A Parallel Variant of LDSieve for the SVP on Lattices

Artur Mariano
Institute for Scientific Computing
Technische Universität Darmstadt
Germany

Thijs Laarhoven
IBM Research
Zürich, Switzerland

Christian Bischof
Institute for Scientific Computing
Technische Universität Darmstadt
Germany

Abstract—**In this paper, we propose a parallel implementation of LDSieve, a recently published sieving algorithm for the SVP, which achieves the best theoretical complexity to this day, on parallel shared-memory systems. In particular, we propose a scalable parallel variant of LDSieve that is probabilistically lock-free and relaxes the properties of the algorithm to favour parallelism. We use our parallel variant of LDSieve to answer a number of important questions pertaining to the algorithm. In particular, we show that LDSieve scales fairly well on shared-memory systems and uses much less memory than HashSieve on random lattices, for the same or even less execution time.**

I. Introduction

In the late 90s, the news broke that classical public-key cryptosystems (usually referred to as *schemes*) such as RSA are vulnerable against attacks conducted with quantum computers. Although there are no full purpose quantum computers at this point in time, the cryptography community has been devoting substantial efforts to the development of quantum resistant (usually referred to as *post-quantum*) cryptosystems. Very recently, NSA has also announced preliminary plans to transition to quantum-resistant schemes[1]. The security of post-quantum cryptosystems is being investigated in parallel, as that is essential to create confidence on the cryptosystems.

Lattices are discrete subgroups of the n-dimensional Euclidean space $\mathbb{R}^n$, with a strong periodicity property. A lattice $\mathcal{L}$ generated by a basis $\mathbf{B}$, a set of linearly independent vectors $\mathbf{b}_1$,...,$\mathbf{b}_n$ in $\mathbb{R}^n$, is denoted by:

$$\mathcal{L}(\mathbf{B}) = \{\mathbf{x} \in \mathbb{R}^n : \mathbf{x} = \sum_{i=1}^{n} u_i \mathbf{b}_i,\ \mathbf{u} \in \mathbb{Z}^n\}. \qquad (1)$$

The security of these systems is, as in classical cryptosystems, based on hard math problems. The Shortest Vector Problem (SVP) is a pivotal problem in this context, as many lattice-based cryptosystems base their security on (i) this problem, (ii) variations of this problem and/or (iii) problems that can be reduced to the SVP. The SVP consists in finding the non-zero vector $\mathbf{v}$ of a given lattice $\mathcal{L}$, whose Euclidean norm $\|\mathbf{v}\|$ is the smallest among the norms of all non-zero vectors in the lattice $\mathcal{L}$. Its norm is denoted by $\lambda_1(\mathcal{L})$. We refer to an algorithm that solves this problem as an *SVP-solver*.

[1]https://www.nsa.gov/ia/programs/suiteb_cryptography/

It is essential to estimate the actual computational hardness of these problems in practice, as the parameters of cryptosystems are chosen based upon this hardness. Both over- and underestimating this hardness (i.e. estimating that the problem is harder or easier than what in reality it is) is problematic. Overestimations compel cryptographers to set overly strong parameters, which decreases the efficiency of the scheme and may ultimately render it impractical. On the other hand, underestimating this complexity misguides cryptographers in selecting parameters, ultimately rendering the schemes insecure. Usually, the only way to estimate the actual hardness of these problems is to develop and assess highly optimized, parallel solvers (i.e. *attacks*) in practice.

There are a few classes of SVP-solvers, one of which is sieving. In the last years, sieving algorithms have been enhanced to become more efficient in practice. Very recently, a new sieving algorithm, LDSieve, was proposed [1]. The first and perhaps most relevant question concerning LDSieve, is whether it can beat HashSieve, the best sieving algorithm for the SVP to date, in practice [5], [8]. In particular, it is relevant to assess this claim (1) with sequential implementations of the algorithms, (2) that are equally optimized, (3) verifying if LDSieve lends itself to parallelism and finally (4) if so, whether a parallel implementation of LDSieve compares well to a parallel version of HashSieve, which is known to scale well on shared-memory systems [8].

Our contributions. In this paper, we propose a parallel variant of LDSieve, which scales fairly well on shared-memory systems, and we answer the four aforementioned questions. To this end, we conduct various benchmarks with our parallel implementation and the original implementation of LDSieve (cf. [1]). In addition, we provide insight on the selection of parameters of the algorithm such as the codesize and memory consumption.

II. LDSieve

The recent paper [1] introduced a novel way to find nearby vectors in a large set of vectors in Euclidean space, and showed how this affects the performance of lattice sieving. Similar to HashSieve [5], this technique can be applied to GaussSieve [11] to speed up the algorithm, at the cost of more space. Figure 1 shows the asymptotic tradeoffs for different

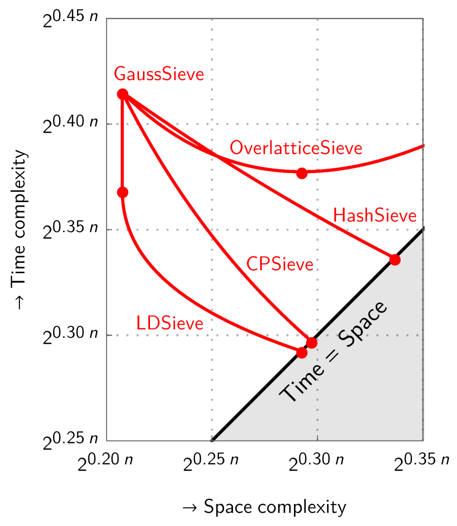


Fig. 1: Theoretical time and space complexities of different sieving algorithms.

