# Impact of Optimization and Parallelism on Factorization Speed of SIQS

**Dominik BREITENBACHER**
**Faculty of Information Technology, BUT**
**Bozetechova 1/2, 612 66 Brno, Czech Republic**

**Ivan HOMOLIAK**
**Faculty of Information Technology, BUT**
**Bozetechova 1/2, 612 66 Brno, Czech Republic**

**Jiri JAROS**
**Faculty of Information Technology, BUT**
**Bozetechova 1/2, 612 66 Brno, Czech Republic**

**and**

**Petr HANACEK**
**Faculty of Information Technology, BUT**
**Bozetechova 1/2, 612 66 Brno, Czech Republic**

## ABSTRACT

This paper examines optimization possibilities of Self-Initialization Quadratic Sieve (SIQS), which is enhanced version of Quadratic Sieve factorization method. SIQS is considered the second fastest factorization method at all and the fastest one for numbers shorter than 100 decimal digits, respectively. Although, SIQS is the fastest method up to 100 decimal digits, it cannot be effectively utilized to work in polynomial time. Therefore, it is desirable to look for options how to speed up the method as much as possible. Two feasible ways of achieving it are code optimization and parallelism. Both of them are utilized in this paper. The goal of this paper is to show how it is possible to take advantage of parallelism in SIQS as well as reach a large speed-up thanks to detailed source code analysis with optimization. Our implementation process consists of two phases. In the first phase, the complete serial algorithm is implemented in the simplest way which does not consider any requirements for execution speed. The solution from the first phase serves as the reference implementation for further experiments. An improvement of factorization speed is performed in the second phase of the SIQS implementation, where we use the method of iterative modifications in order to examine contribution of each proposed step. The final optimized version of the SIQS implementation has achieved over 200x speed-up.

**Keywords**: Factorization, SIQS, Parallelism, OpenMP, Profiling, RSA cryptanalysis.

## 1. INTRODUCTION

A factorization is a process which aims at finding the factors of a given composed number in reversible fashion. The factorization is NP-hard computational problem which means that it cannot be efficiently resolved in polynomial time. One of the example areas, where factorization is being utilized, is the RSA cipher cryptanalysis [16]. The RSA presumes that for sufficiently long keys (2048 bits and longer), the attacker is unable to compute this computational problem and decipher an encrypted message.

Many factorization methods have been presented, while SIQS is one of them. It is described for example by Contini in [5]. SIQS is the most optimized version of QS which is the fastest method for factorization of composite numbers up to 100 decimal digits (332 bits) and the second fastest in general [14], [15]. The drawback of SIQS is its difficult comprehensibility. On the other hand, it is much more comprehensible in comparison to General Number Field Sieve [2]. The factorization speed of SIQS depends on many aspects. As we will show later, it is possible to split SIQS into several submodules. Each submodule has its own complexity and issues which have to be considered and resolved separately in order to make implementation efficient. In this paper, we discuss common issues and present our approaches to deal with them. Our approaches are primarily based on a code profiling analysis and memory utilization analysis which are accompanied by a parallelism.

The paper is organized as follows. Section 2 describes SIQS in detail and splits it into logical submodules. Section 3 proposes our approach and describes the methodology and the process of SIQS implementation. The performance issues and optimization process for achieving faster factorization are covered in Section 4. Section 5 discusses SIQS demands on a memory subsystem as well as a way of optimizing them. Section 6 presents the influence of parallelism on the factorization speed. The comparison of our optimized SIQS implementation with another one is covered in Section 7. The conclusion summarizing the achieved results is presented in Section 8.

## 2. THE SIQS FACTORIZATION

The Quadratic Sieve (QS) is one of the most used methods for factorization of large composite numbers. QS is described by Pomerance in [14], [15], and it origins from the Fermat factorization method, which is based on the fact that each odd number can be expressed as a subtraction of two squares:

$$n = u^2 - v^2 = (u - v)(u + v) \quad (1)$$

Fermat factorization method sets $u = \lceil \sqrt{n} \rceil$, and if $u^2 - n \neq v^2$, then a factor is not found, $u$ is incremented by 1 and the process is repeated again. Fermat factorization is very time consuming, because this method is efficient only if the composite number is composed of a product of two close numbers.

