Parallel Improved Schnorr-Euchner Enumeration SE++ for the CVP and SVP

Fábio Correia*, Artur Mariano†, Alberto Proença‡, Christian Bischof† and Erik Agrell§

*†Technische Universität Darmstadt
‡ University of Minho
§Chalmers University of Technology
*fabio.lei.67@gmail.com

Abstract—The Closest Vector Problem (CVP) and the Shortest Vector Problem (SVP) are prime problems in lattice-based cryptanalysis, since they underpin the security of many lattice-based cryptosystems. Despite the importance of these problems, there are only a few CVP-solvers publicly available, and their scalability was never studied.

This paper presents a scalable implementation of an enumeration-based CVP-solver for multi-cores, which can be easily adapted to solve the SVP. In particular, it achieves super-linear speedups in some instances on up to 8 cores and almost linear speedups on 16 cores when solving the CVP on a 50-dimensional lattice. Our results show that enumeration-based CVP-solvers can be parallelized as effectively as enumeration-based solvers for the SVP, based on a comparison with a state of the art SVP-solver. In addition, we show that we can optimize the SVP variant of our solver in such a way that it becomes 35%-60% faster than the fastest enumeration-based SVP-solver to date.

I. INTRODUCTION

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

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

where n is the *rank* of the lattice. When $n = m$, the lattice is said to be of *full rank*. Lattices have a wide range of applications. These span from mathematics (e.g. geometry of numbers [9]) to computer science (e.g. integer programming [18] and lattice-based cryptography [16], [22]). The use of lattices in cryptography started in the beginning of the 80's, when the Lenstra–Lenstra–Lovász (LLL) algorithm [20] was used to break knapsack cryptosystems, and became prominent in cryptography in the mid-90's, when the first lattice-based encryption schemes were proposed (e.g. [3]).

Today, lattice-based cryptography is especially attractive because, among other reasons, it is believed to be resistant against attacks operated with quantum computers. Lattice-based cryptosystems can only be broken when specific lattice problems can be solved in a timely manner. In this context, two lattice problems are especially relevant: the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP). The SVP consists in finding the shortest nonzero vector of the lattice, whose norm is denoted by $\lambda_1(\mathcal{L})$, or, in other words, to find $\mathbf{u} \in \mathbb{Z}^n \backslash 0$ that minimizes the Euclidean norm $\|\mathbf{B} \cdot \mathbf{u}\|$. The CVP consists in finding the closest vector of the lattice to a given target vector $\mathbf{t} \in \mathbb{R}^m$, i.e. to find $\mathbf{u} \in \mathbb{Z}^n$ minimizing $\|\mathbf{B} \cdot \mathbf{u} - \mathbf{t}\|$. Algorithms that solve these problems are usually referred to as *SVP- and CVP-solvers*. There is a natural connection between the SVP and CVP: the closest vector to the origin, excluding the origin itself, is the vector with norm $\lambda_1(\mathcal{L})$. From a computational perspective, the decisional variant of the CVP is known to be NP-hard [8], whereas the decisional variant of SVP is known to be NP-hard under randomized reductions [2], [13].

Both CVP- and SVP-solvers work faster on reduced lattice bases, i.e., lattices whose bases have short, nearly orthogonal vectors. The main algorithms used in practice to reduce lattices are the Lenstra-Lenstra-Lovász (LLL) and the Block Korkine-Zolotarev (BKZ) algorithms. There is a close relation between lattice reduction algorithms and SVP-solvers. BKZ, for instance, uses SVP-solvers as part of its logic, as a way to improve the quality of their output.

The SVP has been extensively studied during the last three decades and two main families of SVP-solvers have emerged and evolved. The first is the family of sieving algorithms, i.e., probabilistic, randomized algorithms that repeatedly sieve a list of vectors, until a given stop criterion is met [4], [19]. Enumeration algorithms, on the other hand, enumerate all the possible vectors within a given search radius around the origin, and select the shortest among those [23], [17], [19].

The CVP has also been studied during the last decades, but to a lesser degree than the SVP [13]. In particular, the computational practicability of the CVP has received little attention. While several SVP-solvers have been implemented on various computer architectures [10], [15],

few open implementations of CVP-solvers are available. In particular, the scalability of CVP-solvers was never studied. One of the reasons why the CVP has attracted less attention than the SVP might be the lack of a public repository for the assessment of CVP-solvers, such as the SVP-challenge[1], which only covers the SVP.

The lack of study of the practicability of the CVP is a considerable gap in knowledge. The design and assessment of efficient implementations of CVP-solvers are of prime importance, because (1) they provide us with knowledge of the security of lattice-based cryptosystems, (2) they might be used as efficient building-blocks of SVP-solvers and (3) CVP is a problem of major relevance in other fields, such as in multiple-input multiple-output (MIMO) wireless networks, for coded and uncoded signals, where it is called *sphere decoding*. An example where the CVP is used as a building-block is the deterministic SVP-solver with the best known complexity, based on the Voronoi-cell of a lattice [1], which is based on executing a big number of CVP calls.