GaussSieve-based sieving algorithms (GaussSieve, HashSieve, CPSieve [3], LDSieve [1] and OverlatticeSieve [2]).

A. *GaussSieve*

Let us first recall the description of the GaussSieve algorithm. To find shortest vectors, the algorithm iteratively samples lattice vectors using a sampling algorithm, such as Klein's sampler [4]. Each time a new vector $\vec{v}$ is sampled, it is reduced against the list of previously sampled/reduced lattice vectors $\vec{w}$ as follows: if $\vec{v} \pm \vec{w}$ is shorter than $\vec{v}$, then we replace $\vec{v}$ by $\vec{v} \pm \vec{w}$. Note that adding or subtracting two lattice vectors always results in another lattice vector. This is done until no vector in our list can reduce $\vec{v}$ anymore, in which case we reduce the list of vectors with our new, short vector $\vec{v}$. Vectors from the list which are reduced in norm are pushed to a stack, for later consideration. At the end, the new vector $\vec{v}$ is added to the list, and we continue the same procedure either with vectors from the stack or, if the stack is empty, with new sampled vectors.

The main idea of this algorithm is to saturate the space of short lattice vectors with our list. By keeping the list *pairwise reduced* using the above procedure, it is guaranteed that the list size will not exceed $(4/3)^{n+o(n)} \approx 2^{0.21n}$. As the list will at some point reach a size of $2^{0.21n}$, and a reduction is done by checking the entire list of past vectors for reductions with the target $\vec{v}$, the time complexity is bounded from below by $2^{0.21n}$ reductions of time $2^{0.21n}$ each. This leads to a space complexity of $2^{0.21n}$ and a time complexity of $2^{0.42n}$, as indicated by the top-left point in Figure 1.[2]

B. *Nearest-neighbor searching*

To improve upon the GaussSieve algorithm, HashSieve and LDSieve use a fine-grained *nearest-neighbor search* technique to reduce the cost of reductions. The first observation is that many of the comparisons of a target $\vec{v}$ with a list vector $\vec{w}$ do not lead to a reduction; in many cases $\vec{v}$ and $\vec{w}$ are approximately orthogonal, and so $\vec{v} \pm \vec{w}$ is longer than $\vec{v}$. In GaussSieve, we therefore spend a lot of time on comparisons that do not to lead to reductions.

The idea of nearest-neighbor searching is to use a bucketing approach as follows. Besides storing the list vectors in memory, we also store many buckets with vectors which satisfy a certain property. This property is chosen such that vectors which are nearby in space are more likely to share this property. Each time a new vector is stored in memory, it is also stored in all buckets for which it satisfies this property. Then, given a target vector $\vec{v}$, we do not compare it to all list vectors $\vec{w}$, but only to those which are in the same bucket as $\vec{v}$ and are therefore *nearby* in space. This concept allows us to do the search for reductions faster: instead of checking all vectors $\vec{w}$ against $\vec{v}$, we only compare $\vec{v}$ with a small subset of vectors $\vec{w}$ which are in the same buckets. This comes at the cost of a larger memory requirement (as buckets have to be stored).

C. *LDSieve*

LDSieve differs from HashSieve and CPSieve in the way the buckets are designed (i.e. the property defining when a vector is to be added to a bucket), and in the method for finding the right buckets. The design of the buckets is quite straightforward: choose a random direction in space (by e.g. sampling a vector $\vec{c}$ at random from the unit sphere), and a vector is now added to this bucket if its normalized inner product is larger than some constant α: $\frac{\langle \vec{v},\vec{c}\rangle}{\|\vec{v}\|\cdot\|\vec{c}\|} \geq \alpha$. Intuitively, if both $\frac{\langle \vec{v},\vec{c}\rangle}{\|\vec{v}\|\cdot\|\vec{c}\|}$ and $\frac{\langle \vec{w},\vec{c}\rangle}{\|\vec{w}\|\cdot\|\vec{c}\|}$ are large, then we also expect $\frac{\langle \vec{v},\vec{w}\rangle}{\|\vec{v}\|\cdot\|\vec{w}\|}$ to be large, and so $\vec{v} - \vec{w}$ is likely to be shorter than $\vec{v}$ or $\vec{w}$.

