial division in the forms of parallel implementations as well as using a list of possible divisors as well as other small improvements. No other forms of improvement will be tested.

All algorithms will be implemented in order to assure that they operate similarly and use the same types of libraries. This also allow time testing to be standardized as all implementations will be made using the same language.

No other factor algorithm other than trial division will be tested, as this is a study of how the different improvements effect Trial division in particular.

1.3 Purpose

The purpose of this paper is to present proposed improvements to trial division as well as test them to evaluate their feasibility, both in terms of potential speedup and in terms of difficulty of implementation.

2 Background

This section will cover basic definitions used in this paper, the basics of integer factorization, the Sieve of Eratosthenes - a simple prime number list generator, and Trial Division and its possible improvements. It will also briefly describe the most used factoring algorithm, GNFS.

2.1 Definitions

$\mathbb{N}$**:** The set of all non negative integers $0, 1, 2, ...$

$\mathbb{Z}$**:** The set of all integers $..., -2, -1, 0, 1, 2, ...$

x **mod** n**:**
The 'modulo' operator, sometimes denoted as x % n in programming, calculates the remainder of division between x and n. $r = x$ mod n where r satisfies

$$q \in \mathbb{Z}$$

$$x = nq + r$$

$$|r| < |n|$$

$x \equiv y$ **mod** n**:**
x and y is in the same equivalence class. It is equivalent to x mod $n =$ y mod n.

$\gcd(x, y)$**:**
Greatest common divisor of x and y. $gcd(10, 15) = 5$ because 5 is the highest number that can divide both 10 and 15.

$O(f(n))$**:**
Describes the speed of an algorithm by stating that the algorithm will always use a number of steps that is proportional to $f(n)$. Assume the real number of steps used by algorithm for a input of size n is $T(n)$. $O(f(n))$ means that there is an m and an n_0 that satisfies $T(n) \leq Mf(n)$ for all $n \geq n_0$.

$\pi(x)$**:**
The number of primes lower than $x \in \mathbb{N}$.

2.2 Integer factorization

The report "Modern Factoring Algorithms" defines integer factorization as:

> Given a composite integer N, find a non-trivial factor e of N, i.e. find e such that $e|N$.

Integer factorization is one of mathematics oldest problems. Algorithms like the sieve of Eratosthenes are still used and studied today.[3]

This problem stems from one of the fundamental theorems of arithmetic that states that every positive integer can be written uniquely as a product of primes. Knowing that such a sequence is possible for any given number will lead the curious to find a method that produces the sequence.

It is interesting to note that the maximum number of factors for a given number is number in the form 2^k, which results in the fact that all numbers $n \in \mathbb{N}$ are limited to a maximum of $log_2\ (n)$ number of factors.

2.3 Sieve of Eratosthenes

The Sieve of Eratosthenes is an algorithm that generates prime numbers. It works by generating a list of all integers from 2 to the highest number that should be included, n. Look at the first number in the list, in this case 2. Cross out all multiples of 2 except 2 itself, this means crossing out all numbers $2i$ where $2i \leq n$ where $i > 1$. When all multiples of 2 are crossed out, go to the next number that is not crossed out, in this case 3. Repeat the same strategy - cross out all numbers $3i$ where $3i \leq n$ and $i > 1$. Repeat this until you have gone through all numbers in the list. All numbers that are not crossed out are prime.[4]

2.4 Trial Division

It is possible to find all possible divisors of N by simply checking each possible divisor d, and checking if $d|N$. This algorithm is sometimes called *trial division* and has a time complexity of $O(\sqrt{N})$.[2]

The algorithms checks all possible divisors, by going through all numbers $n \in \mathbb{N}$ that satisfy

$$2 \leq n \leq \sqrt{N} \tag{1}$$

for a total span of

$$|\{\ n \mid 2 \leq n \leq \sqrt{N}, n \in \mathbb{N}\}| = \sqrt{N} - 2 \tag{2}$$

By dividing N with the factors found it is made sure that no non-prime numbers are found. If N is not divided by the found factors, false 'prime' factors might appear. Example:

1. Factor the number 81 using trial division
2. Search the span $2 \leq n \leq 9$ for factors
3. $2 \not| N \rightarrow 2$ is not a factor
4. $3|N \rightarrow 3$ is a factor
5. $c \not| N, 4 \leq c \leq 8 \rightarrow$ 4-8 is not a factor
6. $9|N \rightarrow 9$ is a factor - OBS non-prime factor found

In this example 9 was found as a factor even though it is the same as 3^2 and not a prime. A solution to this problem can be seen in Algorithm 1 by removing

every found factor from the number.