Kraitchik proposed improvements of the Fermat factorization method in [15]. He found out that we do not need to look only for numbers where the equation $n = u^2 - v^2$ is true, but it is enough to find number, where $u^2 - v^2$ is a multiple of composite number $n$. It can be also written as:

$$u^2 \equiv v^2 \ (mod\ n) \quad (2)$$

We can realize that using this method can lead to gaining trivial factors as a result (i.e. $n$ and 1) which does not pose any value for factorization. If inequality $u \not\equiv \pm v \ (mod\ n)$ is true, then the result is non-trivial and each factor can be computed as $GCD(u - v, n)$ or $GCD(u + v, n)$, where $GCD$ means greatest common divisor. Kraitchik suggested that it is efficient to find numbers $u_i$ for $i \in \{1, \dots, k\}$, which are subject to:

$$\begin{aligned} u^2 &= u_1^2 \times \dots \times u_k^2 \\ &\equiv (u_1^2 \ mod\ n) \times \dots \times \left(u_k^2 \ mod\ n\right) = v^2 \end{aligned} \quad (3)$$

The fact that we can write every number $m$ as a product of all its prime factors:

$$m = \prod p_i^{e_i}, \quad (4)$$

where $p_i$ are primes and $e_i$ are their exponents, helps us to solve the Eq. (3). Considering the exponents of the prime factors of number $m$, we can make a vector:

$$e(m) = (e_1, e_2, \dots) \quad (5)$$

which is also called a relation. We are looking for a square, however the vector of exponents provides us with higher amount of information than we need. Thus, the vector of exponents is usually reduced to modulo 2. The goal of this method is to find vectors that produce a square. If the sum of two or more vectors results in the null vector, then a square is found. The precise number of vectors needed to find the null vector is described by Brillhart and Morrison in [12]. As we are always limited by available memory, it is appropriate to limit the length of the exponents in a vector.

As the next step, we have to determine how large factor base should be used before performing a factorization by the QS. The factor base is a set of the first $F$ primes. The $F$-th prime of the factor base is denoted as $B$. Every number that has only prime factors smaller or equal to $B$ is called $B$-smooth number. Only $B$-smooth numbers are used to find the null vector. In order to find the null vector, we have to gather at least $F + 1$ vectors. If a linear dependency exists among vectors, then at least one instance of the null vector occurs in a set of vectors. Notice that squares gained by this method may not lead to a non-trivial result, and therefore it is always necessary to check whether $u \not\equiv \pm v \ (mod\ n)$ is true.

The SIQS method is an improvement of QS which uses polynomial for generating numbers:

$$Q_{a,b}(x) = a(ax^2 + 2bx + c), \quad (6)$$

instead of:

$$Q(x) = x^2 - n \quad (7)$$

as QS does. The method of computing the coefficients $a$, $b$ and $c$ is described in [5]. QS uses only one polynomial to find $B$-smooth numbers, however, SIQS can use many polynomials to find $B$-smooth numbers thanks to the coefficients represented by the polynomials. Therefore, the SIQS uses variable $x$ only at a specified interval. The polynomial is changed after depleting the whole interval, and thus searching for $B$-smooth numbers is more efficient.

The SIQS algorithm can be divided into several submodules and every submodule can be implemented in many different ways. As we mentioned earlier, the factorization is NP-hard computational problem, and therefore it is desirable to implement it as efficiently as possible. This paper divides the SIQS algorithm into the following parts:

- **A. SIQS parameters configuration,**
- **B. polynomial generation,**
- **C. sieving** and
- **D. resolution of linear dependency,**

which will be closer described in following rows.

### A. SIQS Parameters Configuration

The SIQS parameters configuration part has significant impact on the factorization speed because this part affects the whole process of the factorization. The major parameters that we configure in this phase are the size of the factor base and the size of the sieving interval. Each parameter has its own influence on the performance of SIQS. If the parameters are chosen improperly, then the factorization fails or is not efficient. It is necessary to make dedicated configuration of parameters for every input composed number in order to make the method more efficient.

### B. Polynomial Generation

The implementation of polynomial generation influences the speed and a quality of generated polynomials. The high quality polynomials are desirable for speeding up the factorization, because they cause higher likelihood of finding the relation and also lower the likelihood of duplicate relations' occurrence. The process of polynomials generation and quality discussion about them can be found in [3]. We experimentally found out, that the logarithm of coefficient $a$ of the given polynomial should not differ by more than 0.01 from its optimal value. According to [5], the optimal value of the coefficient $a$ is computed by solving the equation:

$$a_{optimal} = \frac{\sqrt{2n}}{M} \quad (8)$$

It should be noted that coefficient $a$ is created as a product of $s$ primes from factor base, and thus causes $a$ being very close to its optimal value (not equal to it).

### C. Sieving

The sieving phase is the most time consuming phase of SIQS and its objective is to gather the necessary amount of unique relations. Sieving can be divided into three parts:

A. *computation of polynomial roots,*
B. *candidate selection* and
*C. candidate verification*

Computation of polynomial roots has to be done for every prime that we have in our factor base. If the sum of primes' logarithms for a given $x$ of the current polynomial exceeds the threshold, then $x$ is marked as a candidate. The threshold is computed by solving the equation [5]:

$$log(2x\sqrt{N}) \quad (9)$$

The candidates are verified after the candidate selection. If a candidate is successfully verified, then a relation is created. The candidate verification is performed by division using all primes in the factor base. If the result of a division is equal to 1, then the candidate is successful. The candidate verification part does not take a lot of time, as there are only few candidates necessary to be verified.