Our contribution: In this paper, we address two fundamental problems. On one hand, we study the practicability of the CVP, to which end we implement and assess the performance of an enhanced version of the Schnorr-Euchner enumeration routine, described in [12], a CVP-solver that can easily be modified to solve the SVP, from here on referred to as SE++. In particular, we propose a parallel version of this algorithm for shared-memory CPU systems, implemented with OpenMP, and we analyze its performance on a 16-core CPU system against the parallel SVP-solver proposed in [10].

On the other hand, we improve the SVP variant of the SE++ algorithm, discarding the computation of symmetrical branches of the enumeration tree, which generate vectors with identical norm and are, therefore, irrelevant in the context of the SVP. We refer to this implementation as "Improved SE++".

Results: Our results show that enumeration-based CVP-solvers, whose scalability was never studied, can be parallelized at least as efficiently as enumeration-based SVP-solvers, based on a comparison of the CVP and SVP versions of our algorithm and the state of the art SVP implementation described in [10]. In particular, our parallel version of this algorithm achieves super-linear speedups in some instances on up to 8 cores and a speedup factor of 14.8x for 16 cores when solving the CVP on a 50-dimensional lattice, on a dual-socket machine with 16 physical cores. These speedups are only possible due to the introduction of two parameters that improve load balancing between the threads, and minimization of synchronization, as explained in Section V. On the SVP variant of the SE++ algorithm, we improved the algorithm in such a way that it outperforms that of

[10] by a factor of 35%-60%, depending on the lattice dimension, thus becoming the fastest full enumeration-based SVP-solver to date.

Roadmap: The rest of this paper is organized as follows. Section II introduces the notation used and definitions. Section III overviews CVP/SVP-solvers and available implementations. Section IV overviews the SE++ algorithm in detail. Section V describes the optimization that avoids symmetric branches, our parallel implementation, and its mechanism that balances the workload among threads. Section VI shows the results of the performance and scalability of our implementation, for both the CVP and the SVP, as well as in comparison to the implementation of the SVP-solver described in [10]. Finally, Section VII concludes the paper.

II. NOTATION AND DEFINITIONS

The Euclidean norm of a vector $\mathbf{v} \in \mathbb{R}^n$, denoted by $\|\mathbf{v}\|$, is the distance spanned from the origin of the lattice to the point given by the vector $\mathbf{v}$, i.e. $\|\mathbf{v}\| = \sqrt{\sum_{i=1}^{n} \mathbf{v}_i^2}$, where $\mathbf{v}_i$ is the ith coordinate of $\mathbf{v}$. 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 $\mathbf{M}$, and their scalar elements are denoted by $\mathbf{v}_i$ and $\mathbf{M}_{i,j}$, respectively. The absolute value of a is given by $|a|$. The lattice $\mathcal{L}$ generated by a basis $\mathbf{B}$ is denoted $\mathcal{L}(\mathbf{B})$.

III. RELATED WORK

This section overviews the development of the enumeration algorithms for the SVP and CVP, in Section III-A, and the corresponding sequential and parallel implementations, in Section III-B. Lattice-reduction algorithms, and algorithms for the approximate SVP fall out of the scope of this paper, and are not, therefore, overviewed in this section. Algorithms for the approximate CVP are briefly recapped in Section III-A.

A. Algorithms

1) Exact CVP- and SVP-solvers: : P. van Emde Boas showed, in 1981, that the general closest vector problem as a function of the dimension n is NP-hard [8]. The breakthrough papers in the SVP and the CVP date back to 1981, when Pohst presented an approach that examines lattice vectors that lie inside a hypersphere [23], and to 1983, when Kannan showed a different approach using a rectangular parallelepiped [17]. Extensions of these two approaches were published later on, by Fincke and Pohst, in 1985 [11], and by Kannan (following Helfrich's work [14]), in 1987 [18]. In 1994, Schnorr and Euchner proposed a significant improvement of Pohst's method [25], that was later on found to be substantially faster than Pohst's and Kannan's approaches [1]. The improvement proposed by Schnorr and Euchner was influenced by the Nearest Plane algorithm by Babai, a polynomial-time method to find vectors that are close to a given
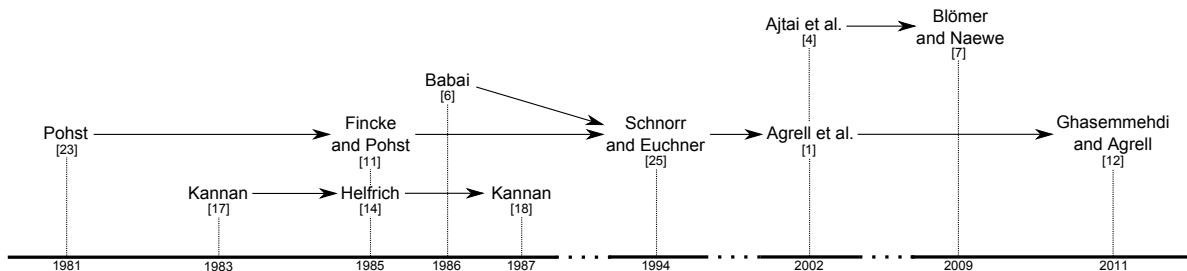
Fig. 1. Timeline of the most relevant publications regarding CVP-solvers, enumeration SVP-solvers, approximate CVP-solvers, and their connections.

target vector [6]. Recently, Ghasemmehdi and Agrell showed that there are some redundant operations in the algorithm, which can be eliminated, thereby accelerating it substantially [12].

