A comprehensive empirical comparison of parallel ListSieve and GaussSieve

Artur Mariano[1], Özgür Dagdelen[2], and Christian Bischof[1]

[1] Institute for Scientific Computing, Technische Universität Darmstadt
[2] Cryptography and Computer Algebra, Technische Universität Darmstadt
`artur.mariano@sc.tu-darmstadt.de`,`oezguer.dagdelen@cased.de`
`christian.bischof@sc.tu-darmstadt.de`

Abstract The security of lattice-based cryptosystems is determined by the performance of practical implementations of, among others, algorithms for the Shortest Vector Problem (SVP).
In this paper, we conduct a comprehensive, empirical comparison of two SVP-solvers: ListSieve and GaussSieve. We also propose a practical parallel implementation of ListSieve, which achieves super-linear speedups on multi-core CPUs, with efficiency levels as high as 183%. By comparing our implementation with a parallel implementation of GaussSieve, we show that ListSieve can, in fact, outperform GaussSieve for a large number of threads, thus answering a question that was still open to this day.

Keywords: sieving, superlinear speedup, shortest vector, parallel

1 Introduction

Cryptography aims mostly at protecting information sent over an insecure channel. The implementation of cryptosystems is usually a three-phase engineering process. First, a cryptosystem with a certain underlying mathematical problem is specified. Second, an implementation of an algorithm that solves the underlying mathematical problem is used to identify hard instances of the problem. Third, these instances are used to define parameters for the implementation of the cryptosystem. Therefore, practical implementations of these algorithms are required for the implementation of secure, real-world cryptosystems.

In 1996, Ajtai found out that the mathematical properties of some lattice problems have interesting properties for cryptography, such as average-case to worst-case hardness [1]. Since Ajtai's discoveries, a significant amount of work has been done in this field, commonly referred to as *lattice-based cryptography.*

A lattice Λ is a discrete additive subgroup of $\mathbb{R}^m$. The *dimension* $n \leq m$ of a lattice Λ is the maximum number of mutually linearly independent vectors in Λ. Any such n linearly independent vectors form a *basis*, which represents the lattice. We denote a basis in a column matrix $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$. The lattice $\Lambda(\mathbf{B})$ is defined by the linear integer span of the basis vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n$, namely

$$\Lambda(\mathbf{B}) = \left\{ \sum_{i=1}^{n} a_i \mathbf{b}_i \ : \ a_i \in \mathbb{Z} \right\} .$$

For $n > 2$ there are infinitely many possible bases of a lattice.

Lattice-based cryptography is particularly attractive since it is believed to be resistant against attacks operated with quantum computers, in contrast to problems from number theory, such as factorization of large composite numbers or the computation of discrete logarithms [15]. The security of lattice-based cryptosystems is based on the hardness of specific lattice problems. One of these problems is the Shortest Vector Problem (SVP). The SVP can be formally defined as the computation of a vector $\mathbf{v} \in \Lambda \setminus \{0\}$ where $\|\mathbf{v}\| = \min_{\mathbf{x} \in \Lambda \setminus \{0\}} \|\mathbf{x}\|$. The norm of the shortest vector is denoted by λ_1. This problem can be stated for every norm; in this work, we address the Euclidean norm, the most common in this context. Algorithms that solve this problem are called *SVP-solvers*.

The SVP is known to be NP-hard under randomized reductions [2], and therefore, no polynomial-time algorithms for this problem are expected to be found. In fact, only algorithms that find an approximation to the solution of the SVP, such as the LLL [9] and the Block Korkine Zolotarev (BKZ) [14], are known to run in feasible time for high lattice dimensions. However, the returned vector, while somewhat short, might not be short enough to break a cryptosystem. In fact, LLL and BKZ are lattice basis reduction algorithms, i.e., given a lattice basis, they find another basis with short, nearly orthogonal vectors. BKZ uses an SVP-solver as a sub-routine, which finds the shortest vector of small lattice dimensions, called blocksize [14].

There are two main families of SVP-solvers: enumeration and sieving algorithms. Currently, the fastest SVP-solver is enumeration with extreme pruning [6], which consists in a depth-first traversal of a pruned tree. While enumeration algorithms were extensively studied and implemented in several computer architectures [5,4,8], sieving algorithms attracted lesser attention in this regard.