### D. Resolution of Linear Dependency

The last phase of the SIQS represents resolution of linear dependency. This phase performs transformation of relations to a matrix which is exploited for finding of linear dependent rows. Usually, several linear dependencies are found. Each linear dependency is then checked whether it leads to a factor of the given composed number.

## 3. IMPLEMENTATION DETAILS

The SIQS method has been implemented in the C++ language on *x86-64* architecture. C++ has been chosen because its standard libraries provide many data types that are often used in the implementation as well as there are many C++ based profiling tools available. The next reason, why we have chosen C++, is OpenMP [4] support which is employed for our parallelism. Also, MPIR library (version 2.6.0) has been employed because it is capable of holding very large numbers and provides various operations with them. The Single Large Prime Variation (SLPV) has been implemented in order to speed up the SIQS algorithm [7], [8].

Our implementation consists of two phases. In the first phase, the complete and functional algorithm is implemented in the simplest way which does not consider the requirements for execution speed. Then, the soundness of the implementation is verified on the smaller numbers that have up to 40 decimal digits. This version serves as the reference SIQS implementation for further experiments.

The speed optimization is the second phase of our SIQS implementation, where the method of iterative optimization has been utilized. The SIQS algorithm is composed of several steps which are logically connected. Each step has its own time complexity ranging from the linear complexity to the cubic one. With increasing length of the composite numbers, each iteration of optimization reveals specific critical parts of the algorithm. Every optimization phase is examined and the influence on the execution time as well as the memory consumption is measured. Also, the influence of the performance on the initial settings is evaluated. The latest version of our implementation has been compared to the MSieve[1] which is an open source implementation of SIQS.

### A. Multiple Interpretations of SIQS Algorithm

Our implementation of SIQS follows theoretical and mathematical principles described in [5]. There are many ways of implementing the SIQS method which is the reason why different SIQS implementations exist. Therefore, each implementation of the SIQS algorithm may differ in its factorization speed compared to the other ones. In the following sections we closely describe the main principles which make our implementation unique.

### B. NEXKSB and Binary Search Tree

To ensure that our polynomial generation submodule is able to generate polynomials of high quality, the part for generation of coefficient $a$ has been implemented according to [3]. The NEXKSB algorithm has been implemented for lexicographical prime selection [13] which enables us to ensure that the coefficient $a$ always differs at least in one prime. Using the NEXKSB algorithm, we select the first $s-2$ primes for generating of the coefficient $a$. Our reference implementation utilizes NEXKSB with selection of $s-3$ primes.

Binary Search Tree (BST) is implemented in order to provide us the remaining primes [6]. BST aims at ensuring that the logarithm of the coefficient $a$ is as close as possible to its optimal value. Each node of BST contains a pair of primes from the selected subset of the factor base together with a logarithm of their product. The floating point data type is used as a key in the BST implementation. First, the optimal value of the coefficient $a$ is computed. Then, $s-2$ primes are selected by the NEXKSB algorithm followed by the logarithm computation of the product of these primes. The key for searching in BST is the difference between the optimal value of the coefficient $a$ and the previous logarithm value. The algorithm enables us to traverse in BST until the closest key is found. The difference between the input key and the closest one represents the difference between the generated coefficient $a$ and its optimal value. As we mentioned in Section 2.B, the difference should not be higher than 0.01 which is achieved by appropriate selection of primes in a subset of the factor base.

### C. Gaussian Elimination Method

When the required amount of the relations is gathered, we need to find the relations which produce the null vector. Thus, we create the matrix of relations. Then, the Gaussian Elimination (GE) method is utilized in order to find the null vector among rows in the matrix of relations [10]. Sieving process usually gathers more than enough amount of relations. Thanks to this, more than one null vector may exist among the gathered relations. The GE method finds all null vectors that exist in the matrix. Referring to Section 2, it may happen that an instance of the null vector will result into trivial factor – which is not desired. As we have more than one null vector available, it is very unlikely that all of them will lead to trivial factors[2].

### D. The Utilization of Parallelism

This section describes how the parallelism is implemented in our SIQS algorithm. As was previously mentioned, the SIQS algorithm can be divided into logically separated parts. The first

[1] https://sourceforge.net/projects/msieve/

[2] If it happens, then we need to repeat the whole process of factorization.

of them is parameters configuration of the SIQS which performs adjustments of many parameters before the sieving process can start. There is no possibility of any efficient parallelism, as the parameters are being adjusted as a complex unit.

When the parameters are adjusted, we can generate the polynomials and start sieving. We are able to generate many polynomials which are independent to each other, and thus they can also independently contribute to the factorization itself. This fact enables us to exploit many program threads, whereas each of them generates one polynomial. Each program thread can perform the sieving process independently to the other threads, as it has its own polynomial generated. When polynomials are depleted, gathered relations are stored in the shared array of the relations. If there are not enough relations gathered yet, the thread can continue in generating new polynomial and repeats the process of sieving with new polynomial.