2) Approximate CVP-solvers: : There are essentially two different approximate CVP-solvers: the *Nearest Plane algorithm*, developed by Babai in 1986 [6], and specific sieving algorithms. The first algorithm uses LLL to solve the approximate CVP in polynomial time, with an approximation ratio of $2(\frac{2}{\sqrt{3}})^n$, where n is the rank of the lattice. A distilled, yet precise, description of the algorithm can be found in [19].

The root of sieving algorithms dates back to 2001, when Ajtai et al. proposed a randomized algorithm that solves the exact version SVP, with very high probability [4]. This algorithm became known as AKS and it was later on extended to solve the approximate CVP [5]. It is still unclear (1) how practical this algorithm can be for the CVP and (2) if and how other sieving algorithms, such as GaussSieve [21], can be modified to solve the problem. Further improvements on the AKS were proposed by Blömer et al. [7].

Figure 1 shows a time-line and the connections between the most relevant of these publications.

B. Implementations

GPU and CPU parallel implementations of the Schnorr-Euchner enumeration, otherwise known as ENUM, were proposed in 2009 [15], and 2010 [10], respectively. The latter achieves almost linear speedups on a 16-core machine, for the SVP. In Section VI we show a comparison between our implementation and that described in [10].

The fplll library includes an implementation of the Kannan–Fincke–Pohst algorithm for the SVP [24]. It is still unclear what performance levels a modified version of this algorithm can attain on CVP, since it is neither included in the fplll library nor other available implementations are known. Another implementation of an enu-

meration process can be found in Magma[2]. However, it requires users to contribute to distribution costs (licenses start at 1000€).

The NTL library includes an implementation of the Nearest Plane algorithm for the approximate CVP (fplll includes a non-supported implementation). Comparisons with these implementations fall out of the scope of our paper, since we are only interested in performance comparisons for the SVP and CVP.

In summary, there are neither sequential nor parallel publicly available CVP-solvers, to the best of our knowledge. However, implementations of this kind are very relevant, because they permit to assess the security of lattice-based cryptosystems whose hardness is proportional to the hardness of the CVP.

IV. THE SE++ ALGORITHM

This section provides a brief description of the closest point search algorithm, dubbed SE++, proposed by Ghasemmehdi and Agrell [12]. This algorithm is an improved version of the algorithm described by Agrell et al. called SE [1], which is based on the Schnorr-Euchner variant [25] of the Fincke-Pohst method [11].

The SE++ algorithm can be separated in two different phases: the basis pre-processing and the sphere decoding. In the pre-processing phase, the matrix that contains the basis vectors, denoted by **B**, is reduced (e.g., with the BKZ or LLL algorithms). The resultant matrix **D**, is transformed into a lower-triangular matrix, which we refer to as **G**, with either the QR decomposition or the Cholesky decomposition (see [1] for further details). This transformation can be seen as a change of the coordinate system. The decomposition of **D** also generates an orthonormal matrix **Q**. The target vector **r**, i.e., the vector that we want to compute the closest vector to, is also transformed into the coordinate system of **G**, i.e., $\mathbf{r}' = \mathbf{r}\mathbf{Q}^T$. Finally, the sphere decoding is fed with the dimension of the lattice n, the transformed target vector

---

[2]http://magma.maths.usyd.edu.au/magma/

r' and the inverse of **G**, i.e., $\mathbf{H} = \mathbf{G}^{-1}$, which is itself a lower triangular matrix.

There are two different ways of thinking about the sphere decoding process. Mathematically, it is the process of enumerating lattice points inside a hypersphere (cf. [12] for a detailed mathematical description). Algorithmically, this can be described as a traversal of a tree, a useful view to understand the logic behind the proposed parallelization approach. In particular, it consists in a depth-first traversal on a weighted tree formed by all vectors of projections of $\mathcal{L}$ orthogonally to basis vectors. We will refer to the process of visiting a child node (decrementing i, where i denotes the depth of the node that is being analyzed at any given moment) as *moving down* and the process of visiting a parent node (incrementing i) as *moving up*.