Published in 2010 [11], ListSieve and GaussSieve are currently the most relevant sieving algorithms. While ListSieve was considered impractical and important mainly for theoretical purposes, GaussSieve was presented as a practical, efficient heuristic of ListSieve. Some work has been done on sieving algorithms since [10,3], but there are still some open questions. For instance, although ListSieve is considered impractical, there are neither assessments of ListSieve in practice nor empirical comparisons of both algorithms. Moreover, only one study [13] (cf. Section 2) focused on the practical behaviour of the original GaussSieve algorithm.

The parallelization of GaussSieve was investigated on multi-core CPUs. Very recently, it was shown that GaussSieve can scale linearly, using scalable lock-free lists [10]. However, the first steps in the parallelization of GaussSieve date back to 2010, when Milde et al. implemented GaussSieve in parallel, with a ring structure of several instances of GaussSieve [12]. As the scalability of the implementation was limited, Milde et al. suggested that ListSieve could possibly outperform GaussSieve for a large number of threads, a question that remains open to this day and we answer in this paper.

The contribution of this paper is twofold. First, we present the first empirical analysis of the workflow of ListSieve and of how it compares to GaussSieve. Second, we propose the first parallel implementation of ListSieve, which relaxes its properties, thus lowering the workload in comparison to the original algorithm. As a direct result, it achieves super-linear speedups on multi-core CPUs, up to 32 threads, and it outperforms the parallel GaussSieve implementation presented in [12] for a big number of threads, while returning the same vector.

Notation. Vectors and matrices are written in bold face, vectors are written in lower-case, and matrices in upper-case, as in vector $\mathbf{v}$ and matrix $\mathbf{M}$. The i^{th} coordinate of a vector $\mathbf{v}$ is denoted by $\mathbf{v}_i$. $\langle \mathbf{v}, \mathbf{p} \rangle$ denotes the inner product of two vectors $\mathbf{v}$ and $\mathbf{p}$. The Euclidean norm of $\mathbf{v}$ is given by $||\mathbf{v}||$. $\mathbf{v}$ is called a *zero vector* if $||\mathbf{v}|| = 0$.

Roadmap. Section 2 provides some background of sieving algorithms and discusses a previous study of their properties. Section 3 presents the results of our experiments with ListSieve and GaussSieve and Section 4 presents the first parallel implementation of ListSieve. Section 5 concludes the paper.

2 The ListSieve and GaussSieve algorithms

All sieving algorithms follow an identical structure. They build a list L of somewhat random vectors, referred to as *samples*, typically generated with Klein's algorithm [7], remove the zero vectors from L and apply a sieving technique on it. The sieving process is iteratively executed until a certain stopping criterion, $K \geq c$, where K is the number of collisions, is met. c is usually set in the form $c = \alpha \times mls + \beta$, where mls is the maximum size of L up to that point. When the sieving process finishes, the shortest vector of the lattice is expected to be in the list L, with a certain, yet high probability. The generation of vectors and the sieving process go hand in hand in practical implementations, since the number of samples that are necessary for the algorithm to converge is not known upfront.

In ListSieve's original form (see [11]), samples are generated with perturbations, a technique useful to infer the asymptotic complexity of the algorithms. The pseudo-codes presented in Algorithm 1 and Algorithm 2, on the other hand, are practical implementations of ListSieve and GaussSieve, wherein randomly generated vectors are not perturbed.

ListSieve samples vectors with Klein's algorithm and reduces them as much as possible against the vectors stored in L, where the freshly reduced vector is inserted in L once the reduction process is finished. When the sample is reduced to the zero vector, a collision takes place and the whole iteration is wasted. Once in L, vectors are never removed or modified. According to the original description of the algorithms, the reduction process can pick the vectors in L in any order. However, it is known that keeping L ordered by increasing norm is more efficient in practice, since the process can be aborted when a vector bigger than the sample is found.

In contrast to ListSieve, GaussSieve also reduces the elements already in L against one another. As a result, the elements in L will be pairwise reduced, which means that the inequality $\min(||\mathbf{p} \pm \mathbf{v}||) \geq \max(||\mathbf{p}||,||\mathbf{v}||)$ holds for all $\mathbf{v}, \mathbf{p} \in L$. This is precisely the property that governs the Gauss/Lagrange basis reduction algorithm for two dimensional lattices, hence the name of the algorithm. We note that the asymptotic time complexity of the algorithm is not known.