Notice, that in practice, it is not possible to sieve fully independently because of two reasons. The first reason is that a situation, where two threads generate the same polynomial leading to the acquirement of the same relations, may occur. To avoid this situation, we have to control the access to the NEXKSB by using a critical section provided by OpenMP directive `#pragma omp critical`. The second reason is the fact that two threads may find a relation at the same time. Therefore, the storage of gathered relations has to be controlled by other critical section. Except of these two situations, each thread runs independently to the others.

With growing length of a factorized number, these situations occur more rarely which also contribute to parallelism efficiency. Only the sieving process and polynomial generation are parallelized in the first phase of our implementation. The pseudo-code of the parallel algorithm is shown in Algorithm 1.

**Algorithm 1**: Proposed parallelism of the SIQS

**Input**: Composed number
**Output**: Factor

```
1:  ConfigureSIQS();
2:  begin #pragma omp parallel
3:    while (num_of_relations < desired_num) do
4:      GeneratePolynom();
5:      Sieve();
6:    end while
7:  end
8:  SolveMatrix();
9:  ComputeResult();
```

## 4. EXPERIMENTS AND RESULTS

The speed measurements were performed and examined after the implementation and validation of our reference version. The first measurement was executed on 30 numbers with 40 decimal digits and 30 numbers with 50 decimal digits. Later, we chose one number for each length (60, 70, 80 and 90 decimal digits) as the representative demonstrating the behavior of our implementation. Each number in our testing dataset was a semi-prime, which means that it was a product of two prime numbers. The reference version of our implementation was executed both in serial and parallel mode. Development and measurements were performed on a machine equipped with Intel i7 4700MQ having 4 physical cores. The Hyper-Threading and the TurboBoost technologies were enabled during our experiments. The goal of the performed measurements was to evaluate the speed of the SIQS algorithm and its behavior depending on increasing size of a factorized number. The results are depicted in Table 1. The column *Task* represents the size of a factorized number expressed in decimal digits.

TABLE 1
Performance of reference version

| Task | Serial Mode | Parallel Mode | Speed-up |
|---|---|---|---|
| 40 dec | 72.68s | 22.71s | 3.20 |
| 50 dec | 984.07s | 307.96s | 3.20 |
| 60 dec | 9144.23s | 3217.55s | 2.84 |

As the performance of the reference version was not sufficient enough, the code profiling was executed in order to find and examine the most time consuming parts of the code. The Intel VTune Amplifier XE 2013[3] was employed for profiling purposes.

### A. Profiling and Optimization

We identified the critical parts of the code by using the profiling tool. The profiling was performed in iterations, where each iteration revealed the most critical part of the algorithm. After each iteration, the solution for the actual issue was proposed and implemented. Thanks to this approach, we achieved a significant speed-up. The overall influence of the proposed modifications is shown in Table 2.

TABLE 2
Impact of optimization – serial mode

| Task | Reference Version | Optimized Version | Speed-up |
|---|---|---|---|
| 40 dec | 72.68s | 2.12s | 34.28 |
| 50 dec | 984.07s | 12.49s | 78.79 |
| 60 dec | 9144.23s | 102.19s | 89.48 |

The essential modifications were applied to the sieving process and the resolution of linear dependency part. Also, optimized memory management brought significant contribution to the speed-up of the algorithm. All data objects were allocated at the time they were required, and de-allocated when they had no longer been needed.

The way of storing the relations was changed as well. Before the modifications, the relations were stored as an array of Boolean data type, where each item held information about one prime. This approach allowed a programmer to easily work with the relations and perform required operations with the relations, however operations with Boolean data type were very time consuming. Therefore, we proposed the modification which substituted the Boolean data type to the integer one. The relations were then stored as an array of integers, where each of them held information about multiple primes. Thus, the operations performed on the relations were applied on multiple primes at the same time.

At the beginning of the sieving process, the roots of the polynomial are being computed for each prime in the factor base. The roots determine values of variable $x$ in which the polynomial is divisible by the given prime. It means that we are

[3] https://software.intel.com/en-us/intel-vtune-amplifier-xe

able to identify all the factors of the given value considering the polynomial with specified $x$. Factors of each $x$ value are stored during the sieving process considering specified polynomial. This immediately allowed us to check whether the candidate is a $B$-smooth number by dividing it by the factors of the candidate. Moreover, the profiling revealed that the storing of factors was inefficient, and therefore it was better to ignore previous proposal and just use the Trial Division method [7] for validation of the candidate.