The algorithm iterates over all the nodes in a zigzag pattern. It starts at the root and stops when it reaches the root again. The node at depth $(i-1)$ that is being visited is determined by $\mathbf{u}_i$. The siblings of this node are visited in a zigzag pattern, based on the Schnorr-Euchner refinement [25]. $\boldsymbol{\Delta}_i$ contains the step that has to be taken to visit the next node at depth $(i-1)$ and is used to update $\mathbf{u}_i$. The squared distance from the target vector **r** to the node that is being analyzed is denoted by λ_i, while C is the squared distance of **r** to the closest vector to **r** found so far. C is initialized to infinity. If $\lambda_i < C$, the algorithm will *move down*, otherwise it will *move up* again.

Whenever a leaf is reached, the values of vector **u** are saved in $\hat{\mathbf{u}}$, which represents the closest vector to **r** found so far, and C is updated, which reduces the number of nodes that still have to be visited. Although the algorithm behaves as a tree traversal, there is no physical tree (i.e. a data structure) implemented.

As proposed by Ghasemmehdi and Agrell [12], a vector **d** is used to store the starting points of the computation of the projections. The value $\mathbf{d}_i = k$ determines that, in order to compute $\mathbf{E}_{i,i}$ (see [12] for further details about matrix **E**), it is only necessary to calculate the values of $\mathbf{E}_{j,i}$ for $j = k-1, k-2, \ldots, i$, thereby avoiding redundant calculations.

V. IMPLEMENTATION

A. Avoiding symmetric branches

Both the SE algorithm and its enhanced version SE++, respectively presented in [1] and [12], compute the whole enumeration tree, thereby computing several vectors that are symmetric of one another. Since the purpose of the algorithm is to find the shortest vector **v** of norm $\|\mathbf{v}\|$, it is not relevant whether **v** or $-\mathbf{v}$ is found, since **v** and **-v** have exactly the same norm. Therefore, the computation of one of these vectors can be avoided, thus reducing the number of vectors that are ultimately computed. We have incorporated this optimization in SE++, an
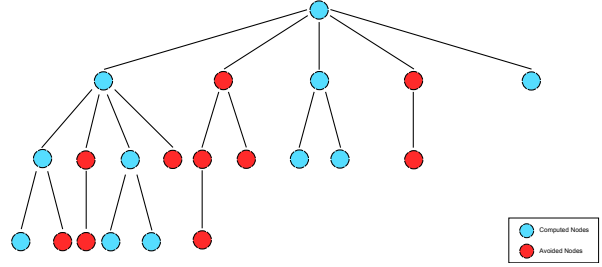

Fig. 2. Representation of the symmetric subtrees whose computation can be avoided.

implementation we refer to *Improved SE++*, from here on.

The ENUM algorithm avoids these computations already, by using a variable, called $last\_nonzero$, which stores the largest index i of the coefficient vector **u** for which $\mathbf{u}_i \neq 0$. For example, if $\mathbf{u}_i = 0$, but its parent $\mathbf{u}_{i+1} \neq 0$, then all its subtrees have to be computed. On the other hand, there are only symmetric subtrees on nodes where $\mathbf{u}_j = 0, j = i, ..., n$. As shown in Figure 2, there are only subtrees whose computation can be avoided on the leftmost nodes of each level. Since **u** defines the subtree of each level that will be computed next, it is updated differently for nodes that contain symmetric subtrees than for nodes that do not contain them. On trees that contain symmetric subtrees, the value of $\mathbf{u}_i$ is incremented, searching only in one direction. On the other hand, on trees that do not have symmetric subtrees $\mathbf{u}_i$ is updated in a zigzag pattern, searching in both directions (positive and negative values of $\mathbf{u}_i$).

Each time the algorithm moves up on the tree and $i \geq last\_nonzero$, the variable is updated, indicating the new lowest level that contains symmetric subtrees. At the beginning of the execution, $last\_nonzero$ is initialized to 1, the index of the leaves. Due to the similarities between both algorithms, we applied this strategy to SE++ in the same way.

B. Parallelization

As previously mentioned, the workflow of the algorithm can be naturally mapped onto a tree traversal, where different branches can be computed in parallel. Figure 3 shows a partition of these branches into several tasks that can be computed in parallel, by different threads. (Very fine grained) synchronization is only used to update the best vector found at any given instant (the closest to the target vector, at a given moment), as explained below. The proposed implementation was written in C, and creates these tasks with OpenMP. Once tasks are created, they are added to a queue of tasks, and scheduled by the OpenMP run-time system among the running threads. This system also defines the order of execution of the created tasks, in run-time.
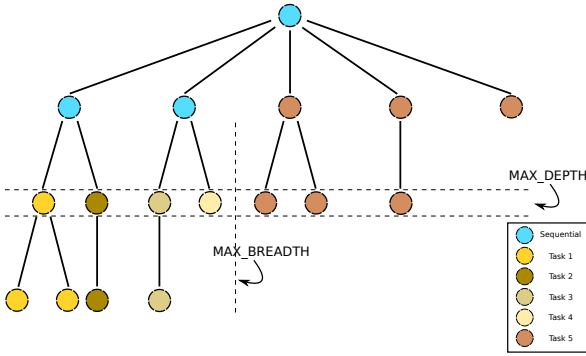
Fig. 3. Map of the algorithm workflow on a tree, partitioned into tasks, according to the parameters MAX_BREADTH and MAX_DEPTH.