Algorithm 1: Trial division without any improvements removing factors as they are found

```
   Data: N - Number to be factorized
   Result: list_of_factors - List of factors of N
 1 list_of_factors[] = [];
 2 for i = 2; i ≤ √N; i++ do
 3 |   while N mod i == 0 do
 4 |   |   N = N/i;
 5 |   |   list_of_factors.append(i);
 6 |   end
 7 end
 8 if N ≠ 1 then
 9 |   list_of_factors.append(N);
10 end
11 return list_of_factors
```

2.5 Improved trial division

If division by 2 is checked before going through possible divisors, the search interval is nearly halved as checking divisibility by even numbers is no longer required. This improvement changes the results of equations 1 and 2. It is no longer needed to start on 2 since it has been deduced whether or not 2 is a factor, and checking divisibility by even numbers is no longer necessary. The search space for the improved algorithm can now be described as:

$$3 \leq n \leq \sqrt{N} \quad \forall n \in \mathbb{N}(n \ mod \ 2 == 1) \tag{3}$$

and the span from the largest to the smallest number is

$$|\{\ n \mid 3 \leq n \leq \sqrt{N}, n \in \mathbb{N}\}| = \sqrt{N} - 3 \tag{4}$$

Note that equation 4 refers to the span, not the number of factors checked. The amount of factors checked is closer to half of $\sqrt{N} - 3$.

If division by 2 is checked beforehand, it is no longer necessary to start checking for divisors starting at 2 as seen on line 2 in Algorithm 1, as it is more efficient to instead start checking for divisors at `i` = 3. Another change to the algorithm concerns the incrementation of `i`. Instead of incrementing using `i++` as seen on line 2 in Algorithm 1 we instead increments using `i+` = 2.

2.5.1 Using a list of primes

Another possible solution is to generate a list of primes from 2 to $\sqrt{N}$ and use them as candidates, this way the 'false factors' described in section 2.4 are no longer a problem.

The speed of normal trialdivision was $O(\sqrt{N})$ because the algorithm went through exactly $\sqrt{N}$ candidates for division as described in 2.4 Trial Division.[2] Describing the time complexity for trial division using a list of pre-calculated primes is a little harder to determine.

The book '*The distribution of prime numbers*' written by *A. E. Ingham* describes the useful function $\pi(x)$ - a function that determines the number of primes below x. It also suggests a method of approximating it [5]

$$\pi(x) \approx \frac{x}{log\ x} \tag{5}$$

If a list of primes are used, we can assume as we did in section 2.4 Trial Division that we only need to test numbers that fit withing the span described by equation 1. This is equivalent to saying that we need to check divisibility $\pi\left(\sqrt{N}\right)$ numbers. The algorithm is therefore of time complexity $O\left(\pi\left(\sqrt{N}\right)\right)$ which is equivalent to

$$O\left(\frac{\sqrt{N}}{log\ \sqrt{N}}\right) \tag{6}$$

by inserting $\sqrt{N}$ into equation 5.

2.6 Parallel trial division

Because the workload in trial division can be so easily divided, parallel implementations of trial division achieves a linear speedup by dividing the workload into equal pieces.[6]

The improved version of trial division described in 2.5 Improved trial division can be used as the basis for a parallelized version. In this case the search span of the improved version of trial division, described in equation 4, will also be the span of the parallelized version.

If we assume k number of parallel threads, and that each thread should search through an equal sized space the span of the search space can be described as the total space seen in equation 4, divided by the number of threads k.

$$threadSpan = \frac{\sqrt{N} - 3}{k} \tag{7}$$

The exact span of thread number i can now be described similarly to how the spans where described in equations 1 and 3 with the size from equation 7.

$$\begin{aligned} 3 + threadSpan * i \leq n \leq 3 + threadSpan * (i + 1) \\ \forall n \in \mathbb{N}(n\ mod\ 2 == 1) \end{aligned} \tag{8}$$

2.6.1 Factors found by multiple threads

It was mentioned in section 2.4 that unless factors that are found are removed from N by dividing N with the factor as discussed in 2.4 Trial Division, false factors or repeat factors could be found.

Three ways to solve this problem is described in figure 1.

2.7 General Number Field Sieve (GNFS)

GNFS or General Number Field Sieve is the fastest known factoring algorithm. It has a time complexity of

$$O\left(exp\left[\left(\frac{64}{9}N\right)^{1/3}(log\ N))^{2/3}\right]\right)$$

1. Only check prime divisors
2. Share the number N between threads, as well as its updated values after factors are found
3. Check the factors found by each thread for false factors. This in turn can be done in two ways:
   (a) Check the primality of all factors found. Non-primes cannot be primefactors
   (b) Make sure no factor found is a multiple of another factor

Figure 1: Possible solutions to the problems posed in section 2.6.1 Factors found by multiple threads

to find the prime factors of a number with $n \in \mathbb{N}$ decimal digits. [2]

To find the factors of n using GNFS one first find two integers x and y that satisfy

$$x^2 \equiv y^2 \bmod n \tag{9}$$

as well as $x \not\equiv \pm y \bmod n$. If integers x and y are found, then $gcd(x-y, n)$ must be a non trivial divisor of n. [7]

3 Method

The method section will describe how the tools to implement and test trial division was chosen. It contains explanations for the choices that were made, as there are many possible ways to implement and measure Trial Division.

3.1 Language

To implement threaded trial division, a language with thread capabilities is needed. Since speed is an important aspect, a compiled language is preferred. The language also needs support for large number arithmetic as well a good way to standardized testing of algorithms.

This leaves basically two popular alternatives: *C* and *C++*, since they both are compiled and have support for the *GMP* library described in 3.2 Libraries and classes. *C++* was chosen because of its added class functionality as well as the availability of libraries.