Another difference between the algorithms lies in the data structures that they use. GaussSieve uses a list L that, in contrast to ListSieve, can both grow and shrink, and a stack S that temporarily keeps vectors that are removed from L. The use of the stack eases the handling of the vectors that no longer verify the aforementioned inequality. This happens because when a vector $\mathbf{v}$ is generated and reduced against an element in L, there might be elements in L that are no longer pairwise reduced with $\mathbf{v}$. Reverting this is not as simple as reducing such vectors by $\mathbf{v}$, because it might happen that they become no longer Gauss-reduced with other elements in L thereafter. These elements are therefore brought to stack S (and reduced against $\mathbf{v}$) and picked in the subsequent iteration, as if they were freshly generated vectors, thus becoming pairwise-reduced with the whole list L.

Previous studies. The original paper of ListSieve and GaussSieve does not show tests pertaining to the workflow of the algorithm, since it only aimed at proving that GaussSieve outperformed another SVP-solver known at that time, NVSieve [11]. The authors showed (1) the number of samples that GaussSieve requires to converge, in comparison to NVSieve, which fell into disuse ever since, and (2) the execution runtime of GaussSieve in comparison to NVSieve and Schnorr-Euchner enumeration in NTL[3], when solving the SVP on lattices in various dimensions.

Since ListSieve and GaussSieve were published, only one study about the practical behaviour of GaussSieve was presented, by Schneider [13], and no studies were published on ListSieve. In particular, Schneider investigated the following parameters of GaussSieve:

- its performance, on various types of lattices, namely ideal, cyclic and random lattices. GaussSieve's performance, in terms of runtime, iterations, list size and collisions, was not affected by the type of the underlying lattice.

[3] http://www.shoup.net/ntl/

Algorithm 1: ListSieve

```
Input: Basis $\mathbf{B}$, stopping criterion $c$;
Init.: $L \leftarrow \{\}$

while $K < c$ do
    $\mathbf{p} \leftarrow$ SampleKlein($\mathbf{B}$);
    $\mathbf{v} \leftarrow$ ListReduce($\mathbf{p}$,$L$);
    if $||\mathbf{v}||=0$ then
        $K \leftarrow K + 1$;
    else
        $L \leftarrow L \cup \{\mathbf{v}\}$;
return BestVector($L$);

function ListReduce($\mathbf{p}$,$L$)
while $\exists \mathbf{v}_i \in L : ||\mathbf{p}-\mathbf{v}_i|| \leq ||\mathbf{p}||$
$\wedge\ ||\mathbf{p}|| \geq ||\mathbf{v}_i||$ do
    $\mathbf{p} \leftarrow \mathbf{p}-\mathbf{v}_i$;
return $\mathbf{p}$;
end function

function BestVector($L$)
return $\mathbf{p}$ : $\forall \mathbf{v} \in L$, $||\mathbf{p}|| < ||\mathbf{v}||$;
end function
```

Algorithm 2: GaussSieve

```
Input: Basis $\mathbf{B}$, stopping criterion $c$;
Init.: $L \leftarrow \{\}, S \leftarrow \{\}, K \leftarrow 0$

while $K < c$ do
    if S.size()!=0 then
        $\mathbf{v} \leftarrow$ S.pop();
    else
        $\mathbf{v} \leftarrow$ SampleKlein($\mathbf{B}$);
    $\mathbf{v} \leftarrow$ GaussReduce($\mathbf{v}$,$L, S$);
    if $||\mathbf{v}||=0$ then
        $K \leftarrow K+1$;
    else
        $L \leftarrow L \cup \{\mathbf{v}\}$;
return BestVector($L$)

function GaussReduce($\mathbf{p}$,$L, S$)
while $\exists \mathbf{v}_i \in L : ||\mathbf{v}_i|| \leq ||\mathbf{p}|| \wedge ||\mathbf{p}-\mathbf{v}_i|| \leq ||\mathbf{p}||$ do
    $\mathbf{p} \leftarrow \mathbf{p}-\mathbf{v}_i$;
while $\exists \mathbf{v}_i \in L : ||\mathbf{v}_i|| > ||\mathbf{p}|| \wedge ||\mathbf{v}_i-\mathbf{p}|| \leq ||\mathbf{v}_i||$ do
    $L \leftarrow L \setminus \{\mathbf{v}_i\}$;
    S.push($\mathbf{v}_i-\mathbf{p}$);
return $\mathbf{p}$;
end function
```

- (1) the number of vectors removed from the list and pushed to the stack and (2) the reductions, i.e., the number of vectors used to reduce a vector generated with Klein's algorithm. Schneider concluded that both are approximately ten times the list size. This means that, on average, ten points are used to reduce each vector, and the same number reduced and removed from the list. Considering an exponential list size, this amount is negligible.
- the quality of the best vector in GaussSieve over time. The norm of this vector decreases only a few times during the execution of the algorithm.