Our implementation combines a depth-first traversal with a breadth-first traversal. The work is distributed among threads in a breadth-first manner (across one or more levels), while each thread computes the work that it was assigned in a depth-first manner. First, a team of threads, whose size is set by the user, is created. Then, a number of tree nodes, based on two parameters, MAX_BREADTH and MAX_DEPTH, are computed sequentially. These two parameters also define the number and size of the tasks that are created. Once the MAX_DEPTH level is reached, a task for each of the nodes in that level is created, as also shown in Figure 3. However, when creating the task number MAX_BREADTH, i.e. $|\mathbf{\Delta}_i|$ = MAX_BREADTH, the task entails not only the current node but also all the siblings of that node (excluding those within other tasks) and their child nodes, as shown in Task 5, in Figure 3. Once tasks are created, they are (either promptly or after some time) assigned to one of the threads within the team, by the OpenMP run-time system. As there is an implicit barrier at the end of the single region, which means that all the created tasks will be, at that point, already processed.

There is a number of problems that must be addressed in such an implementation. In first place, the tree is considerably unbalanced. $|\mathbf{\Delta}_i|$ can be used to estimate the size of the subtree of the node that is being analyzed, because it can be seen as a relative distance between a node at depth $(i-1)$ that is being analyzed and the first vector of the same subtree that was analyzed. Therefore, $|\mathbf{\Delta}_i|$ is used to identify heavier and lighter subtrees: the lower $|\mathbf{\Delta}_i|$ is, the heavier the tree. As mentioned, MAX_BREADTH determines the value of $|\mathbf{\Delta}_i|$ from which subtrees are grouped together, thus preventing the creation of too fine-grained tasks. We set MAX_BREADTH = 6, based on empirical tests presented in Section VI.

Some of the heavier subtrees need to be split in order to attain better load balancing. The maximum depth is chosen based on the number of threads on the system. In particular, the more threads, the more split the tree is. Therefore, we define MAX_DEPTH $= n - \log_2(\#Threads)$, which determines the lowest depth that is reached to split subtrees. Like MAX_BREADTH, the value for this parameter was also chosen based on empirical tests. Together, these 2 parameters represent the trade-off between the granularity and the number of created tasks.

When a thread processes a task, it computes all the nodes on the branch spanned from the root of the enumeration tree up to the root of the subtree in the task, then computing the subtree entailed by the task. The level of the subtree that was assigned, given by i_lvl, and the nodes that have to be recomputed, given by the vector **u_Aux**, are passed as arguments. Additionally, the value of $|\mathbf{\Delta}_i|$ is also sent to the task, allowing to distinguish subtrees that were grouped together from single subtrees.

The recomputation of nodes that belong to previous levels of the tree allows to lower the number of memory allocations and boost performance. Instead of allocating each vector and matrix for each task, it is only necessary to allocate a much smaller vector **u_Aux** that contains the coefficients of the nodes that have to be recomputed. Therefore, each thread concurrently allocates its own (private) block of memory (a struct) for matrix **E** and vectors **u**, **y**, λ, $\mathbf{\Delta}$ and **d** (for more about these these structures see [12]) and re-uses the same memory for the execution of all the tasks that are assigned to it. Empirical tests showed that performance can be improved by a factor of as much as 20% with this optimization.

The value of C is stored in a global variable, accessible by every thread. Threads check the value of C, which dictates the rest of the nodes that are visited by each thread. C is initialized with $1/\mathbf{H}_{1,1}$, instead of infinity, to prevent the creation of unnecessary tasks. For the same reason, $\hat{\mathbf{u}}$ is initialized with $\hat{\mathbf{u}} = (1, 0, \ldots, 0)$. Although these variables are shared among all the threads, only one thread updates them at a time. An OpenMP critical zone is used to manage this synchronization. Every time a thread executes the critical zone, it checks $\lambda_1 < C$ again, since other threads might update those values in the meantime.

VI. RESULTS

The tests were performed on a dual-socket machine with 2 Sandy Bridge Intel Xeon E5-2670 processors, each with 8 cores, and simultaneous multi-threading (SMT) technology. The machine has a total of 128 GB of RAM. The codes were written in C and compiled with the GNU g++ 4.6.1 compiler, with the -O2 optimization flag (-O3 was slightly slower than -O2). Additionally, the
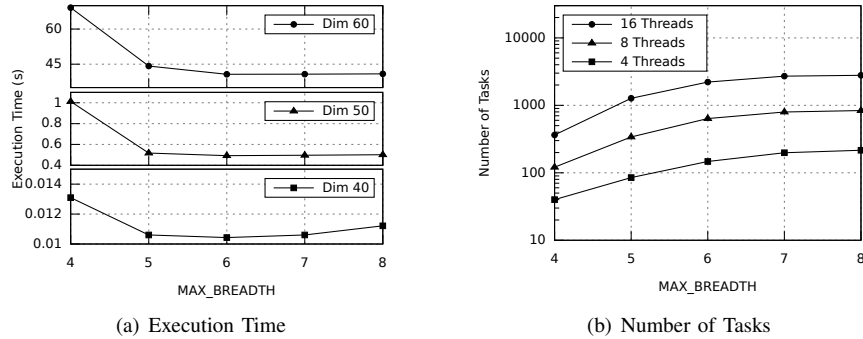
Fig. 4. Execution time of our implementation with 16 threads solving the CVP on random lattices in dimensions 40, 50 and 60, in (a), and number of tasks created for 4, 8, and 16 threads for a lattice in dimension 50, in (b). BKZ-reduced bases with block-size 20.