3.2 Libraries and classes

GMP:
GMP is a free library that allows for fast large integer computing. Since this study intends to compare speeds of algorithms that often cover large integers, the GMP library is needed. GMP allocates more memory if the number is getting too large, to ensure that the actual number can continue to grow theoretically infinite. The reallocation does however take a lot of time.[8]

GMP is licensed under GNU LGPL v3 as well as GNU GPL v2. As a result, all resulting code created will also be free and can be found at [9]. [8]
Thread:
The thread class allows for simple threading in *C++*. Each thread object represents a separate computation in a program, but threads can share address spaces and by extension data.[10, 11] Sharing data and allowing for concurrency will be important when implementing parallel versions of trial division.
Mutex:
Mutex, Mutual Exclusion, controls access to data shared between threads. This combats data races; when two threads try to access the same address at the same time. By locking the data with Mutex, no other threads can access its data. When unlocked, the data is available again. [10, 11]
Chrono:
Chrono was designed with compatibility with multiple systems in mind. It has a simple yet flexible interface for measuring timepoints and durations.[11]

3.3 Variations of the algorithm

In order to test algorithms, they must first be constructed. The following algorithms were to be implemented:
Serial Trial division - Simple serial trial division with only the minimal improvements described in section 2.5 Improved trial division.
Parallel trial division - Threaded trial division described in section 2.6 Parallel trial division.

Serial Trial division using a list of primes - Using a list of primes as candidates for division should be faster as described in section 2.5 Improved trial division.
Parallel trial division using a list of primes - This solution was briefly mentioned in step 1 in figure 1. This implementation will be largely based on the two previous ones and should be saved for last, as a lot of its implementation will be similar from the aforementioned algorithms.
Sieve of Eratosthenes - To generate the list of primes needed the Sieve of Eratosthenes mentioned in section 2.3 can be used. Saving the output ensures that the algorithm only needs to be run once in order for the serial and parallel versions of trial division that require the list of primes to work.

3.4 Hardware

The tests where made using an AMD FX-8350 Eight-Core @ 4.014GHz 64-bit and 4 4096GB DDR3 Corsair CMZ8GX3M2A1866C9 memory and a KINGSTON SH103S3 120GB SSD.

The computer used for testing also used Debian 8.8 jessie running the x86_64 Linux 3.16.0-4-amd64 kernel.

Only 6 threads were used to make sure that some threads where unused by the program and available for the OS to use.

3.5 Testing

Bash allows for output manipulation as well as the handling of executables in an great environment. Using it to control the executables that measures time will allow for an easier time when it comes to collecting all results and transforming them to more readable mediums, such as plots and tables.

Keeping the time within *C++* gives should create an accurate measurement since it minimized overhead. To minimize the difference when testing different functions the function that controls time measurement should be the same for all functions. This can be accomplished by having the function that should be tested as an input for a universal timetesting function.

To test the runtime of an implemented algorithms function with the given input N the steps described in figure 2 should be be taken. Step 1 is taken to make sure initialization times are kept to a minimum in order to ensure more accurate testing. To test the time it takes for an algorithm to factorize every number in a given range we iterate through every number and run the steps shown in figure 2 once per number.

1. Run the algorithm once
2. Start the clock
3. Run the algorithm *rep* number of times
4. Stop the clock
5. Divide the measured time by *rep* to get the average runtime of the algorithm.

Figure 2: Steps taken to measure the time it takes for a given algorithm and a given number N

4 Result

4.1 Time measurement

The figures 3, 4, and 5 where all measured in the same multiple hour session. T he number of repetitions mentioned in fig 7 argument 4 as well as in fig 2 step 3 were set to 1, because of time limitations.

The data for figure 6 was generated at a later date, and measured much smaller spans of numbers distributed over a wider range of numbers, to provide timed data when factoring larger numbers.

4.2 Timekeeping with *timetest*

The time measuring tool was given the fitting name *timetest*. It worked in conjunction with a *Bash*-file named *result.sh* described in more detail in section 4.3 Bash - *result* and *result2*. The executable was terminal based and required 5 arguments described in figure 7 to function. The executable goes through all number between and including the start and end value (argument 2 and 3) and factorizes them a number of times equal to the 4 argument number of times as well as an extra time. The one extra time is to initiate and allocate size for all needed variables. *timetest* also prints a total of times equal to the argument 5.

An example of using *timetest* can be found in appendix C and an implementation of the polymorphic algorithm timekeeper can be found in appendix A.

4.3 Bash - *result* and *result2*

The *Bash*-file had an array containing all available algorithms, as well as variables describing each and every argument mentioned in figure 7.

result2 was made at a later date in order to generate more data. It was used to generate figure 6 and works similar to *result*. It works by only testing the first 10000 numbers off a multiple of 10. This allows for fast testing of multiple magnitudes of ten.

