Parallel Enumeration of Shortest Lattice Vectors

Özgür Dagdelen[1] and Michael Schneider[2]

[1] Center for Advanced Security Research Darmstadt - CASED
oezguer.dagdelen@cased.de

[2] Technische Universität Darmstadt, Department of Computer Science
mischnei@cdc.informatik.tu-darmstadt.de

Abstract. Lattice basis reduction is the problem of finding short vectors in lattices. The security of lattice based cryptosystems is based on the hardness of lattice reduction. Furthermore, lattice reduction is used to attack well-known cryptosystems like RSA. One of the algorithms used in lattice reduction is the enumeration algorithm (ENUM), that provably finds a shortest vector of a lattice. We present a parallel version of the lattice enumeration algorithm. Using multi-core CPU systems with up to 16 cores, our implementation gains a speed-up of up to factor 14. Compared to the currently best public implementation, our parallel algorithm saves more than 90% of runtime.

Keywords: lattice reduction, shortest vector problem, cryptography, parallelization, enumeration

1 Introduction

A lattice $\mathcal{L}$ is a discrete subgroup of the space $\mathbb{R}^d$. Lattices are represented by linearly independent basis vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^d$, where n is called the dimension of the lattice. Lattices have been known in number theory since the eighteenth century. They already appear when Lagrange, Gauss, and Hermite study quadratic forms. Nowadays, lattices and hard problems in lattices are widely used in cryptography as the basis of promising cryptosystems.

One of the main problems in lattices is the *shortest vector problem* (SVP), that searches for a vector of shortest length in the lattice. The shortest vector problem is known to be NP-hard under randomized reductions. It is also considered to be intractable even in the presence of quantum computers. Therefore, many lattice based cryptographic primitives, e.g. one-way functions, hash functions, encryption, and digital signatures, leverage the complexity of the SVP problem. In the field of cryptanalysis, lattice reduction is used to attack the NTRU and GGH cryptosystems. Further, there are attacks on RSA and low density knapsack cryptosystems.

Lattice reduction still has applications in other fields of mathematics and number theory. It is used for factoring composite numbers and computing discrete logarithms using diophantine approximations. In the field of discrete optimization, lattice reduction is used to solve linear integer programs.

The fastest algorithm known to solve SVP is the enumeration algorithm of Kannan [8] and the algorithm of Fincke and Pohst [3]. The variant used mostly in practice was presented by Schnorr and Euchner in 1991 [14]. Nevertheless, these algorithms solve SVP in exponential runtime. So far, enumeration is only applicable in low lattice dimensions ($n \leq 60$).[3] For higher dimensions it is only possible to find *short* vectors, but not a *shortest* vector. Mostly, these approximate solutions of the SVP are sufficient in practice. In 1982 the famous LLL algorithm was presented for factoring polynomials [9]. This algorithm does not solve SVP rather it finds a vector with length exponential in the lattice dimension. However, LLL is the first algorithm having a polynomial asymptotic running time. LLL can be run in lattice dimension up to 1000.

In practice, the most promising algorithm for lattice reduction in high dimensions is the BKZ block algorithm by Schnorr and Euchner [14]. It mainly consists of two parts, namely enumeration in blocks of small dimension and LLL in high dimension. BKZ finds shorter vectors than LLL, at the expense of a higher runtime.

Considering parallelization, there are various works dealing with LLL, e.g., [15,2]. The more time-consuming part of BKZ, namely the enumeration step (assuming the use of high block sizes) was considered in the master's thesis of Pujol [11] and in a very recent work [13]. A GPU version of enumeration was shown in [7].

The enumeration in lattices can be visualized as a depth first search in a weighted search tree, with different subtrees being independent from each other. Therefore, it is possible to enumerate different subtrees in parallel threads without any communication between threads. We have chosen multi-core CPUs for the implementation of our parallel enumeration algorithm.