### B. Expanding the Parallelism

The next goal of our optimization phase was to parallelize the sieving process and resolution of linear dependency in order to achieve the maximum utilization of the CPU. We analyzed the possibilities and performed appropriate modifications. We proposed removal of unacceptable relations after sieving as one of the possibilities for further optimization. As unacceptable relations are threat relations which are singletons, duplicates and null vectors. Singleton is a relation that contains a prime which is not present in any other relation. Therefore, the singleton will never be part of a set of relations forming the null vector. Duplicate relations have more than one occurrence in the array of gathered relations, and thus naturally form null vectors leading to the trivial factor. Also, the situation leading to trivial factor is similar in the null vector but in this case it is necessary to check whether one of them does not lead to a non-trivial result. If so, the resolution of linear dependency can be skipped and the result can be immediately displayed. The removal of unacceptable relations always requires to go through the whole array of gathered relations in order to check whether any unacceptable relation is present.

During this process, the iterations are independent of each other which means this part can also be parallelized. We utilized the OpenMP directive `#pragma omp parallel for` in this case.

Regarding the resolution of linear dependency accomplished by GE method, we realized that its iterations are independent of each other too, therefore this part can be also parallelized using the same directive as in previous case.

The results achieved by the mentioned optimization and the code parallelism are depicted in Table 3 and Table 4. It can be seen that the modifications caused large speed-up, e.g. number with 60 decimal digits (198 bits) was factorized 100 times faster compared to the reference version. This also shows that although SIQS is considered as the second fastest factorization method in general, the speed of two individual implementations may differ in significant scale.

TABLE 3
Performance after optimization – parallel mode

| Task | Reference Version | Optimized Version | Speed-up |
|---|---|---|---|
| 40 dec | 22.71s | 1.32s | 17.20 |
| 50 dec | 307.96s | 3.87s | 79.58 |
| 60 dec | 3217.55s | 32.23s | 99.83 |

TABLE 4
Performance of optimized version

| Task | Serial Mode | Parallel Mode | Speed-up |
|---|---|---|---|
| 40 dec | 2.12s | 1.32s | 1.61 |
| 50 dec | 12.49s | 3.87s | 3.22 |
| 60 dec | 102.19s | 32.23s | 3.17 |

## 5. OPTIMIZATION OF MEMORY ACCESS

During the profiling process, we reached a state where further modifications led to factorization speed-up of numbers up to 60 decimal digits, but on the other hand, factorization of numbers with more than 70 decimal digits (235 bits) became slower than before. Therefore, we performed code profiling of factorization of numbers with 60 and 70 decimal digits. The results of the profiling are depicted in Figure 1 and Figure 2. It can be seen that distribution of time consumption significantly differs in both cases.

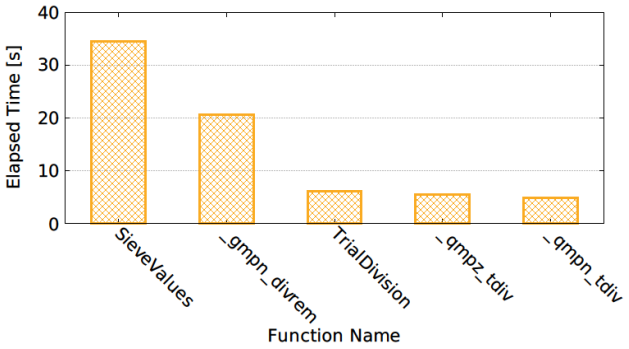


*Fig. 1. Code profiling of the factorization of number with 60 decimal digits*

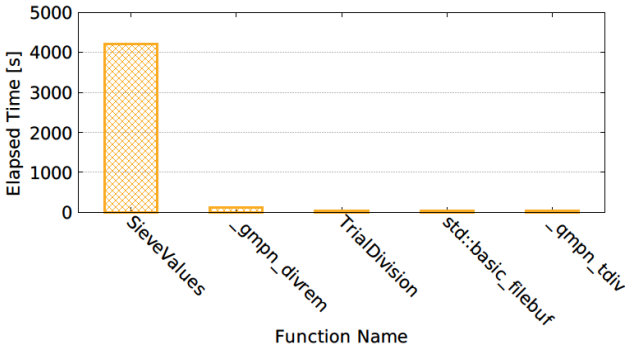


*Fig. 2. Code profiling of the factorization of number with 70 decimal digits*

Profiling of memory demands of factorization was performed using number with 70 decimal digits. The requested memory bandwidth is depicted in Figure 3 by orange (light) color. It shows that memory requirements were such a high that the memory subsystem was not capable of transferring the requested amount of data which further led to CPU stall. Average memory latency equaled to 76 cycles and Cycles Per Instruction (CPI) equaled to 4.707 which is far away from the optimal state. It was even faster to perform some computations again instead of storing them in memory and fetching them later. The section of the code which had the highest requirements on memory bandwidth was a part of the sieving process – see Algorithm 2. For each prime, we update the array of the roots by adding the logarithm of the current prime in the index where the prime is a root of the polynomial.