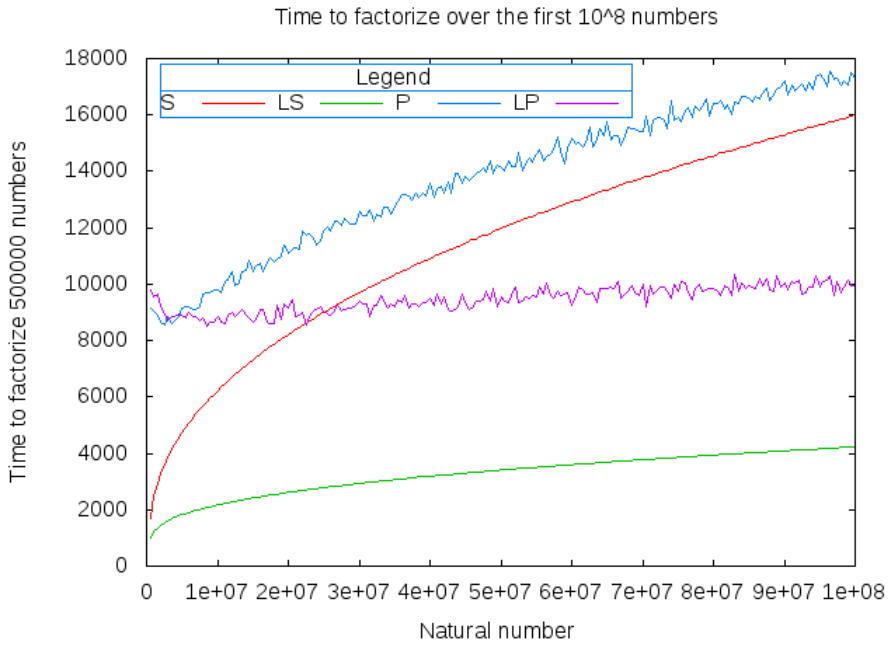

Figure 3: All algorithms factoring the numbers 0 to 10^8 with 500000 number between each time measurement

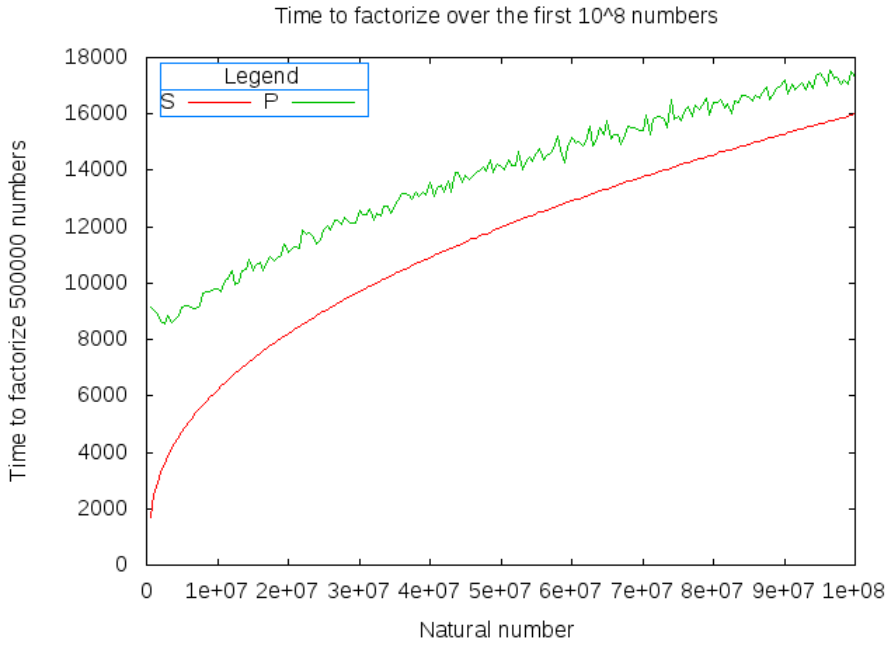

Figure 4: Serial and parallel trial division factoring the numbers 0 to 10^8 with 500000 number between each time measurement

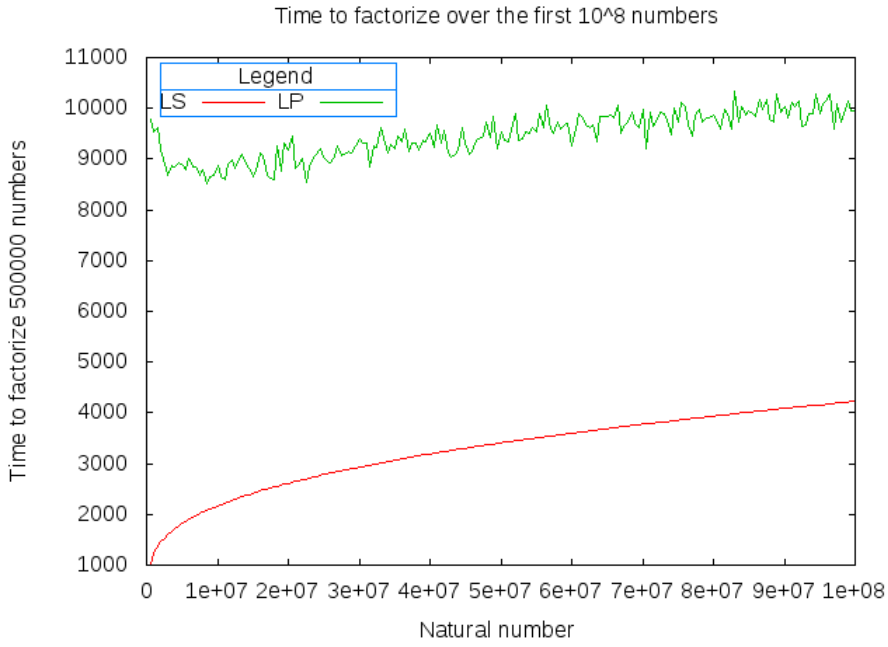

Figure 5: Serial and parallel trial division using lists factoring the numbers 0 to 10^8 with 500000 number between each time measurement