As using completely random, independent buckets (defined by the vectors $\vec{c}$) would lead to a large overhead of finding the buckets that a vector is in (as many inner products have to be performed), the decoding procedure and the way the random vectors $\vec{c}$ are chosen is somewhat complicated. Instead of using T random vectors $\vec{c}_1, \dots, \vec{c}_T$, we add some structure to these vectors by choosing a random subcode $S \subset \mathbb{R}^{n/m}$ of size $T^{1/m}$, and defining the code words $C = \{\vec{c}_1, \dots, \vec{c}_T\}$ as the product code $C = S \times S \times \cdots \times S = S^m$. In other words, a code word $\vec{c}_i$ is formed by concatenating m code words $\vec{c}_{i_1}, \dots, \vec{c}_{i_m} \in S$. By choosing m small (i.e. *almost*-constant; say $m = O(\log n)$), it is guaranteed that the product code S^m is almost random as well [1, Theorem 5.1], while taking m super-constant allows us to find the right buckets corresponding to a vector with almost no overhead: if we have t buckets in total and $t_0 \ll t$ buckets contain $\vec{v}$, then the time for finding these buckets is proportional to t_0 rather than to t (cf. [6] for a discussion on m). This *efficient decodability* is crucial for obtaining an improved performance.

Summarizing, the LDSieve starts by generating a random subcode S in dimension $n/m = O(n/\log n)$ of size $t^{1/m}$,

[2]For further details why the number of reductions per vector is $2^{o(n)}$, where the space complexity bound comes from, and why *collisions* do not often occur, we refer the reader to previous literature on the topic, e.g. [10].

which defines the concatenated product code $C = S^m$ of size t in dimension n. Then, given a target $\vec{v}$, we first compute the partial inner products of its m subvectors of dimension n/m with the entire code C, and store these in partial inner product lists $L_1, ..., L_m$. The efficient decoding algorithm described in [1] then continues with sorting each of these lists based on the sizes of the inner products, and uses a depth-first search in a lattice enumeration-like tree to find all buckets corresponding to vectors $\vec{c}$ in C such that the normalized inner product between $\vec{v}$ and $\vec{c}$ is larger than α. These buckets in memory are then checked for reductions. Note that the overhead of this decoding procedure is theoretically subexponential in the lattice dimension n, which justifies the use of the words "efficient decoding"; however, as we will see later, in practice the generation of these partial inner product lists (and sorting these lists) is quite time-consuming.

After the reductions are completed, the new vector is also added to these buckets in memory, and a new vector is popped from the stack or generated using the sampler. Note that we have not yet defined m, T, α; these are parameters which are to be optimized later.

III. A PARALLEL VARIANT OF LDSIEVE

In this section, we introduce a parallel variant of LDSieve and its assessment. Our variant is parallel at a coarse-grained level, where each thread executes the sequential sieving kernel, i.e., generation of a sample (either from scratch or popped from the stack), reduction against existing samples, and storing in memory (e.g. in a global list, as in GaussSieve or in hash tables, as in HashSieve). This scheme is suited for shared-memory systems, where memory is visible to all running threads, which implies that concurrent accesses are explicitly handled.

A. Concurrency in LDSieve

In HashSieve (cf. [8]), concurrency happens at two distinct points: (1) when concurrently inserting or removing vectors from each bucket, and (2) when concurrently accessing a vector stored in the buckets to reduce it and use it to reduce other vectors. It is important to note that while (1) happens only when two or more threads access the same hash table (and the same bucket), (2) can happen even if different threads are working on different hash tables, as vectors are pointed to by all hash tables in the system.

When it comes to concurrency, LDSieve is similar to HashSieve: the concurrent operations are additions/removals of vectors from the same buckets, and the concurrent use (with at least one thread writing it) of vectors. In addition, different tables for the inner products between samples and codes have to be used by different threads. Although LDSieve uses a single hash table, in contrast to HashSieve, the problem remains as different threads can access the same buckets and the same vectors simultaneously. In particular, in LDSieve the number of pointers to each vector is governed by, approximately, $2^{0.09n}$ (i.e. for a lattice in dimension $n = 80$, there should be approximately 150 pointers to each vector). In HashSieve, each vector is pointed to by all the hash tables once, and the number of hash tables is governed roughly by $2^{0.129n}$ (i.e. for a lattice in dimension $n = 80$, there should be approximately 1300 pointers to each vector), which means that there are fewer pointers to vectors in LDSieve than in HashSieve, hence a lower likelihood of concurrency in parallel versions.

B. Variant's workflow

Our variant is, similarly to the implementation in [8], based on the concept of probable lock-freeness. The underlying idea of this model is that efficient locks are employed per bucket and vector, but as contention is very low, executions will run without actually using locks, hence becoming lock-free code. Contention is increased with increased numbers of threads and lower numbers of buckets. However, the number of buckets in LDSieve is big enough, for a reasonably high dimension (cf. T in Table I), so locks are never used.

In addition, some vectors are ignored during the reduction process if they are locked by another thread. It was empirically shown before that disregarding vectors at some specific points of the reduction process (in contrast to disregarding them from a specific point on) does not affect the convergence rate, because the vast majority of vectors is not eligible for reduction anyway [10], [9], [5].

We employ one efficient lock per vector and bucket, which threads use when accessing these structures. Threads spin when they cannot acquire the locks of the buckets. If the vectors are locked, they are disregarded and the reduction process moves onto the next candidate vector. As mentioned above, the likelihood of successful vector reductions is low, and if one vector is disregarded, the bucket can still contain other vectors to proceed with the reduction process.

Performance Optimizations. Based on the results of [7], we optimized our implementation, written in C, with the following techniques (which resulted in similar speedups as those reported in [7]).