Additionally, Schneider found out that collisions happen only once the shortest vector of the lattice is found, growing exponentially from then on. When the norm of the shortest vector is known, it has been shown that the algorithm can be greatly accelerated. Last but not least, it has been shown that lattice reduction affects GaussSieve in a positive manner, but to a much lesser degree than it affects enumeration algorithms.

While these experiments provide important insight about the practical behaviour of GaussSieve, they neither show how GaussSieve and ListSieve compare regarding the selected parameters nor they cover the whole spectrum of parameters of interest. The trials reported in this paper cover the following relevant additional parameters, in ListSieve (**LS**) or both algorithms (**LS-GS**):

- (**LS-GS**) The over-time progression of the *b*est vector (the shortest vector in L at every instant). This analysis was previously done for GaussSieve only.
- (**LS-GS**) Comparison of the algorithms in terms of runtime.
- (**LS**) The maximum number of vectors used in a sample reduction, and the position of the latest used vector among all iterations.
- (**LS**) The performance of ListSieve in parallel and its scalability on CPUs.

3 Analysis of ListSieve and GaussSieve

This section presents the results of our study on ListSieve and GaussSieve. For GaussSieve, we used a publicly available version[4], referred to as the *gsieve* library, from which we also generated ListSieve, by removing specific operations. For the sake of fairness, both implementations use dynamic data structures and have no optimizations but the ones provided by `gcc -O2`.

Section 3.1 shows the progression of the best vector on two different Goldstein-Mayer lattices, in dimensions 50 and 60, available from the svp-challenge website[5]. For determining the function that governs the runtime of the algorithms, detailed in Section 3.2, we used a broader spectrum of lattices (dimensions 50 to 66, in steps of 2). We investigate further properties of ListSieve and GaussSieve, on lattices in dimensions 40, 50 and 60, in Section 3.3. The experiments were conducted on a server equipped with 2 Intel E5-2670 CPU-chips, each with eight 64-bit instruction set cores equipped with Simultaneous Multi-Threading (SMT), running at 2.60 GHz, and with 128 GB of RAM. The machine runs Ubuntu 11.10, and no other user-level processes were running during the trials.

3.1 Quality of the best vector over time

From here on, let the term *best vector* be the shortest vector that an algorithm knows at a given point in time. Once the algorithm ends, this vector will coincide with a shortest vector of the lattice, unless the shortest vector is not found. The interest of studying its progression over time is twofold. First, it provides insight about the smoothness of the algorithm, and identifies possible discontinuities in its progression. Second, it is essential to determine the progression of the quality of its solution, since very short vectors can suffice to break cryptosystems.

In these experiments, ListSieve and GaussSieve ran on lattices that were BKZ-reduced with blocksize 10. Bigger blocksizes rend the assessment of the progression of the algorithm over time useless, since BKZ almost finds the shortest vector per se. We depict the norm of the first vector in L (the current *best vector*), at the end of each iteration, i.e., after the generation of the random vector as well as its reduction (pairwise-reduction in GaussSieve). Figure 1 shows the evolution of the quality of the *best vector* in ListSieve and GaussSieve, over time, for lattices in dimensions 50 and 60, respectively. Not surprisingly, GaussSieve converges faster than ListSieve. Nonetheless, the total number of changes of the *best vector* during a run is of the same order. Although the smoothness of the algorithms are somewhat alike, GaussSieve finds new *best vector*s faster.

As shown in Figure 1, no new vectors are found until roughly half of the total running time, for both algorithms. The total running time for ListSieve (resp. GaussSieve) in dimension 50 is 52 (resp. 9.8) seconds. However, a shortest vector is already found after 30.16 (resp. 7.94) seconds. The reason why both algorithms do not terminate at that point is because the norm of the shortest

[4] `http://cseweb.ucsd.edu/~pvoulgar/impl.html`
[5] `http://www.latticechallenge.org/svp-challenge/`