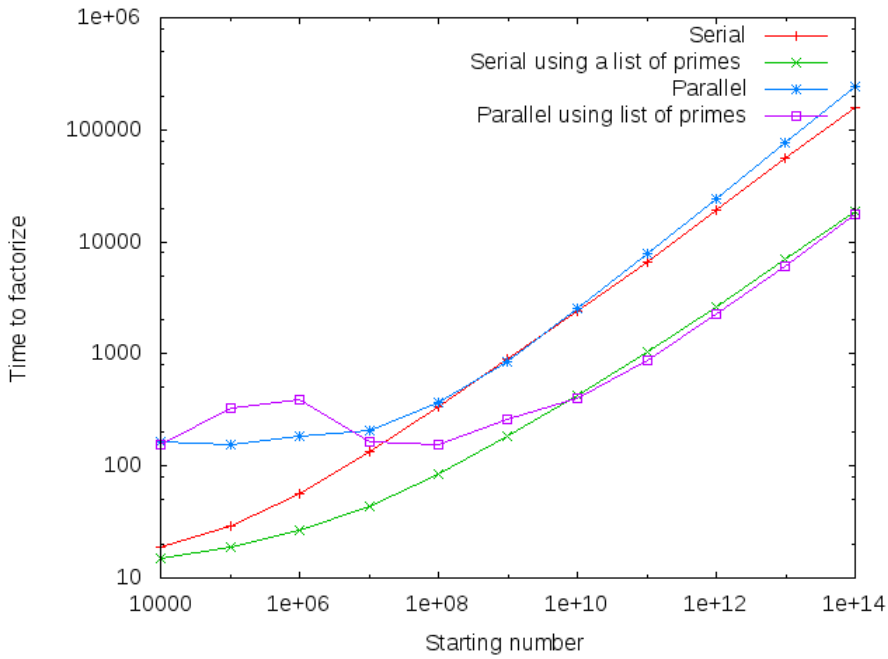

Figure 6: Factoring the numbers $10^k \leq n \leq 10^k + 10000$ for every $k \in \mathbb{N}, 4 \leq k \leq 14$. Note that the graph displayed has both axes log scaled

./timetest [1] [2] [3] [4] [5]

1. Algorithm that will be timed and tested - The options are T (serial trial division) and *NM*
2. First number that should be factorized - This number has practically no size limitation due to the *GMP* library
3. Last number that should be factorized - This number has practically no size limitation due to the *GMP* library
4. Number of times the algorithm should prime factorize each number - Bounded by *unsigned long long* or *std::stol*, whichever is smallest.
5. Number of datapoints that will be output - Bounded by *unsigned long long* or *std::stol*, whichever is smallest.

Figure 7: The 5 arguments needed by the *timetest* function

4.4 Serial trial division

Algorithm 2: Trial division with halved search span compared to algorithm 1.

Data: `N` - Number to be factorized
Result: `list_of_factors` - List of factors of N

```
1  list_of_factors[] = [];
2  while N mod 2 == 0 do
3  |  N = N/2;
4  |  list_of_factors.append(2);
5  end
6  for i = 3; i ≤ √N; i+ = 2 do
7  |  while N mod i == 0 do
8  |  |  N = N/i;
9  |  |  list_of_factors.append(i);
10 |  end
11 end
12 if N ≠ 1 then
13 |  list_of_factors.append(N);
14 end
15 return list_of_factors
```

4.5 Parallel trial division

Algorithm 3: Parallelized trial division

Data: `N` - Number to be factorized
`nr_of_threads` - Number of threads
Result: `list_of_factors` - List of factors of N

```
1  list_of_factors[] = [];
2  while N mod 2 == 0 do
3  |  N = N/2;
4  |  list_of_factors.append(2);
5  end
6  threadSearchSpace = (√N − 3)/number_of_threads;
7  threadList[nr_of_threads];
8  threadFactors[nr_of_threads][];
9  for i = 0; i < nr_of_threads; i++ do
10 |  lowerLimit = 3 + threadSearchSpace ∗ i;
11 |  upperLimit = 3 + threadSearchSpace ∗ (i + 1);
12 |  threadList[i] =
   |  new thread(threadTrial, N, lowerLimit, threadFactors[i]);
13 end
14 for i = 0; i < nr_of_threads; i++ do
15 |  threadList[i].join();
16 |  combine(list_of_factors, threadFactors[i], N);
17 end
18 if N ≠ 1 then
19 |  list_of_factors.append(N);
20 end
21 return list_of_factors
```

Each thread in Algorithm 3 runs *threadTrial* which is described by Algorithm 6 in Appendix B. Algorithm 6 is heavily based upon Algorithm 2 with a few main differences. *threadTrial* needs to have a start as well as an end-point. The span of each instance of *threadTrial* is decided in Algorithm 3 on lines 10 and 11 by the equations 7 and 8.