First, we hand-vectorized (1) the inner products to test vector reductions, (2) the vector addition when a reduction is successful, and (3) the bucket calculation, to which end many partial inner products between the vector and the generator vectors are performed. We used SSE4.1, and we store 8 coordinates per SSE register as all these structures are arrays of `shorts`. We aligned data to 16 bytes for vectors, and we use unaligned loads/stores for partial inner products.

We use a pool of vectors, private per thread, to store samples as the algorithm creates them, whose benefit is two-fold. First, it minimizes the number of malloc calls. Second, it improves spacial locality, since vectors are consecutively stored in memory. We also use `ltcmalloc`, as it accelerates parallel memory allocation.

To hide latency of memory requests, we used software-based prefetching, with hand-inserted prefetch directives. For example, when a candidate vector (stored in the hash table) is reduced against one sample during the reduction process, one calculates the set of buckets that the candidate vector belongs to, then removes it from the buckets, one by one. Here, we prefetch the bucket of iteration i+1 when removing the candidate vector from one bucket in iteration i. As such, in iteration i+1, the data will already be in cache (or the latency will be smaller, since data is requested beforehand).

C. Assessment

The original paper about LDSieve reported on experiments with an implementation of the algorithm[3], which we refer to as the *baseline implementation* from here on. We carried out an extensive assessment of our implementation, measuring the performance against the baseline implementation, its scalability and other relevant factors, such as the best codesize in practice. Although many other tests could have been conducted, these are absolutely core to provide insight about LDSieve and its behavior in practice. We used several random lattices, generated with Goldstein-Mayer bases, in multiple dimensions, available on the SVP-challenge[4] website (all of which of seed 0). The test platform has two Intel E5-2698v3 chips, at 2300 MHz, each of which has 16 cores plus hiper-threading. The machine as 756 GBs of RAM.

The code was compiled with Intel icpc 13.1.3. We compiled the code with the -O2 optimization flag, since it was slightly better than -O3. The elapsed time of the reduction of the lattice basis is not included in the reported timings and we used the target norm, except when said otherwise, as this eases comparisons against the baseline implementation (since the experiments reported in the original paper were conducted with the target norm as well).

We deactivated the Turbo Mode, which allows cores to run faster than the marked frequency, and the Enhanced Intel Speedstep Technology (EIST), which dynamically adjusts the frequency and voltage based on performance and power requirements, in order to mitigate noise in benchmarking. Despite our precautions, the variations in most experiments (especially those with the parallel version) account for as much as 30%. To correct this, except when impractical, we repeated each experiment five times for the parallel implementation and three times for the sequential implementation, and we report the average of those runs.

1) Comparison against the baseline: We compare the performance of our LDSieve variant, for a single thread (as the baseline is not parallel), in terms of Time To Solution (TTS) and vectors to reach convergence, with that of the original paper. It is important to note that, with a single thread, our implementation behaves as the original algorithm as no reductions are missed. A prime point of a fair comparison is to use parameters that do not favour either of the implementations. The implementations use different numbers of vectors to reach convergence, as they use different samplers and other implementation details that lead to different convergence rates. To carry out a fair comparison, we conducted preliminary experiments to determine what parameters of our implementation led on to an identical workload and the closest possible number of vectors of the baseline implementation. In particular, we tested a number of values for the sampler parameter d (cf. [8] for a detailed explanation) that provides a trade-off between sampling quality and sampling execution time, up to dimension 60, and reported the execution that delivered the closest number of vectors to the baseline implementation. It should be noted that a lower number of vectors does not necessarily lead to better performance, even for the same implementation, because the convergence rate of sieving algorithms is not uniquely determined by the number of vectors used.

In Table I, we present and compare the performance of both implementations. A few results in these measurements deserve particular attention. First, the type of lattice reduction used before LDSieve has considerable more impact on our implementation than on the baseline implementation. The execution time even decreases when we change the reduction, from LLL to BKZ-30, despite we solve the SVP on a high dimension. We surmise this happens as the baseline implementation does not include the basis vectors in the algorithm, while ours does. Second, the number of vectors our implementation requires to converge is roughly the same until dimension 48. For higher dimensions, the implementations use different numbers of vectors, but the difference is bounded by a factor of 2.5x. Again, a lower number of vectors does not necessarily lead to better performance, so this is not a reason for better performance. Third, the difference of performance of both implementations grows with the lattice dimension. For instance, our implementation is 16x faster for the lattice in dimension 64, but 27x and 50x faster for dimensions 68 and 72, respectively.

2) Scalability: The measurements with parallel executions were considerably less stable than those with a single thread, with variations of as much as 30% between different runs. This is because the convergence rate depends upon the scheduling of threads, and even small variations in the rate at which short vectors are found have considerable impact on the overall progress of the algorithm. To mitigate noise, we run five samples per data point, and we chose the best of the runs. Figure 2 shows the scalability of our implementation, for lattices in dimensions 72, 76 and 80 (parameters defined as in Table I). We ran our implementation with 1-64 threads (64 threads makes use of SMT), with different thread/core affinities. No specific thread affinity scheme delivered better results consistently, although the interleave scheme, which we chose for the benchmarks with `numactl`, was slightly better for more than 16 threads. We do not report further on the differences between the schemes, as studying the effect of different thread affinities is out of the scope of our work.