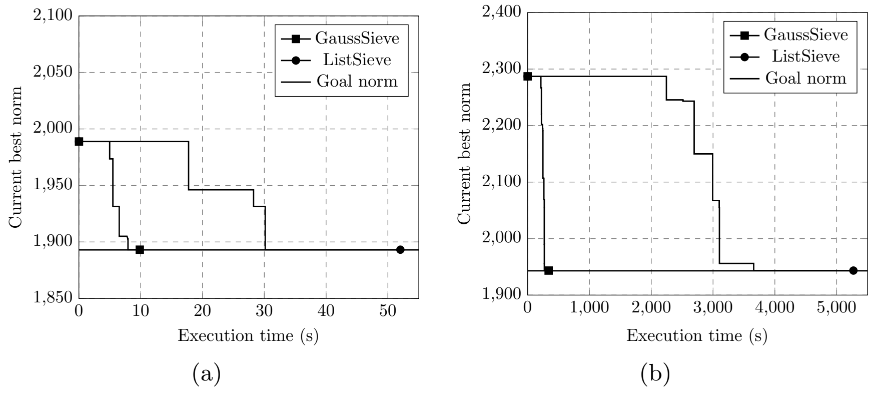

Figure 1. Best vector's quality over time in ListSieve and GaussSieve, for lattices in dimension 50 in (a) and 60 in (b).

vector (i.e., λ_1) is not known upfront for a given lattice. In fact, the algorithms terminate if a given number of collisions take place (cf. Section 2). If λ_1 was known upfront, ListSieve (resp. GaussSieve) could be sped up by a factor of 1.72 (resp 1.23). In dimension 60, the results are very similar.

3.2 Runtime complexity

We conducted a sequence of trials to empirically determine the growth of List-Sieve's runtime, in comparison to GaussSieve. Prior to these trials, it had been shown that the empirical growth of GaussSieve was governed by $2^{0.57n-23.5}$, for a lattice in dimension n, on a given architecture [13], but no comparison with ListSieve was provided.

Our trials show that the execution time of ListSieve grows according to $2^{0.58n-22.33}$, whereas GaussSieve's grows according to $2^{0.568n-25.46}$. To arrive at this model, we ran the algorithms with several lattices, in dimensions 50-66, in steps of 2. Figure 2 shows both the runtime of ListSieve and GaussSieve and the total number of iterations required for convergence. The execution time of both algorithms differs by an (almost) constant factor. The same holds for the total number of iterations.

3.3 Used vectors, list size and iterations

In both algorithms, L is consulted in every iteration to reduce the sampled vector. The longer this list, the longer the runtime, unless the algorithm only accesses the vectors ultimately selected for reduction, regardless of L's size.

To partially overcome this problem, these algorithms stop accessing vector $\mathbf{v}_{i+1}$ and subsequent ones when vector $\mathbf{v}_i$ is not suitable for the reduction process due to its norm. To this end, the vectors in L must remain ordered by increasing norm, and it is additionally assumed that a vector $\mathbf{p}$ is not reduced against a

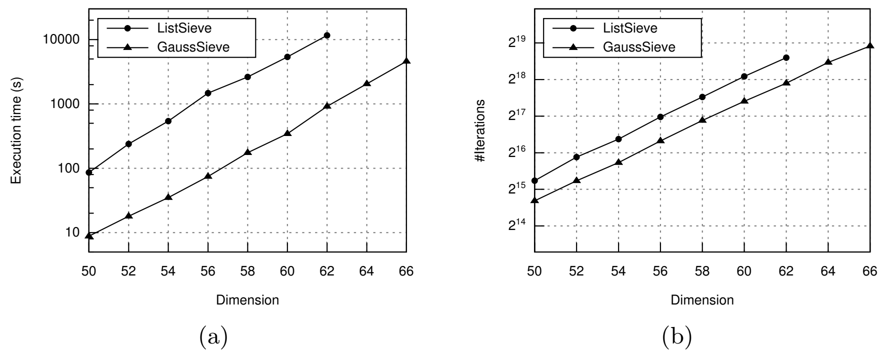

Figure 2. Runtime, in (a), and number of iterations, in (b), for ListSieve and GaussSieve, for lattices in dimensions spanning from 50 to 66, in steps of 2.

vector **v** in the list, if **v**'s norm is larger than **p**'s. While checking L for suitable vectors becomes less of a problem with this optimization, both the complexity and time of adding vectors to L are increased. As both lookups and insertions in L occur at every iteration, except for collisions, this does not represent an improvement in terms of complexity. In short, this only shifts the problem, it does not solve it.