Important to note is that the list used within *threadTrial*, the input list named *factorList*, is a reference to one of the sublists in the main algorithm - the `threadFactors` list declared on line 8 in Algorithm 3.

4.6 Serial with list of primes

Described in algorithm 4, the version of serial trial division that uses a precalculated list of primes is very similar to the one without the list described in section 4.4 Serial trial division. How the list is generated is described in section 4.8 Sieve of Eratosthenes.

Important to note is the variable `LARGEST_PRIME` on line 12 which denotes the largest prime in the list. It is used to determine if all factors were found or if more searching is required. If N is larger than `LARGEST_PRIME` after all found factors were removed from it, more searching is required. If it is smaller, all factors have been found.

If not all factors are found, normal trial division is done to rule out remaining

possible factors. *spanSerialFactorization* on line 13, is a slightly modified version of the improved trial division described in algorithm 2 that accomplishes this. The difference is that *spanSerialFactorization* has *start* and *end* values that determines the span that should be searched. When it is called from algorithm 4 the lower bound will always be `LARGEST_PRIME`.

Algorithm 4: Serial trial division using a list of precalculated primes

```
   Data: N - Number to be factorized
   Result: list_of_factors - List of factors of N
 1 list_of_factors[] = [];
 2 list_of_primes[] = readlist();
 3 for p in list_of_primes do
 4 |   if p > √N then
 5 |   |   break;
 6 |   end
 7 |   while N mod p == 0 do
 8 |   |   list_of_factors.append(p);
 9 |   |   N/ = p;
10 |   end
11 end
12 if N > LARGEST_PRIME then
13 |   new_factors[] =
   |   spanSerialFactorization(N, LARGEST_PRIME, sqrtN);
14 |   list_of_factors.append(new_factors);
15 |   return list_of_factors
16 else if N ≠ 1 then
17 |   list_of_factors.append(N);
18 end
19 return list_of_factors
```

4.7 Parallel with list of primes

Parallel trial division using a list of precalculated primes is very much a mix of algorithms 3 and 4. Instead of every thread being given a search interval like described in algorithm 6, every thread gets an equal number of primes that should be tried.

To To solve this a binary search is made in the list of primes to determine what the largest prime that should be tested is. This is looking for largest possible index `a` that satisfies `list_of_primes`[a] $\leq \sqrt{\texttt{N}}$. Every thread then gets a search span of size $\frac{a}{k}$ where k is the number of threads.

4.8 Sieve of Eratosthenes

The Sieve of Eratosthenes was only implemented to supply a list of prime to the algorithms described in sections 4.6 Serial with list of primes and 4.7 Parallel with list of primes. With that in mind speed was not a concern as it only had to run once to determine the primes. In the timetests the other algorithms used a

list of primes from 2 to 10^8. The actual runtime of the sieve for the given input 10^8 was not measured as it has no relevance to the study.

The actually used version is very similar to the description given for algorithm 5 except it opened and printed directly to a file and was an independent executable, not just a function.
Note that the mysterious function '*setAllTrue*' on line 2 in algorithm 5 is not a

Algorithm 5: Implementation of the Sieve of Eratosthenes described in section 2.3 Sieve of Eratosthenes

Data: `x` - Number to find all primes below
Result: `list_of_numbers` - Array of booleans of size `x`. For simplicity sake it is 1 indexed so that `list_of_numbers[i] = true` means that `i` is prime, `list_of_numbers[i] = false` means that `i` is not.

```
1  list_of_numbers[x];
2  setAllTrue(list_of_numbers);
3  list_of_numbers[1] = false;
4  for i = 2; i ≤ N; i++ do
5  |  if list_of_numbers[i] = true then
6  |  |  for j = 2 * i; j ≤ N; j+ = i do
7  |  |  |  list_of_numbers[j] = false;
8  |  |  end
9  |  end
10 end
11 return list_of_numbers
```

function that was actually used. Getting all values to be *true* was accomplished by initiating the vector in the following way:

```
std::vector<bool> isprime(N + 1,true);
```

5 Discussion

The part of the study explores the results and shortcomings of the implementations. As some of the results contradict the theory and need a plausible explanation.

5.1 Slowness of parallel versions

Both parallel versions of trial division seems to have issues with start up times, even though all threads should be pre-initialized before every run. Sadly the allocation of the *mpz_t* objects used by the *GMP* library was not pre-initialized, and their size changed with runtime as the number stored within got larger. Since every thread needed to allocate their own instances of iterating variables and the like, every thread needed to allocate and reallocate a lot of times when these numbers grew. This was briefly mentioned in section 3.2 Libraries and classes.