The scalability of the implementation is fairly good for all numbers of threads. We obtain linear speedups up to 16 threads, and slightly lower speedups for 32/64 threads, presumably due to additional overhead of inter socket communication. Superlinear speedups were achieved in specific instances because thread scheduling can result in different, potentially faster reduction processes. We extrapolate that

[3]https://github.com/lducas/LDSieve/
[4]www.latticechallenge.org/svp-challenge/

			Baseline implementation		Our implementation					
N	**Reduction**	**Solution**	**TTS (s)**	**Vectors**	**TTS (s)**	**Vectors**	**Speedup**	*d*	C	T
40	LLL-0.99	$\sqrt{2898385}$	9.61	832	3.70	731	2.6x	21	40	40^4
44	LLL-0.99	$\sqrt{2825372}$	15.18	1220	6.38	1240	2.4x	22	40	40^4
48	LLL-0.99	$\sqrt{3222704}$	32.71	2288	10.77	2498	3x	25	40	40^4
52	LLL-0.99	$\sqrt{3633604}$	66.13	3866	14.66	5481	4.5x	27	40	40^4
56	LLL-0.99	$\sqrt{3894744}$	140.04	7158	32.83	13774	2.4x	30	40	40^4
60	BKZ-30	$\sqrt{3776807}$	355.18	14538	13.34	20298	26.6x	40	40	40^4
64	BKZ-30	$\sqrt{4423362}$	1294.16	30866	80.72	73457	16x	40	46	46^4
68	BKZ-30	$\sqrt{4585399}$	4656.58	53886	169.24	116147	27.5x	40	52	52^4
72	BKZ-30	$\sqrt{4749371}$	21002.40	104083	418.31	239811	50.2x	50	58	58^4
76	BKZ-30	$\sqrt{5020176}$	intractable		1851.46	571753	-	60	64	64^4
80	BKZ-34	$\sqrt{5162837}$	intractable		4320.90	1083952	-	70	70	70^4

TABLE I: Comparison between the baseline implementation and our implementation, running with one thread. M is set to 4. C is the size of the subcodes, so that C^M = T.

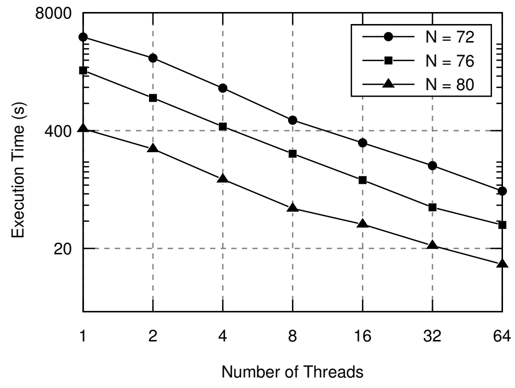


Fig. 2: Scalability of our implementation of LDSieve on the 32 core machine, for 1-64 threads (64 threads implies use of SMT).

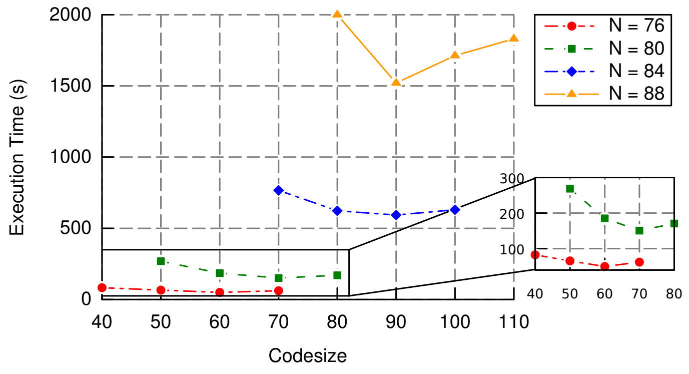
higher dimensions will result in identical scalability levels, as no further negative cache effects are expected to happen, since cache is already exhausted for these dimensions. Lower dimensions might result in weaker scalability, due to lack of enough work to overcome the overhead of thread creation and scheduling, but those are of lesser interest.

3) Codesize in practice: In theory, the ideal codesize is T $= 2^{0.292n+O(n)}$, where the $O(n)$ term is less relevant in practice, but may greatly influence the performance of the algorithm in practice. In this subsection, we show the effect of different codesizes on the execution time of the algorithm, for different lattices, to determine the best parameters in practice. Figure 3(a) shows the execution time of our implementation, running with 64 threads, on lattices in dimensions 76, 80, 84 and 88. Note the zoom-in into the range of N between 40 and 80 on the right side of the figure. The curves of the execution time for different lattice dimensions have the form of a parabola, i.e. the execution time decreases with increased codesize, but starts to increase after some point. In particular, all curves have a well defined minimum (60 for dimension 76, 70 for dimension 80 and 90 for dimensions 84 and 88). The difference in the execution time between the best codesize and the second best varies between 10 and 30%.