Ideally, L would only keep vectors that are ultimately selected for the reduction process. While maintaining the benefits of an ordered list, it would also avoid (expensive) insertions of worthless vectors in L. To this end, one could set a norm bound, after which vectors would be discarded. However, the selection of a bound is not a simple task. First, no good bound is known upfront and experiments should be conducted to empirically determine one. Second, the precise impact of not using all the possible vectors in the reduction process is yet to be determined, although this greatly affected GaussSieve in a negative way [12].

To this end, we verified how many vectors are actually used during an execution of ListSieve and what is the latest element in L that sampled vectors are reduced against. This is very difficult to determine in GaussSieve because vectors might fluctuate between the list and the stack, and their reduction process is not concluded at the end of each iteration. Table 1 shows the final and max list size, the iterations of each algorithm, and two additional parameters that we analyzed for ListSieve and GaussSieve:

- *Max. used vectors*, which indicates the maximum number of vectors in L used to reduce a sampled vector, among all the iterations of the algorithm.
- *Latest used vector*, which indicates the latest position in the list that the randomly generated vector was reduced against.

As explained in Section 2, the size of L in GaussSieve changes during its execution. Table 1 shows the maximum list size in GaussSieve's execution. The number of iterations for convergence grows slightly faster for ListSieve than

Algorithm	**ListSieve**			**GaussSieve**		
Lattice dimension	40	50	60	40	50	60
Max. used vectors	83	109	153	Not applicable		
Latest used vector	2969	14241	77125	Not applicable		
Final list size	5748	39385	271766	Not relevant		
Max list size	Not applicable			1130	4182	17826
Iterations	6523	43474	299083	5044	28777	184790

Table 1. Stats for ListSieve and GaussSieve, for lattices in dimension 40, 50 and 60.

for GaussSieve. That is, while ListSieve requires ≈30% more iterations than GaussSieve for dimension 40, that factor is ≈50% (resp. ≈60%) in dimension 50 (resp. 60). Moreover, only a small number of vectors are used for the reduction process in ListSieve. For instance, only at most 83 vectors are used in the reduction process of the lattice in dimension 40, while the list contains 5748 vectors at the end of the execution. Interestingly, the number of used vectors merely doubles from dimension 40 to 60, whereas the list size becomes 47 times bigger.

Another important observation is that ListSieve does not use any vector for reduction after a certain position. The latest used vector was only roughly at the middle of the list for dimension 40, and seems to be smaller for higher dimensions. As a result, it might be possible to set a limit of vectors that L holds, without impairing performance, an issue which we will investigate in the future.

4 Parallel implementation of ListSieve

In contrast to GaussSieve, which has been parallelized on multi-core CPU-platforms [10,12], there are no studies concerning the parallelization of ListSieve. We think that the main reason for the lack of studies addressing the parallelization of ListSieve is the (unverified) belief in its impracticality. There are essentially three reasons for studying the parallelization of ListSieve. First, as we show in this paper, the performance variations between ListSieve and GaussSieve are not as big as thought. Second, it was previously suggested that parallel versions of ListSieve could possibly outperform parallel versions of GaussSieve [12], a claim that we address first-hand. Third, ListSieve is much easier to port to parallel architectures, such as GPUs.

We implemented and assessed the performance of a parallel version of ListSieve, written in C, which makes use of OpenMP to manage the execution of threads. The list L was implemented as a singly linked list, where each element points to its successor. Each thread follows the workflow of the original algorithm: they sample a vector $\mathbf{p}$, reduce it against every element $\mathbf{v}$ in L, thereby generating $\mathbf{p'}$, and insert $\mathbf{p'}$ in the list L. To avoid the use of synchronization, each thread inserts an element $\mathbf{p'}$, between two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, in the list L, by setting $\mathbf{p'}$'s next pointer pointing to $\mathbf{v}_2$ and then setting $\mathbf{v}_1$'s next pointer pointing to $\mathbf{p'}$.

This relaxes the properties of the algorithm, which results in a smaller reduction process. There are two relaxations that *might* occur. First, (1) a given thread t_1 might be reducing its sample $\mathbf{p}_1$ in position k_1 of the list L while another thread t_2 inserts a vector $\mathbf{p}_2$ in position k_2, with $k_2 < k_1$. As a result, the reduction of $\mathbf{p}_1$, by thread t_1, will not take the vector $\mathbf{p}_2$ into account in that iteration, which we refer to as a *missed reduction*. Note that the vector will be visible in the following iteration. Second, (2) a sample is lost if two threads try to insert their samples at contiguous positions of L.