Our Contribution. In this paper, we parallelize the enumeration (ENUM) algorithm by Schnorr and Euchner [14]. We implement the parallel version of ENUM and test it on multi-core CPUs. More precisely, we use up to 16 CPU cores to speed up the lattice enumeration, in lattice dimensions of 40 and above. Considering the search tree, the main problem is to predict the subtrees that are examined during enumeration beforehand.

We gain speed-ups of up to factor 14 in comparison to our single core version.[4] Compared to the fastest single-core ENUM implementation known our parallel version of ENUM saves more than 90% of runtime. We add some clever additional communication among threads, such that by using s processor cores we even gain a speed-up of more than s in some cases.

By this work, we show that it is possible to parallelize the entire BKZ algorithm for the search for short lattice vectors. The strength of BKZ is used to assess the practical hardness of lattice reduction, which helps finding suitable parameters for secure lattice based cryptosystems.

[3] The recent work of [5] could no more be considered for our final version.
[4] On a 24 core machine we gain speed-up factors of 22.

The algorithm of Pujol [11,13] uses a volume heuristic to predict the number of enumeration steps that will be performed in a subtree. This estimate is used to predict if a subtree is split recursively for enumeration in parallel. In contrast to that, our strategy is to control the height of subtrees that can be split recursively.

Organization. Section 2 explains the required basic facts on lattices and parallelization, Section 3 describes the ENUM algorithm by [14], Section 4 presents our new algorithm for parallel enumeration, and Section 5 shows our experimental results.

2 Preliminaries

Notation. Vectors and matrices are written in bold face, e.g. $\mathbf{v}$ and $\mathbf{M}$. The expression $\lceil x \rfloor$ denotes the nearest integer to $x \in \mathbb{R}$, i.e., $\lceil x \rfloor = \lceil x - 0.5 \rceil$. $\|\mathbf{v}\|$ denotes the Euclidean norm, other norms are indexed with a subscript, like $\|\mathbf{v}\|_\infty$. Throughout the paper, n denotes the lattice dimension.

Lattices. A lattice is a discrete additive subgroup of $\mathbb{R}^d$. It can be represented as the linear integer span of $n \leq d$ linear independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^d$, which are arranged in a column matrix $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{d \times n}$. The lattice $\mathcal{L}(\mathbf{B})$ is the set of all linear integer combinations of the basis vectors $\mathbf{b}_i$, namely $\mathcal{L}(\mathbf{B}) = \{\sum_{i=1}^{n} x_i \mathbf{b}_i : x_i \in \mathbb{Z}\}$. The dimension of the lattice equals the number of linearly independent basis vectors n. If $n = d$, the lattice is called full-dimensional. For $n \geq 2$ there are infinitely many bases of a lattice. One basis can be transformed into another using a unimodular transformation matrix. The first successive minimum $\lambda_1(\mathcal{L}(\mathbf{B}))$ is the length of a shortest non-zero vector of a lattice. There exist multiple shortest vectors of a lattice, a shortest vector is not unique. Define the Gram-Schmidt-orthogonalization $\mathbf{B}^* = [\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*]$ of $\mathbf{B}$. It is computed via $\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^*$ for $i = 1, \ldots, n$, where $\mu_{i,j} = \mathbf{b}_i^T \mathbf{b}_j^* / \left\|\mathbf{b}_j^*\right\|^2$ for all $1 \leq j \leq i \leq n$. We have $\mathbf{B} = \mathbf{B}^* \mu^T$, where $\mathbf{B}^*$ is orthogonal and μ^T is an upper triangular matrix. Note that $\mathbf{B}^*$ is not necessarily a lattice basis.