The codesize influences the number of vectors that the algorithm uses. However, the codesize that renders LDSieve the faster is not necessarily the codesize that leads to fewer vectors. Figure 3(b) shows the number of vectors (in thousands) used by our implementation as a function of the codesize. For instance, for the lattice in dimension 84, the implementation uses approximately 4.4, 2.6, 1.7 and 1.3 million vectors for codesizes 70, 80, 90 and 100, respectively. Although codesize 100 results in 30% fewer vectors than codesize 90, it is the latter that provides the best execution time. There are two fundamental reasons for this: (1) higher codesizes render the algorithm increasingly selective in the bucket selection, which means that fewer reductions are missed (smaller list size) but also more buckets must be checked for reductions, and (2) increasing codesize renders the partial inner product lists bigger, which in turn leads to more cache misses and more requests to higher levels of the memory hierarchy. For high codesizes, the inner product lists become a dominant factor in memory usage. In particular, the memory usage grows substantially with the codesize, in all dimensions. For instance, for a lattice in dimension 88, the implementation spends approximately 52.7 GBs of memory with codesize 80, and over 122 GBs for codesize 110.

IV. LDSieve vs HashSieve

The main motivation for implementing and appraising the practicability of LDSieve is that both theoretical analyses and preliminary experiments suggest that LDSieve can outperform the best sieving SVP algorithm in practice, HashSieve, for a sufficiently large lattice dimension. Preliminary experiments with LDSieve (cf. [1]) suggest that, for sequential implementations, LDSieve outperforms HashSieve for lattices in dimension 72 and onwards. An important goal of this paper



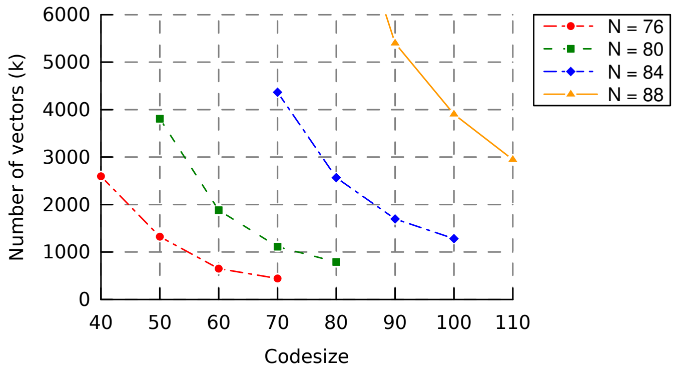


Fig. 3: (a) Execution time for different codesizes in LDSieve, for lattices in dimensions 76, 80, 84 and 88. (b) Vectors (in thousands) for different codesizes in LDSieve, for lattices in dimensions 76, 80, 84, 88 and 92. 64 threads. BKZ-34 and M = 4.

is to verify this claim when highly (but equally) optimized, parallel versions of LDSieve and HashSieve are compared against each other.

To carry out this comparison, we used the HashSieve implementation by Mariano et al. (cf. [8] and Section III-A), as it is the best known parallel implementation of HashSieve (and sieving algorithms) at present. We report on (i) the impact of the codesize in LDSieve's performance, (ii) the execution time of both implementations, (iii) the number of vectors used and (iv) memory consumption. We conducted the comparison for high dimensions (80 and onwards) as high dimensions are of particular interest.

An important caveat in the performance of LDSieve is the codesize. We conducted preliminary tests to find out what was the best codesize in practice, before comparing LDSieve to HashSieve. In some cases, *bad* codesizes rendered our variant impractical, so providing insight on the selection of good codesizes is essential to run LDSieve. We point out that we used *good* codesizes in LDSieve, but they demand preliminary tests in order to be found. As for HashSieve, we chose the so-called *optimal parameters* $\mathtt{K} = \lfloor 0.2209n \rceil$ and $\mathtt{T} = \lfloor 2^{0.1290n} \rceil$ (i.e., the leading theoretical terms rounded to the nearest integer).

Codesizes: To analyse LDSieve in high dimensions, we ran our implementation with 64 threads, with different codesizes in intervals of 10 units, for lattices in dimensions 81, 84, 87, 88, 90 and 92, thus extending the experiments in Section III-C3, and we picked that with the lowest execution time. We do not report on the variation in the execution time and vectors as functions of the codesize due to lack of space. Extending these tests for higher dimensions, to find the best codesizes for our implementation, is impractical, as we run our implementation five times for each combination of parameters (codesize and dimension), due to noise introduced by the machine, as said before. For instance, for the lattice in dimension 92, we tested our implementation with four codesizes (90, 100, 110 and 120). With an average of about 2 hours per run, the experiments for dimension 92 took about 40 hours. For dimension 96, these would take around 100 hours and for dimension 100 about 400 hours.

Execution time of LDSieve and HashSieve: Figure 4 shows the execution time of our parallel implementation of LDSieve, with M=3 for lattices in dimension 81, 87 and 90 (which are divisible by 3), and M=4 for lattices in dimension 80, 88 and 92 (which are divisible by 4), against HashSieve. The lattice in dimension 84, which is divisible both by 3 and 4, was solved with LDSieve set both with M=3 and M=4. We did not test higher dimensions as HashSieve requires more memory than that available in the system (e.g. in dimension 94, HashSieve requires about 800GBs of memory). Although probing can be used to reduce memory consumption, it increases the execution time of the algorithm (cf. Section 5 of [5]), which would render our comparison unfair, as LDSieve does not require or use probing.