We found out that this synchronization-avoiding relaxation of the properties of ListSieve did not change the quality of the output, since the output of every run of our parallel ListSieve implementation was identical to the output of the *gsieve* library. This is not completely surprising, as (1), i.e. missing reductions, does not seem problematic because reductions will only be missed in a specific iteration (if missed at all), and the reduction could actually be unsuccessful in first place and (2), i.e. losing vectors, is very unlikely to happen, due to the length of the list, and the fact that threads are not likely to insert vectors in L at the same time.

The stopping criterion of the implementations is as defined in Section 2, set up with $\alpha = 0.1$ and $\beta = 200$. The code was compiled with `g++ 4.6.1` (since NTL, used for BKZ, is written in C++) with the optimization flag `-O2`, which showed to be slightly better than `-O3`. Lattices were reduced with BKZ, with blocksize 20. Every experiment was repeated three times and the best sample was chosen. The elapsed time of lattice reduction is not included.

Figure 2 shows the execution time of our parallel version of ListSieve, for lattices in dimensions 40, 50 and 60, with 1-32 threads, on the test platform described in Section 3. As shown in Table 2, the speedup and efficiency are quite modest for dimension 40, because there is not enough work to compensate for the parallel execution overhead. For bigger lattices, the speedup is super-linear for all cases except for 32 threads in dimension 50, which concerns the use of SMT. In fact, with SMT, the efficiency drops for the three lattices. For dimension 60, the speedup and efficiency seem to grow with the number of threads, which means that the more threads are used, the more the properties of ListSieve are relaxed. As a result, it might happen that, for a very high number of threads,

	Dimension 40			Dimension 50			Dimension 60		
Threads	R	S	E	R	S	E	R	S	E
1	1.1228	1.00	100%	33.7169	1.00	100%	2210.7188	1x	100%
2	0.4861	2.31x	116%	13.8189	2.44x	122%	770.3057	2.87x	144%
4	0.2587	4.34x	109%	6.1101	5.52x	138%	326.5866	6.77x	169%
8	0.2384	4.70x	59%	3.0440	11.08x	139%	150.8266	14.66x	183%
16	0.2657	4.23x	26%	1.9017	17.73x	111%	75.8777	29.14x	182%
32	0.2414	4.65x	15%	1.7373	19.41x	61%	49.1252	45.00x	141%

Table 2. Runtime (R) in seconds, Speedup (S) and Efficiency (E) of our implementation on three lattices. The grayed out row, for 32 threads, concerns the use of SMT.

the number of missed reductions becomes problematic and more iterations are required for convergence, which will impair scalability.

We also compared our implementation with the parallel GaussSieve implementation described in [12], from here on referred to as *Milde2011*. The code was provided by the authors. The implementation makes use of a ring structure connecting several instances of GaussSieve, each containing a local list, and a private stack S. Each thread samples a new vector $\mathbf{p}$ and reduces it against the elements in its local list. Afterwards, $\mathbf{p'}$ is handed over to the next thread which itself reduces the vector further against the elements in its local list. When the vector returns to the thread that released it, it is added to the local list of that thread.

Figures 3(a) and 3(b) compare the performance of our implementation and *Milde2011*, on lattices in dimensions 60 and 70, respectively. For the sake of convenience, the lattice in dimension 70 was BKZ-reduced with blocksize 32, for both implementations. For the lattice in dimension 60, *Milde2011* scales only up to 4 threads, and with 32 threads, at a limited rate. As our implementation scales super-linearly, it beats GaussSieve for 8 and more threads. For the lattice in dimension 70, *Milde2011* scales better, but also at a much lesser degree than our implementation. As a result, our implementation outperforms *Milde2011* for more than 8 threads. Note that GaussSieve is clearly faster for 1 thread, in both cases. This result indicates that ListSieve is indeed a practical SVP-solver and might take more advantage of massively parallel architectures than GaussSieve.

5 Conclusions

In this paper, we presented the results of a comprehensive empirical comparison of ListSieve and GaussSieve, two sieving algorithms that are very relevant in lattice-based cryptography. Although ListSieve has been considered impractical, we show that it can indeed be practical, especially on parallel platforms.