Lattice Problems and Algorithms. The most famous problem in lattices is the shortest vector problem (SVP). The SVP asks to find a shortest vector in the lattice, namely a vector $\mathbf{v} \in \mathcal{L} \setminus \{0\}$ with $\|\mathbf{v}\| = \lambda_1(\mathcal{L})$. An approximation version of the SVP was solved by Lenstra, Lenstra, and Lovász [9]. The LLL algorithm is still the basis of most algorithms used for basis reduction today. It runs in polynomial time in the lattice dimension and outputs a so-called *LLL-reduced* basis . This basis consists of nearly orthogonal vectors, and a short, first basis vector with approximation factor exponential in the lattice dimension. The BKZ algorithm by Schnorr and Euchner [14] reaches better approximation factors, and is the algorithm used mostly in practice today. As a subroutine, BKZ makes use of an exact SVP solver, such as ENUM. In practice, SVP can only be solved in low dimension n, say up to 60, using exhaustive search techniques or, a second

approach, using sieving algorithms that work probabilistically. An overview of enumeration algorithms is presented in [12]. A randomized sieving approach for solving exact SVP was presented in [1] and an improved variant in [10]. In this paper, we only deal with enumeration algorithms.

Exhaustive Search. In [12] Pujol and Stehlé examine the floating point behaviour of the ENUM algorithm. They state that double precision is suitable for lattice dimensions up to 90. It is common practice to pre-reduce lattices before starting enumeration, as this reduces the radius of the search space. In BKZ, the basis is always reduced with the LLL algorithm when starting enumeration.

Publicly available implementations of enumeration algorithms are the established implementation of Shoup's NTL library and the fpLLL library of Stehlé et al. Experimental data on enumeration algorithms using NTL can be found in [4,10], both using NTL's enumeration. A parallel implementation of ENUM is available at Xavier Pujol's website.[5] To our knowledge there are no results published using this implementation. Pujol mentions a speedup factor of 9.7 using 10 CPUs. Our work was developed independently of Pujol's achievements.

Parallelization. Before we present our parallel enumeration algorithm, we need to introduce definitions specifying the quality of the realized parallelization. Furthermore, we give a brief overview of parallel computing paradigms.

There exist many parallel environments to perform operations concurrently. Basically, on today's machines, one distinguishes between shared memory and distributed memory passing. A *multi-core microprocessor* follows the shared memory paradigm in which each processor core accesses the same memory space. Nowadays, such computer systems are commonly available. They possess several cores, while each core acts as an independent processor unit. The operating system is responsible to deliver operations to the cores.

In the parallelization context there exist notions that measure the achieved quality of a parallel algorithm compared to the sequential version. In the sequel of this paper, we will need the following definitions:

Speed-up factor: time needed for serial computation divided by the time required for the parallel algorithm. Using s processes, a speed-up factor of up to s is expected.

Efficiency: speed-up factor divided by the number of used processors. An efficiency of 1.0 means that s processors lead to a speed-up factor of s which can be seen as a "perfect" parallelization. Normally the efficiency is smaller than 1.0 because of the communication overhead for inter-process communication.

Parallel algorithms such as graph search algorithms may benefit from communication, in such a way that fewer operations need to be computed. As soon as the number of saved operations exceeds the communication overhead, an efficiency of more than 1.0 might be achieved. For instance, branch-and-bound algorithms

[5] `http://perso.ens-lyon.fr/xavier.pujol/index_en.html`

for Integer Linear Programming might have superlinear speedup, due to the interdependency between the search order and the condition which enables the algorithm to disregard a subtree. The enumeration algorithm falls into this category as well.

3 Enumeration of the shortest lattice vector

In this chapter we give an overview of the ENUM algorithm first presented in [14]. In the first place, the algorithm was proposed as a subroutine in the BKZ algorithm, but ENUM can be used as a stand-alone instance to solve the exact SVP. An example instance of ENUM in dimension 3 is shown by the solid line of Figure 1. An algorithm listing is shown as Algorithm 1.