**Algorithm 2**: The part of code with the highest requirements on memory bandwidth

```
1:  for (i = 0; i < number of primes; i++) do
2:    for (root = roots[i].root1;
3:         root < positive endpoint of interval;
4:         root += roots[i].prime) do
5:      xValues[root] += logPrime[i];
6:    end for
7:    for (root = roots[i].root2;
8:         root < positive endpoint of interval;
9:         root += roots[i].prime) do
10:     xValues[root] += logPrime[i];
11:   end for
12:   for (root = roots[i].root1_neg;
13:        root < negative endpoint of interval;
14:        root += roots[i].prime) do
15:     xNegValues[root] += logPrime[i];
16:   end for
17:   for (root = roots[i].root2_neg;
18:        root < negative endpoint of interval;
19:        root += roots[i].prime) do
20:     xNegValues[root] += logPrime[i];
21:   end for
22: end for
```

Then, we need to check the array whether the threshold of overall sum of logarithms is not exceeded in the examined index. If the threshold is exceeded, then the current index is marked as the candidate. Every prime has two roots[4] on the positive side of the given interval and two roots on the negative one. Therefore, we have to store the information about these cases in two arrays. For example, we store circa 5000 primes for factorization of number with 70 decimal digits. Also, we have to store the information about the sums of logarithms for the whole interval. When number with 70 decimal digits is being factorized, the interval is set to $[-196608; 196608]$ in that case. Furthermore, as the logarithms of primes are frequently used during the factorization, their values are computed in the phase of the SIQS configuration and stored in the arrays for further usage. These arrays have to be available for read and write operations during the sieving. The size of the arrays increases with every bigger number we try to factorize, and thus it puts higher and higher demands on memory subsystem.

The critical part of the code was modified in the following way. We divided the interval into blocks and the sums of logarithms were updated for each prime within the current block. When the block was updated, the algorithm proceeded to the next block. This approach significantly reduced demands on the memory subsystem because only a part of the interval was processed in a specific time. This increased the likelihood of storing required data in the cache memory of the CPU which provided data faster. The new requirements on the memory bandwidth of the factorization of number with 70 decimal digits are depicted in Figure 3 by green (dark) color. We can see that our memory access modification decreased requirements on the memory bandwidth by approximately 50 percent.

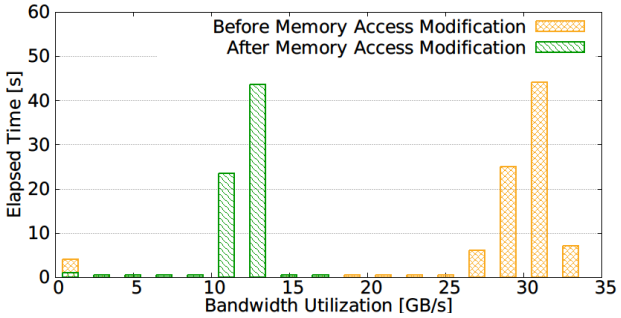


*Fig. 3. Memory demands when factoring number with 70 decimal digits*

Also, we measured the influence of our memory access modification on the factorization speed. The results are shown in Table 5. It should be noted that beside the mentioned modification we also slightly changed the approach to polynomial generation. In the actual version of our implementation, the coefficient $a$ was created by using NEXKSB algorithm for selecting the first $s - 2$ primes. The last two primes were chosen using BST. This modification significantly reduced factorization time for smaller numbers than 50 decimal digits but has negligible impact on speed-up of large numbers.

TABLE 5
Influence of memory modification – parallel mode

| Task | Before | After | Speed-up |
|---|---|---|---|
| 40 dec | 1.32s | 0.25s | 5.28 |
| 50 dec | 3.87s | 2.95s | 1.31 |
| 60 dec | 32.23s | 14.77s | 2.18 |
| 70 dec | 583.67s | 143.67s | 4.06 |

Thanks to our memory modification, the factorization time of numbers with 70 decimal digits was $4x$ shorter than in the case where modification was not utilized. The average memory latency had been reduced from 76 to 14 cycles and the CPI had been reduced from 4.707 to 1.023. Table 6 depicts achieved speed-up of the current SIQS version in comparison with the reference one. We achieved more than 200 times speed-up in the case of numbers with 60 decimal digits. We also performed factorization of larger numbers than 70 decimal digits. Number with 80 decimal digits (266 bits) was successfully factorized in 50 minutes and 30 seconds as well as number with 90 decimal digits (299 bits) which was successfully factorized in 6 hours and 32 minutes.

TABLE 6
Performance after memory optimization – parallel mode

| Task | Reference Version | Optimized Version | Speed-up |
|---|---|---|---|
| 40 dec | 22.71s | 0.25s | 90.84 |
| 50 dec | 307.96s | 2.95s | 104.39 |
| 60 dec | 3217.55s | 14.77s | 217.84 |

[4] Exception is the prime 2 which has only one root.