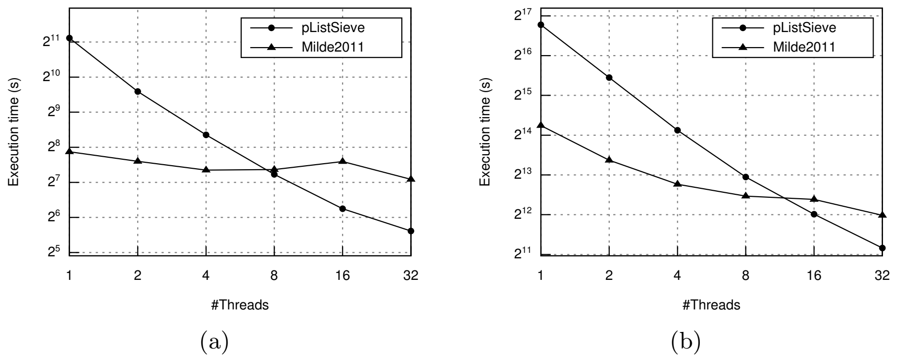

Figure 3. Comparison of our ListSieve implementation (*pListSieve*) and *Milde2011* for a BKZ-reduced lattice with blocksize 20 in dimension 60 (a) and 70 in (b).

Another surprising discovery is that ListSieve only uses a small number of the stored vectors, a ratio that decreases with the dimension of the used lattice. This is very important because it might be used to reduce the memory usage of the algorithm, a critical problem of sieving algorithms.

Our parallel implementation of ListSieve relaxes the properties of the algorithm, allowing (1) some vector reductions to be missed and (2) some vectors to be lost, for the sake of reduced synchronization overhead. It achieves superlinear speedups on a multicore CPU-chip for up to 32 threads. In particular, an efficiency level of 182% is achieved for 16 threads on a lattice in dimension 60. As a result, it outperforms the parallel *Milde2011* GaussSieve implementation for a large number of threads, a question posed in [12] that was open to this day. Among other factors, this result is particularly relevant because the algorithm is a better candidate than GaussSieve to run on massively parallel architectures, such as GPUs, since it has fewer dependencies, especially if the properties of ListSieve are relaxed as we propose in this paper.

References

1. Ajtai, M.: Generating hard instances of lattice problems (extended abstract). In: STOC '96. pp. 99–108. ACM (1996)
2. Ajtai, M.: The Shortest Vector Problem in L2 is NP-hard for Randomized Reductions (Extended Abstract). In: STOC '98. pp. 10–19. ACM, NY, USA (1998)
3. Fitzpatrick et al.., R.: Tuning GaussSieve for Speed. In: LATINCRYPT'14. Florianópolis, Brazil (September 2014)
4. Dagdelen, Ö., Schneider, M.: Parallel enumeration of shortest lattice vectors. In: Euro-Par 2010, LNCS, vol. 6272, pp. 211–222. Springer (2010)
5. Detrey, J., et al.: Accelerating Lattice Reduction with FPGAs. In: LATINCRYPT '10, LNCS, vol. 6212, pp. 124–143. Springer (2010)
6. Gama, N., et al.: Lattice enumeration using extreme pruning. In: EUROCRYPT '10. LNCS, vol. 6110. Springer (2010)
7. Klein, P.: Finding the closest lattice vector when it's unusually close. In: SODA '00. pp. 937–941 (2000)
8. Kuo, P.C., et al.: Extreme Enumeration on GPU and in Clouds. In: CHES 2011, LNCS, vol. 6917, pp. 176–191. Springer (2011)
9. Lenstra, A., et al.: Factoring polynomials with rational coefficients. Mathematische Annalen 261(4), 515–534 (1982)
10. Mariano, A., et al.: Lock-free GaussSieve for Linear Speedups in Parallel High Performance SVP Calculation. In: SBAC-PAD'14. Paris, France (2014)
11. Micciancio, D., Voulgaris, P.: Faster exponential time algorithms for the shortest vector problem. In: SODA '10. pp. 1468–1480. PA, USA (2010)
12. Milde, B., Schneider, M.: A parallel implementation of GaussSieve for the shortest vector problem in lattices. In: PaCT'11. pp. 452–458. Springer (2011)
13. Schneider, M.: Analysis of Gauss-Sieve for Solving the Shortest Vector Problem in Lattices. In: WALCOM '11, LNCS, vol. 6552, pp. 89–97. Springer (2011)
14. Schnorr, C., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. Math. Programming 66(1-3), 181–199 (1994)
15. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. 26(5), 1484–1509 (1997)