NTL[3] (for LLL and BKZ basis reduction) and Eigen[4] (for the QR decomposition, inverse and transpose matrix computations) libraries were also used. Although the code was written in C, the g++ compiler was required for these libraries. We have used Goldstein-Mayer bases for random lattices, available from the SVP Challenge[5], all of which were generated with seed 0. Although the execution times of the programs were fairly stable, each program was executed three times and the best sample was selected. The basis pre-processing, as described in Section IV, was not included in the time measurements.

As aforementioned, two parameters are used to prevent the creation of too many fine-grained tasks and to break down the biggest tasks into smaller tasks. The value for each of these parameters was set based on empirical tests. Several tests were performed in order to find the optimal value of MAX_BREADTH for different lattices and number of threads, for both solvers. For simplicity, Figures 4(a) and 4(b) show the results of only some of them. Different values for MAX_BREADTH were tested for both solvers in order to find its optimal value. Figure 4(a) shows the execution time for different values of MAX_BREADTH for BKZ-reduced lattices (with block-size 20) when running with 16 threads (for other number of threads the results were very similar), when solving the CVP. Figure 4(b) shows the number of tasks that are created in our parallel implementation, for 4, 8 and 16 threads, when solving the CVP. For the SVP and the Improved SE++, the number of tasks as a function of the MAX_BREADTH is similar. The higher the value of MAX_BREADTH the higher the number of tasks that are created. We set MAX_BREADTH = 6, since it was the result that revealed to be slightly better than the others, despite of creating many more tasks than MAX_BREADTH = 5. Since the difference between the number of created tasks is much higher than the difference between the

execution time, it is possible to conclude that OpenMP's implementation of the task queue is very optimized. To choose the best values for MAX_DEPTH, the level at which tasks are created was set manually. For each level, the execution time of the tasks was registered and compared to the total execution time. To ensure linear and super-linear speedups, the execution time of the heaviest task has to be lower than $\frac{1}{\#Threads}$. To avoid creating more tasks than it is necessary to guarantee a good load balancing, MAX_DEPTH is set dynamically as $n - \log_2(\#Threads)$. With the parameters set with these values, an ideal trade-off between load balancing and granularity of the tasks is guaranteed.

We tested the SE++ and the *Improved SE++* with LLL- and BKZ-reduced bases (BKZ ran with block-size 20). For LLL-reduced bases, they were tested with lattices in dimensions 40, 45 and 50. For BKZ-reduced bases, they were tested with lattices in dimensions 40, 50 and 60, since they run much faster in BKZ-reduced bases. Figure 5(a) shows the execution time of SE++, for the CVP, running with 1-32 threads[6], with LLL-reduced bases, and Figure 5(b) shows the same tests for BKZ-reduced bases. Figures 6(a) and 6(b) show the execution time, on the same conditions, for the SVP, of SE++ and the *Improved SE++* (which includes the optimization of avoid symmetric branches), on LLL- and BKZ-reduced bases, respectively.

There is a number of conclusions to be drawn. In first place, SE++ scales linearly for up to 8 threads and almost linearly for 16 threads, for both the CVP and SVP. The implementation can also benefit from the SMT technology, since the dependencies between the instructions prevent the full use of the functional units within each core. In second place, BKZ-reduced bases are much faster to compute, both for the CVP and SVP, than LLL-reduced bases. Last but not least, our implementation solves the CVP much faster than the SVP,
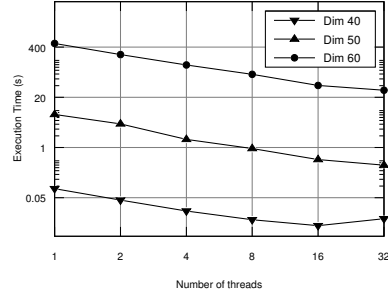
---

[3]http://www.shoup.net/ntl/
[4]http://eigen.tuxfamily.org/
[5]http://www.latticechallenge.org/svp-challenge/