## 6. PARALLELISM EFFICIENCY

The current section performs measurements of parallelism efficiency which evaluates the speed-up of parallelism considering the final optimized version of the SIQS implementation. The results obtained by experiments on Intel i7 are shown in Table 7. It can be seen that the performance had been improved from this point of view as well. The factorization is at least $3x$ faster for each size of measured number which is the outcome of the parallelism only.

TABLE 7
Parallelism efficiency – Intel i7 4700MQ

| Task | Serial Mode | Parallel Mode | Speed-up |
|---|---|---|---|
| 40 dec | 0.82s | 0.25s | 3.28 |
| 50 dec | 9.76s | 2.95s | 3.31 |
| 60 dec | 51.83s | 14.77s | 3.51 |
| 70 dec | 464.98s | 143.67s | 3.24 |

In order to show how CPU can affect the factorization speed, our implementation was later tested on more powerful CPU, Intel Xeon E5 1650v2 having 6 physical cores. Hyper-Threading and TurboBoost were enabled on the Intel Xeon during the tests as well as it was enabled in the case of Intel i7. The factorization speed of our implementation executed on the Intel Xeon is shown in Table 8.

TABLE 8
Parallelism efficiency – Intel Xeon E5 1650 v2

| Task | Serial Mode | Parallel Mode | Speed-up |
|---|---|---|---|
| 40 dec | 0.83s | 0.20s | 4.15 |
| 50 dec | 9.29s | 1.58s | 5.88 |
| 60 dec | 52.07s | 7.83s | 6.65 |
| 70 dec | 459.40s | 71.16s | 6.46 |

It can be seen that the performance of our implementation in parallel mode was significantly faster on Intel Xeon than on Intel i7, which was primarily caused by the fact that Intel Xeon provided more logical threads than Intel i7 (12 vs 8). Furthermore, the Intel Xeon had higher ratio of available cache memory per logical thread. Therefore, it was more likely that required data were stored in the cache, and thus available much faster than in the case of Intel i7 having smaller cache per logical thread. On the other hand, the factorization speed of Intel Xeon was very similar to the Intel i7 in serial mode. We can see that the high frequency base of Intel Xeon did not provide significant difference of speed in serial mode comparing to Intel i7. The previous results showed that chosen CPU can have significant influence on the factorization speed. Also, the high base frequency does not guarantee that the factorization will be fast because its speed depends on many other factors.

### A. TurboBoost & Factorization Speed

We measured the factorization speed of our implementation in serial mode with the TurboBoost technology enabled on Intel i7. Thus, the factorization in serial mode run on higher frequency compared to the parallel mode. Precisely, the factorization ran on 3.4GHz in serial mode and 2.4GHz in parallel mode, respectively. As the next step, we disabled TurboBoost and watched raw contribution of the parallelism employed in the factorization implementation. The results are depicted in Table 9. Comparing the current results with those from Table 7, we can conclude that there is only little difference in parallel mode which does not harvest from TurboBoost, however, the impact is much more significant in the case of serial mode. Therefore, disabling TurboBoost caused parallelism being more efficient and achieved factorization speed-up of more than $4x$.

TABLE 9
Parallelism efficiency – Intel i7 4700MQ
(disabled TurboBoost)

| Task | Serial Mode | Parallel Mode | Speed-up |
|---|---|---|---|
| 40 dec | 1.11s | 0.32s | 3.47 |
| 50 dec | 13.51s | 3.00s | 4.50 |
| 60 dec | 72.01s | 16.05s | 4.49 |
| 70 dec | 641.98s | 143.67s | 4.47 |

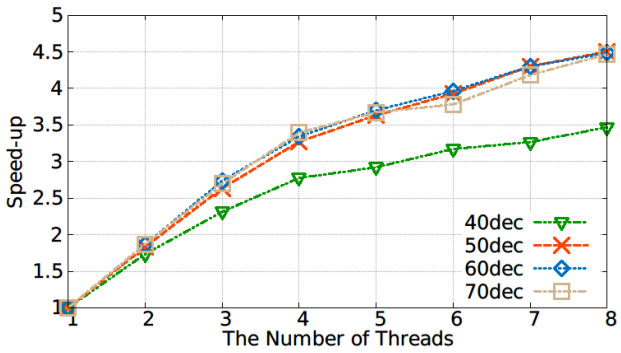


*Fig. 4. Profiling – Scalability*

It should be noted that TurboBoost was controlled by the actual power consumption, therefore TurboBoost was activated only if the power consumption was low during the start of the factorization in parallel mode. However, TurboBoost was deactivated after a short time due to high utilization of all available logical threads causing high power consumption. Nevertheless, it causes only a little difference in the factorization time of parallel mode.

We also evaluated the scalability which is depicted in Figure 4. We can see that the speed-up significantly increased with each added logical thread up to four which correspond to the physical threads of the CPU in this case. When more than four logical threads were utilized, Hyper-Threading was enabled, and therefore further speed-up was not increased in a such significant scale as in the case of utilizing only physical threads.