Our experiments support the claim that LDSieve overcomes HashSieve for high dimensions. In particular, our LDSieve variant performs similarly to HashSieve for the lattice in dimension 81, but beats HashSieve for higher dimensions. This holds for any of the tested lattice dimensions when M=3. However, with M=4, interestingly, this is not true for all dimensions. For instance, LDSieve with M set to 4 is better than HashSieve for the lattice in dimension 88, but is slightly worse for the lattice in dimension 92. We believe that this happens because LDSieve is more efficient in practice with M=3, for the lattice dimensions that we tested. Another result suggesting this is that LDSieve is better than HashSieve for the lattice in dimension 87 when M=3, but not when M=4.

Changing M has implications both in the algorithm and the implementation. With regard to the impact in the algorithm, increasing M has both positive and negative consequences. As M is increased, the size CS of the subcode S decreases, and so the overhead of computing blockwise inner products decreases as fewer inner products have to be computed. This means that finding the right buckets becomes cheaper. On the other hand, increasing M makes the product code more structured and therefore less *random*. As a result, the theoretical analysis

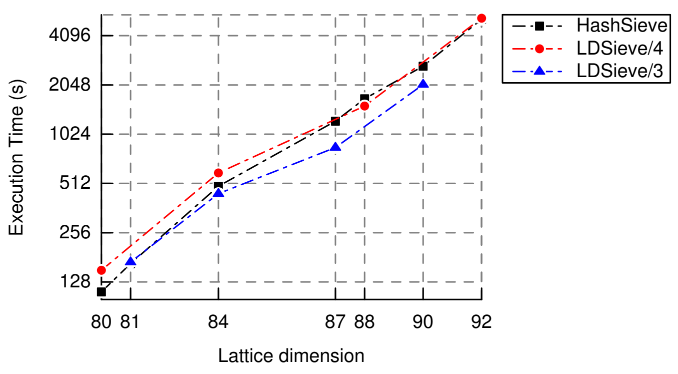


Fig. 4: Comparison of our parallel implementation of LDSieve against HashSieve, for lattices in dimensions 80-92, running with 64 threads. Lattice bases are BKZ-34 reduced. M = 4 for N = 80, 84, 88 and 92. M = 3 for N = 81, 84, 87 and 90.

	HashSieve		LDSieve			
N	**Mem.**	**Vectors**	**Mem.**	**Vectors**	**M**	**CS**
80	32	405	39	1114	4	70
81	35	451	40	752	3	300
84	48	615	45	1460	3	320
84	48	615	65	1697	4	90
87	120	1025	54	2472	3	360
88	135	1156	67	2947	4	90
90	310	1713	66	4422	3	400
92	379	2100	119	10122	4	110

TABLE II: Memory (in GBs) and number of vectors used for HashSieve and LDSieve for lattices in dimensions 80-92, running with 64 threads. LDSieve set with M=3 and M=4, multiple codesizes.

of [1] starts to be less accurate, and there will be more imbalances in the buckets: many buckets will be empty, and a few buckets will contain many vectors. This means the bucketing strategy becomes less effective, as ideally each bucket has roughly the same number of vectors. Theoretically, it is not quite clear which value M is optimal, and only further extensive experiments can answer this question.

As for the implications in the implementation, a few facts deserve attention. We store partial inner product lists `listEntry blockIpLists[M][CODESIZE]`, where `listEntry` is a struct containing a `long` value and an `unsigned int` index. This means that there are `M` rows of `CODESIZE` length. Even for high dimensions, we chose combinations of `M` and `CODESIZE` summing up to a maximum of 1200 elements (when `M` was set to 3 and `CODESIZE` was set to 400), which implies a size of 14KB and does not raise concerns regarding size or cache locality.

Codes are stored in the form of `NodeCS codes[CODESIZE]`, where `NodeCS` is a struct with an array of `N/M` elements. This data structure does not raise size or cache locality issues either. For instance, for a lattice in dimension 90, `M` set to 3 and `CODESIZE` set to 400, the array will require $400 \times 90/3 \times 2 \simeq 25$ KB, as `shorts` are two bytes. These data structures are not problematic concerning cache locality, for any dimension N where the SVP is tractable.

Number of vectors used and memory: Although LDSieve does not make use of large data structures, there are two other details pertaining to its memory consumption that are relevant.

First, the HashSieve implementation we used throughout the benchmarks does not replicate significant data structures among the running threads. Our LDSieve variant, on the other hand, does, as the data structures for the partial inner product lists are replicated among threads. As we showed before, these are small for tractable lattice dimensions, and thus this is not problematic. Nevertheless, even with a thread count as high as 64 threads, our implementation is considerably more energy efficient than HashSieve after dimension 84, both when `M` is set to 3 and 4, as shown in Table II. For instance, our LDSieve variant uses as much as 5 times less memory than HashSieve (e.g. in dimension 90). LDSieve has thus a significant edge over HashSieve as it mitigates the key problem of HashSieve, its memory consumption. It also suggests that if high dimensions are to be solved with sieving algorithms, LDSieve would be a natural choice due to a much smaller memory consumption rate, in comparison to HashSieve.