This would show itself as having the initiation time being equal to the initiation time for the serial version times the number of threads. Since 6 threads where used as described in section 3.4 Hardware the startup time should be roughly 6 times as large. If we look at figure 4 we can see that there is a huge difference in runtime between the algorithms in the beginning, roughly a 5 times difference when factorizing the first 500000 number, and that the gap gets smaller the higher the number go, having only about a $\frac{18000}{16000} = 1.125 = 112.5\%$ larger runtime for the parallel version.

5.2 Time complexity

As seen in figure 6 serial and parallel without list grow comparably fast. This is most likely because of the fact that the parallel versions time complexity is the serial versions time complexity divided by the number of threads. This would indicate that the growth of both algorithms would be similar as the number to be factored approaches infinity.

The same observation can be made regarding the implementations using a list of primes. The fact that they grow at a similar rate is a good indication that the implementations behave similar to the theory and are working as intended.

5.3 Fastest and slowest

Figure 6 shows that the parallel implementation that used a list of primes was shown to be the fastest, overtaking its serial counterpart when it came to factoring numbers somewhere in between 10^9 and 10^{10}.

Since the speed of the serial version of trial division using a list of prime is faster for low numbers and not far slower for the larger numbers tested, it would be the best to use if implementation time is a problem, as the time it took to implement it was comparable to serial trial division without using a list of primes. It has the best speed per implementation time of all algorithms.

It is a surprise that the parallel version without a list of primes was slower than its serial counterpart. Looking at figure 6 it seems as the parallel will never be faster than the serial, as the gap between them seemingly expands. The reason for this probably the issue discussed in 5.1.

5.4 Improvements

5.4.1 Parallel versions

If the allocation time of the numbers could be shortened large improvements to the parallel versions could be made. Any improvements to the parallel versions allocation speed could reasonably also be able to be made to the serial versions implementations, but since the serial version only declares one iterator and the parallel has one iterator per thread improvements would have a larger impact on the parallel versions.

5.5 Acknowledged problems

5.5.1 Printing

All implementations suffer from the same unfortunate problem. Printing the values of the numbers stored in *GMP*-class variables is not possible in the current implementation. All algorithms can only print numbers as large as the function *mpz_get_ui* allows, which is assumed to be the *unsigned long long* limit. The algorithms have still been tested for numbers larger than the print limit, but there is no great way to show their results.

There is surely a way to print them, and if such way reveals itself it is easily changed as the printing is handled completely separate from the rest of the algorithms.

5.5.2 Parallel using list

The parallel implementation that uses a list of primes has a problem that was sadly not prioritized due to lack of time. It can only handle numbers that are smaller than the length of the list of primes squared, which was described in section 4.8 Sieve of Eratosthenes as 10^8. One of the algorithms can only handle numbers smaller than 16 decimal digits long. This problem shouldn't be too hard to solve given a couple of days, but sadly was not completed in time.

5.6 Future work

If the allocation time problem discussed in section 5.4.1 can be minimized more threads would probably speed it up even more. Using CUDA-like technology, hundreds of threads could be used to possible give massive speed gain.

6 Conclusion

There are many ways to improve trial division. The parallel versions implemented and tested in this study had a long startup time but as the numbers to be factored grew larger the parallel versions gained upon the serial and eventually parallel trial division using a list of primes as divisors became the fastest when factoring numbers over 10^{10}.

Using a precalculated list of primes as proved to be the fastest in both the serial and parallel category. Interesting to note is the fact that the parallel version that used a list of primes didn't get faster than the serial version without primes until numbers larger than $2 * 10^7$.

References

[1] "The RSA factoring challenge," https://www.emc.com/emc-plus/rsa-labs/historical/the-rsa-factoring-challenge-faq.htm, accessed: 2017-03-05.

[2] C. Barnes, "Integer factorization algorithms," *Oregon State University*, 2004.

[3] K. Bimpikis and R. Jaiswal, "Modern factoring algorithms," *University of California, San Diego*, 2005.

[4] E. W. Weisstein, "Sieve of eratosthenes," 2004.

[5] A. E. Ingham, *The distribution of prime numbers.* Cambridge University Press, 1932, no. 30.

[6] R. P. Brent, "Parallel algorithms for integer factorisation," *Number theory and cryptography*, vol. 154, pp. 26–37, 1990.

[7] J. Buchmann, J. Loho, and J. Zayer, *An implementation of the general number field sieve.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 159–165. [Online]. Available: http://dx.doi.org/10.1007/3-540-48329-2_14

[8] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6th ed., 2015, http://gmplib.org/.

[9] F. Hedenström, "Comparison of integer factorization algorithms," https://github.com/lie94/trialdivision, 2017.

[10] "International standard ISO/IEC 14882:2014(e) – programming language C++," retrieved from https://isocpp.org/std/the-standard.

[11] N. M. Josuttis, *The C++ standard library: a tutorial and reference.* Addison-Wesley, 2012.