[6]We used the parallel version running with a single thread as a single-core baseline, which is 5% slower than the pure-sequential version.
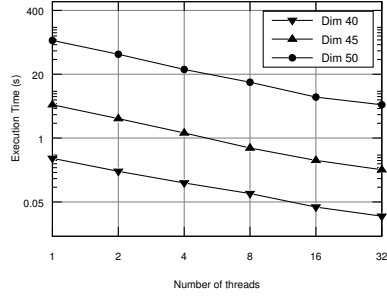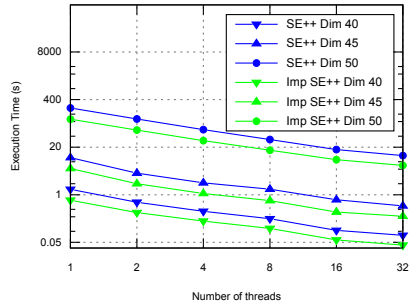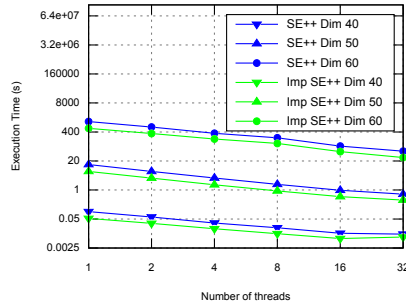
(a) LLL-reduced bases

(b) BKZ-reduced bases

Fig. 5. Execution time of SE++ solving the CVP on random lattices in dimensions 40, 45 and 50 for LLL-reduced bases, in (a), and 40, 50 and 60, for BKZ-reduced bases, in (b), for 1-32 threads.

(a) LLL-reduced bases

(b) BKZ-reduced bases

Fig. 6. Execution time of our implementation solving the SVP on random lattices in dimensions 40, 45 and 50 for LLL-reduced bases, in (a), and 40, 50 and 60, for BKZ-reduced bases, in (b), for 1-32 threads.

but the results are dependent on the target vector and on the tested lattice. In our experiments, we used a vector $\mathbf{t} = \mathbf{sB}$, where $\mathbf{s}_i = 0$ for $i = 1, \ldots, n/2$ and 0.75 for $i = n/2+1, \ldots, n$, and $\mathbf{B}$ is the basis of the lattice. This vector was chosen due to not being too close to the basis vectors, but also not too far away.

A few points need to be addressed regarding the scalability of our implementation. In the first place, and as in [10], the implementation might possibly have a smaller workload than the sequential execution would have. This might occur because some threads might find, at a given point, a vector that is strictly shorter than the distance from the input vector $\mathbf{r}'$ to the $(i-1)$-dimensional layers that would be analyzed in a sequential execution. This justifies the super-linear speedups that are achieved for some cases, such as on the CVP, with a BKZ-reduced 50-dimensional lattice, using four threads.

For the remaining cases, efficiency levels of $>90\%$ are attained for the majority of the instances of up to 8 threads, except for lattices in dimension 40, where the workload is too small to compensate for the creation and management of more than 4 threads. With 16 threads, the scalability is lower than for up to 8 threads, presumably because of the use of two CPU sockets, which is naturally slower than the use of a single socket, due to the Non-Uniform Memory Access (NUMA) organization of the RAM. In addition, Figures 6(a) and 6(b) show that the *Improved SE++* outperforms SE++ for the SVP by a factor of $\approx 50\%$ with similar scalability.

Figure 7 shows a comparison between the *Improved SE++* and the implementation in [10], for the SVP on two random lattices. It is possible to see that, in the general case, our implementation scales better. For the lattice in dimension 45 our approach has higher workload savings than the implementation in [10]. Both implementations are influenced by the NUMA organization of the RAM, as we can see in the case of the lattice in dimension 50. In general, with 1 thread, SE++ seems to be slower than [10], by a factor of 10% to 25%, even though it was 25% faster for the lattice in dimension 40. However, the *Improved SE++* outperforms [10] by a factor of 35% to 60%, thus becoming the fastest deterministic enumeration-based solver to date.

VII. Conclusions

We developed, implemented and assessed a multi-core CPU parallel implementation of the CVP-solver described in [12], a solver that, with slight modifications, can also be used to solve the SVP. We showed that the solver can be efficiently parallelized, achieving linear
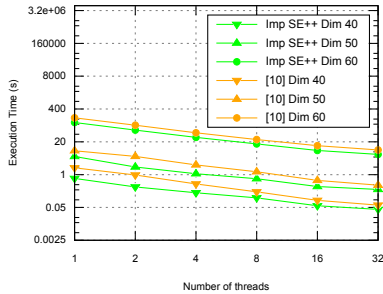
Fig. 7. Execution time of our implementations and the implementation in [10], for the SVP on LLL-reduced lattices..

speedups for up to 8 threads and almost linear speedups for 16 threads, both when running the CVP and the SVP. The use of 16 threads implies the use of two CPU sockets, which can explain the loss of linear scalability, due to the use of point-to-point processor interconnect buses, since the synchronization cost of threads on different sockets is higher. The implementation achieves super-linear speedups in some instances on up to 8 cores, due to workload savings with the parallel algorithm, as explained in Section VI. Some speedup factors of >14x are achieved for 16 cores, with efficiency levels of up to 93%. A crucial part of our parallelization scheme is the thorough load balancing via two added parameters, MAX_BREADTH and MAX_DEPTH, that (1) prevent the creation of too many fine-grained tasks and (2) break down the biggest tasks into smaller tasks.