Second, although our LDSieve variant requires much less memory than HashSieve, it uses substantially higher numbers of vectors than HashSieve to reach convergence, as also shown in Table II. For instance, in dimension 92, our LDSieve implementation uses 5 times more vectors than HashSieve, but the execution time of both implementations is identical, as shown in Figure 4. It is important to point out that the number of vectors stored in memory is not the main contributor to the overall memory comsumption (of either implementation). In both implementations, most of the used memory is used to store the (various) hash table(s) and respective buckets.

Finally, good choices of the codesize and `M` are essential to achieve both low memory consumption levels and satisfactory numbers of used vectors (as we showed in Subsection III-C3). It is hard to infer from theory good values for these parameters, as they depend on many factors and implementation details.

V. Conclusions

Lattice-based cryptosystems can be broken when specific (hard) lattice problems can be solved in a timely manner. The SVP is a central problem in this context. There are many algorithms to address this problem, and sieving algorithms have been attracting increasing attention due to their room for improvement and unique ability to leverage specific types of lattices, such as ideal lattices. In particular, the last two years have witnessed considerable advances in this regard, with the introduction of HashSieve and, more recently, LDSieve. HashSieve represented a notable advance in sieving algorithms, as it lends itself very well to parallelism and is practical even in high dimensions. The recent introduction of LDSieve raised important questions such as whether it is suitable to parallel architectures and is superior to HashSieve in high lattice dimensions.

This paper addresses fundamental questions pertaining to the practicability of LDSieve. We present a very efficient parallel variant of the algorithm, which we used as a fundamental tool to address these questions, and which achieves speedup factors of 50x over the original implementation of LDSieve. The first and perhaps most relevant question concerning LDSieve, is whether it can beat HashSieve, the most practical sieving SVP-solver to date. In particular, it is relevant to assess this claim (1) with equally optimized sequential implementations of both algorithms, (2) verifying if LDSieve lends itself to parallelism and finally (3) whether a parallel implementation of LDSieve compares well to a parallel version of HashSieve, which is known to scale well on shared-memory systems.

To verify this, we conducted a thorough analysis of the behavior of our parallel implementation LDSieve, in comparison to HashSieve. We conclude that our LDSieve variant scales linearly on shared-memory systems (at least up to 16 threads) and is better than a state of the art HashSieve implementation, when M=3, for dimensions 81-90, being competitive when M=4. In addition, we conclude that there are both pros and cons of LDSieve over HashSieve. An advantage of LDSieve is that it spends considerably less memory for very high dimensions (>92), and if memory becomes a limitation in the system, LDSieve is preferable over HashSieve. On the other hand, a disadvantage of LDSieve is that multiple parameters have to be selected to get optimal performance. In particular, a wrong codesize selection may render the algorithm impractical. Unfortunately, the best codesize in practice can only be found through empirical benchmarks, which are time-consuming. In addition, if a new lattice dimension is to be solved, there is no point in running preliminary benchmarks to find out the best codesize, as one would already have the solution after running the preliminary benchmarks. In this paper, we provided insight, for the first time, on the selection of codesizes as a function of the lattice dimension.

In the future, we plan on porting our implementation of LDSieve to distributed-memory systems and adapt it to ideal lattices, to assess the hardness of the SVP on ideal lattices, which is of prime importance for lattice-based cryptography.

Acknowledgments. This work was partially supported by the DFG as part of project P1 within the CRC 1119 CROSSING. The project did also fund a visit from Thijs Laarhoven to Darmstadt, from Oct. 12th till Oct. 17th, 2015. Thijs Laarhoven is supported by the SNSF ERC Transfer Grant CRETP2-166734 FELICITY.

References

[1] A. Becker, L. Ducas, N. Gama, and T. Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *SODA*, 2016.

[2] A. Becker, N. Gama, and A. Joux. A sieve algorithm based on overlattices. In *ANTS*, pages 49–70, 2014.

[3] A. Becker and T. Laarhoven. Efficient (ideal) lattice sieving using cross-polytope LSH. *Cryptology ePrint Archive, Report 2015/823*, pages 1–25, 2015.

[4] P. Klein. Finding the closest lattice vector when it's unusually close. In *SODA*, pages 937–941, 2000.

[5] T. Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *CRYPTO*, pages 3–22, 2015.

[6] T. Laarhoven and B. de Weger. Faster sieving for shortest lattice vectors using spherical locality-sensitive hashing. In *LATINCRYPT*, pages 101–118, 2015.

[7] A. Mariano and C. Bischof. Enhancing the scalability and memory usage of HashSieve on multi-core CPUs. In *PDP16, to appear*, 2016.

[8] A. Mariano, T. Laarhoven, and C. Bischof. Parallel (probable) lock-free HashSieve: a practical sieving algorithm for the SVP. In *ICPP*, 2015.

[9] A. Mariano, Özgür Dagdelen, and C. Bischof. A comprehensive empirical comparison of parallel ListSieve and GaussSieve. In *APCI&E*, 2014.

[10] A. Mariano, S. Timnat, and C. Bischof. Lock-free GaussSieve for linear speedups in parallel high performance SVP calculation. SBAC-PAD'14, 2014.

[11] D. Micciancio and P. Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *SODA*, pages 1468–1480, 2010.