Appendices

A General algorithm testing

This implementation of the polymorphic timekeeping function was made before the inclusion of the *GMP*[8] library. The code supporting *GMP* is a lot harder to understand, especially without knowledge of the *GMP* documentation. The actual implementation can be found at the git [9].

```
void testAlgorithm( unsigned long long start,
                                        unsigned long long end,
                                        unsigned int reps,
                                        Algorithm a){

  unsigned long long update = ((end - start) + 1) / NR_OF_UPDATES;

  std::cerr << "Started␣testing" << std::endl;
  double elapsed_time = 0;
  // Go through every number that should be tested
  std::cerr << "+---------------+--------------------+" << std::endl;
  std::cerr << "|␣Progress\t|␣Time␣elapsed␣␣␣␣␣|" << std::endl;
  std::cerr << "+---------------+--------------------+" << std::endl;
  flush(std::cerr);
  for(unsigned long long j = start; j <= end; j++){
    // Prints an update NR_OF_UPDATES times with equal spacing
    if(update != 0 && j % update == 0){
      std::cerr << "|␣"
                        << 100 * double(j - start)/double(end)
                        << "%\t\t|␣" << elapsed_time
                        << "␣\tms␣|" <<std::endl;
    }


    std::chrono::time_point<std::chrono::system_clock> start, end;
    // Runs the algorithm once un-timed to make sure the cache
    // is more similar every run
    a(j);

    start = std::chrono::system_clock::now();
    for(unsigned long int i = 0; i < reps; i++){
      a(j);
    }
    end = std::chrono::system_clock::now();
    std::chrono::duration<double> rep_time = end - start;
    elapsed_time += (rep_time).count() * 1000;
  }
  std::cerr << "+---------------+--------------------+" << std::endl;
  std::cerr << "|␣100%\t\t|␣" << elapsed_time
  << "\tms␣|" << std::endl;
  std::cerr << "+---------------+--------------------+" << std::endl;
  flush(std::cerr);
  // Total time
  // Note that this is the only thing written to std::cout
  std::cout << elapsed_time << std::endl;
  flush(std::cout);

}
```

B *threadTrial* and *combine*

Algorithm 6: Algorithm employed by each thread in parallelized trial division, referred to by Algorithm 3 as *threadTrial*

```
   Data: N - Number to be factorized
         start - Lower limit of searchspan
         end - Upper limit of searchspan
         factorList - Empty list
 1 for int i = start; i ≤ end; i+ = 2 do
 2 |   while N mod i == 0 do
 3 |   |   N = N/i;
 4 |   |   factorList.append(i);
 5 |   end
 6 end
```

Note: The list *factorList* is linked with one of the sublists in *threadFactors[][]* found in Algorithm 3 initialized on line 8 and used on line 12. It is equally important to note that the value of `N` is NOT synchronized to its value in Algorithm 3 but is only a copy.

Algorithm 7: Combines two list of factors, referred to by Algorithm 3 as *combine*

```
   Data: foundFactors - Collection of all found factors up until now
         threadFactors - Factors found by thread
         N - Number used to keep track of found factors
 1 if threadFactors.size() == 0 then
 2 |   return;
 3 end
 4 else if foundFactors.size() == 0 then
 5 |   for int newFactor in threadFactors do
 6 |   |   foundFactors.append(newFactor);
 7 |   |   N = N/newFactor;
 8 |   end
 9 |   foundFactors = threadFactors.clone();
10 |   return;
11 end
12 int oldFoundFactors[] = foundFactors;
13 for int newFactor in threadFactors do
14 |   bool multiple = false;
15 |   for int oldFactor in oldFoundFactors do
16 |   |   if newFactor > oldFactor ∧ newFactor mod oldFactor == 0
   |   |   then
17 |   |   |   multiple = true;
18 |   |   end
19 |   |   if oldFactor > newFactor ∧ oldFactor mod newFactor == 0
   |   |   then
20 |   |   |   multiple = true;
21 |   |   end
22 |   end
23 |   if ¬multiple then
24 |   |   foundFactors.append(newFactor);
25 |   |   N = N/newFactor;
26 |   end
27 end
```

```
T: 1/10 - 0.78029
T: 2/10 - 0.943989
T: 3/10 - 1.00278
T: 4/10 - 1.2225
T: 5/10 - 1.14112
T: 6/10 - 0.915327
T: 7/10 - 1.153
T: 8/10 - 0.738243
T: 9/10 - 0.747342
T: 10/10 - 0.769042
T done
```

Figure 8: Output from running *std::cerr* found in C *timetest* example

C *timetest* example

Example:
Executing:

```
./timetest.out T 0 1000 5 10 > zerotoonethousand.dat
```

results in an output to *std::cerr* as seen in figure 8 where the initial T signifies that normal trial division was done. The $i/10$ signifies how much of the span from $0 - 1000$ was completed. The decimal number after the - symbol signifies how much time it took to factorize the numbers 5 times each (the 5 originates from the 4:th argument to the executable).

The contents of the poorly named *zerotoonethousand.dat* seen in figure 9 (actual naming conventions used are described in section 4.3 Bash - *result* and *result2*) are used to plot the data. Every line contains two numbers separated by white space. The leftmost number describes which numbers where factored and the rightmost how long it took.

```
100 0.156058
200 0.188798
300 0.200557
400 0.244499
500 0.228225
600 0.183065
700 0.2306
800 0.147649
900 0.149468
1000 0.153808
```

Figure 9: Contents of file *zerotoonethousand.dat* after running the executable *timetest* with arguments found in C *timetest* example