## 7. COMPARISON WITH MSIEVE

We compared our latest version of SISQ with public open source implementation of SIQS called MSieve. Both implementations were compared in serial mode. The reason of such comparison is to show differences in the raw performance and to omit diversity between OpenMP which is utilized in our work and MPI which is utilized by MSieve. The results of the comparison are shown in Table 10. Notice that our implementation is referred as SIQS.

TABLE 10
Comparison of our implementation with MSieve – Serial mode

| Task | SIQS | MSieve |
|---|---|---|
| 40 dec | 0.82s | 0.15s |
| 50 dec | 9.76s | 0.56s |
| 60 dec | 51.83s | 3.55s |
| 70 dec | 459.40s | 39.03s |

There can be seen that our implementation was not as fast as MSieve but our goal was not to make the fastest implementation of SIQS ever but rather to analyze implementation and performance issues as well as explain the ways of resolving them, and thus improve efficiency of the raw SIQS algorithm.

## 8. CONCLUSION

The paper presents the process of iterative performance optimization of the SIQS factorization method. In the first phase, the complete and functional plain SIQS algorithm is implemented which serves as the reference implementation for our further experiments. In the second phase, a speed optimization of the reference implementation is being performed by iterative application of various optimization techniques, while new critical parts of the code are being identified and optimized in each iteration. $200x$ speed-up has been achieved in comparison to the reference version. We also performed efficiency evaluation of a parallelism on the latest optimized version of SIQS running on the two CPUs with enabled TurboBoost. $3x$ speed-up has been achieved on the CPU with 4 physical cores while the speed-up of more than $6x$ has been achieved on the CPU with 6 physical cores. The speed-up has been even better when TurboBoost has been disabled.

In the future work we plan to implement modification based on [1] which may reduce the sieving time. We also plan to replace the SLPV by Double Large Prime Variation method which makes the sieving process more efficient [11]. Next, we plan to employ OpenMPI library which allows the utilization of cluster of computers [9].

## 9. ACKNOWLEDGEMENT

This article was created within the project Reliability and Security in IT (FIT-S-14-2486) and supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science – LQ1602.

## 10. REFERENCES

[1] AOKI, Kazumaro; UEDA, Hiroki. Sieving using bucket sort. In: **Advances in Cryptology-ASIACRYPT 2004**. Springer Berlin Heidelberg, 2004. p. 92-102.

[2] BERNSTEIN, Daniel J.; LENSTRA, Arjen K. A general number field sieve implementation. In: **The development of number field sieve**. Springer Berlin Heidelberg, 1993. p. 103-126.

[3] CARRIER, Brian; WAGSTAFF JR, Samuel S. Implementing the hypercube quadratic sieve with two large primes. In: **International Conference on Number Theory for Secure Communications**. 2003. p. 51-64.

[4] CHAPMAN, Barbara; JOST, Gabriele; VAN DER PAS, Ruud. **Using OpenMP: portable shared memory parallel programming**. MIT press, 2008.

[5] CONTINI, Scott Patrick. Factoring integers with the self-initializing quadratic sieve. 1997.

[6] CORMEN, Thomas H. **Introduction to algorithms**. 3rd ed. Cambridge, Mass.: MIT Press, 2009. ISBN 0262533057.

[7] CRANDALL, Richard; POMERANCE, Carl B. **Prime numbers: a computational perspective**. Springer Science & Business Media, 2006.

[8] GERVER, Joseph L. Factoring large numbers with a quadratic sieve. **Mathematics of Computation**, 1983, 41.163: 287-294.

[9] GROPP, William; LUSK, Ewing; SKJELLUM, Anthony. **Using MPI: portable parallel programming with the message-passing interface**. MIT press, 1999.

[10] KOÇ, Çetin K.; ARACHCHIGE, Sarath N. A Fast Algorithm for Gaussian Elimination over GF (2) and its Implementation on the GAPP. **Journal of Parallel and Distributed Computing**, 1991, 13.1: 118-122.

[11] LENSTRA, Arjen K.; MANASSE, Mark S. Factoring with two large primes. **Mathematics of Computation**, 1994, 63.208: 785-798.

[12] MORRISON, Michael A.; BRILLHART, John. A method of factoring and the factorization of $F_7$. **Mathematics of computation**, 1975, 29.129: 183-205.

[13] NIJENHUIS, Albert; WILF, Herbert S. **Combinatorial algorithms: for computers and calculators**. Elsevier, 2014.

[14] POMERANCE, Carl. The quadratic sieve factoring algorithm. In: **Advances in cryptology**. Springer Berlin Heidelberg, 1985. p. 169-182.

[15] POMERANCE, Carl. A tale of two sieves. **Biscuits of Number Theory**, 2008, 85.

[16] RIVEST, Ronald L.; SHAMIR, Adi; ADLEMAN, Len. A method for obtaining digital signatures and public-key cryptosystems. **Communications of the ACM**, 1978, 21.2: 120-126.