Our results show that enumeration-based CVP-solvers can be parallelized as efficiently as enumeration-based SVP-solvers, since we compared the scalability both with the SVP version of our algorithm and a third party implementation, described in [10]. These results expand the knowledge that the community has on the practicability and scalability of enumeration-based CVP-solvers, which are not as studied as SVP-solvers. Moreover, it can be used as an efficient, parallel building block of algorithms such as the Voronoi-based SVP solver described in [1], which lacks of practical assessment to this day.

In addition, we showed that by avoiding the computation of symmetric branches, SE++ outperforms ENUM, the fastest deterministic enumeration-based SVP-solver known to this day. In particular, the Improved SE++ is faster than [10], by a factor of between 35% and 60%, depending on the lattice dimension, running with one thread. We also showed that the scalability of our implementation compares well with the implementation described in [10].

ACKNOWLEDGEMENTS

REFERENCES

[1] Erik Agrell et al.. Closest Point Search in Lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.
[2] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 99–108, New York, NY, USA, 1996. ACM.
[3] Miklós Ajtai and Cynthia Dwork. A Public-Key Cryptosystem with Worst-Case/Average-Case Equivalence. *Electronic Colloquium on Computational Complexity (ECCC)*, 3(65), 1996.
[4] Miklós Ajtai et al.. A Sieve Algorithm for the Shortest Lattice Vector Problem. In *STOC*, pages 601–610, 2001.
[5] Miklós Ajtai et al.. Sampling Short Lattice Vectors and the Closest Lattice Vector Problem. In *IEEE Conference on Computational Complexity*, pages 53–57, 2002.
[6] László Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.
[7] Johannes Blömer and Stefanie Naewe. Sampling Methods for Shortest Vectors, Closest Vectors and Successive Minima. *Theor. Comput. Sci.*, 410(18):1648–1665, April 2009.
[8] Peter van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in a lattice. Technical Report 81-04, Mathematische Instituut, University of Amsterdam, 1981.
[9] John W. Cassels. *An Introduction to the Geometry of Numbers (Reprint)*. Classics in mathematics. Springer, 1997.
[10] Özgür Dagdelen and Michael Schneider. Parallel Enumeration of Shortest Lattice Vectors. In *Euro-Par (2)*, pages 211–222, 2010.
[11] U. Fincke and M. Pohst. Improved Methods for Calculating Vectors of Short Length in a Lattice, Including a Complexity Analysis. *Mathematics of Computation*, 44:463–463, 1985.
[12] Arash Ghasemmehdi and Erik Agrell. Faster Recursions in Sphere Decoding. *IEEE Transactions on Information Theory*, 57(6):3530–3536, 2011.
[13] Guillaume Hanrot et al.. Algorithms for the Shortest and Closest Lattice Vector Problems. In *IWCC*, pages 159–190, 2011.
[14] Bettina Helfrich. Algorithms to Construct Minkowski Reduced an Hermite Reduced Lattice Bases. *Theor. Comput. Sci.*, 41:125–139, 1985.
[15] Jens Hermans, Michael Schneider, Johannes Buchmann, Frederik Vercauteren, and Bart Preneel. Parallel Shortest Lattice Vector Enumeration on Graphics Cards. In *AFRICACRYPT*, pages 52–68, 2010.
[16] Antoine Joux and Jacques Stern. Lattice Reduction: A Toolbox for the Cryptanalyst. *J. Cryptology*, 11(3):161–185, 1998.
[17] Ravi Kannan. Improved Algorithms for Integer Programming and Related Lattice Problems. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 193–206, New York, NY, USA, 1983. ACM.
[18] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Math. Oper. Res.*, 12(3):415–440, August 1987.
[19] Thijs Laarhoven et al.. Solving hard lattice problems and the security of lattice-based cryptosystems. Cryptology ePrint Archive, Report 2012/533, 2012.
[20] A.K. Lenstra et al.. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.
[21] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. *Electronic Colloquium on Computational Complexity (ECCC)*, 16:65, 2009.
[22] A. M. Odlyzko. The Rise and Fall of Knapsack Cryptosystems. In *In Cryptology and Computational Number Theory*, pages 75–88. A.M.S, 1990.
[23] Michael Pohst. On the Computation of Lattice Vectors of Minimal Length, Successive Minima and Reduced Bases with Applications. *SIGSAM Bull.*, 15(1):37–44, February 1981.
[24] Xavier Pujol and Damien Stehlé. Rigorous and efficient short lattice vectors enumeration. In *ASIACRYPT*, pages 390–405, 2008.
[25] Claus-Peter Schnorr and M. Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. *Math. Program.*, 66:181–199, 1994.