Algorithm 1: Basic Enumeration Algorithm

Input: Gram-Schmidt coefficients $(\mu_{i,j})_{1\leq j\leq i\leq n}$, $\|\mathbf{b}_1^*\|^2 \dots \|\mathbf{b}_n^*\|^2$
Output: $\mathbf{u}_{min}$ such that $\left\|\sum_{i=1}^{n} u_i \mathbf{b}_i\right\| = \lambda_1(\mathcal{L}(\mathbf{B}))$

$A \leftarrow \|\mathbf{b}_1^*\|^2$, $\mathbf{u}_{min} \leftarrow (1, 0, \dots, 0)$, $\mathbf{u} \leftarrow (1, 0, \dots, 0)$, $\mathbf{l} \leftarrow (0, \dots, 0)$, $\mathbf{c} \leftarrow (0, \dots, 0)$
$t = 1$
while $t \leq n$ **do**
  $l_t \leftarrow l_{t+1} + (u_t + c_t)^2 \|\mathbf{b}_t^*\|^2$
  if $l_t < A$ **then**
    if $t > 1$ **then**
      $t \leftarrow t - 1$ ▷ *move one layer down in the tree*
      $c_t \leftarrow \sum_{i=t+1}^{n} u_i \mu_{i,t}, \quad u_t \leftarrow \lceil c_t \rfloor$
    else
      $A \leftarrow l_t, \quad \mathbf{u}_{min} \leftarrow \mathbf{u}$ ▷ *set new minimum*
    end
  else
    $t \leftarrow t + 1$ ▷ *move one layer up in the tree*
    choose next value for u_t using the zig-zag pattern
  end
end

To find a shortest non-zero vector of a lattice $\mathcal{L}(\mathbf{B})$ with $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n]$, ENUM takes as input the Gram-Schmidt coefficients $(\mu_{i,j})_{1\leq j\leq i\leq n}$, the quadratic norm of the Gram-Schmidt orthogonalization $\|\mathbf{b}_1^*\|^2, \dots, \|\mathbf{b}_n^*\|^2$ of $\mathbf{B}$, and an initial bound A. The search space is the set of all coefficient vectors $\mathbf{u} \in \mathbb{Z}^n$ that satisfy $\|\sum_{t=1}^{n} u_t \mathbf{b}_t\| \leq A$. Starting with an LLL-reduced basis, it is common to set $A = \|\mathbf{b}_1^*\|^2$ in the beginning. If the norm of the shortest vector is known beforehand, it is possible to start with a lower A, which limits the search space and reduces the runtime of the algorithm. If a vector $\mathbf{v}$ of length smaller than A is found, A can be reduced to the norm of $\mathbf{v}$, that means A always denotes the size of the current shortest vector.

The goal of ENUM is to find a coefficient vector $\mathbf{u} \in \mathbb{Z}^n$ satisfying the equation

$$\left\|\sum_{t=1}^{n} u_t \mathbf{b}_t\right\| = \min_{x\in\mathbb{Z}^n} \left\|\sum_{t=1}^{n} x_t \mathbf{b}_t\right\| . \tag{1}$$

Therefore, all coefficient combinations $\mathbf{u}$ that determine a vector of norm less than A are enumerated. In Equation 1 we replace all $\mathbf{b}_t$ by their orthogonaliza-

tion, i.e., $\mathbf{b}_t = \mathbf{b}_t^* + \sum_{j=1}^{t-1} \mu_{t,j}\mathbf{b}_j^*$ and get Equation (2):

$$\left\|\sum_{t=1}^{n} u_t\mathbf{b}_t\right\|^2 = \left\|\sum_{t=1}^{n}\left(u_t \cdot (\mathbf{b}_t^* + \sum_{j=1}^{t-1}\mu_{t,j}\mathbf{b}_j^*)\right)\right\|^2 = \sum_{t=1}^{n}(u_t + \sum_{i=t+1}^{n}\mu_{i,t}u_i)^2 \cdot \|\mathbf{b}_t^*\|^2 .$$

Let $\mathbf{c} \in \mathbb{R}^d$ with $c_t = \sum_{i=t+1}^{n}\mu_{i,t}u_i$ (line 8), which is predefined by all coefficients u_i with $n \geq i > t$. The intermediate norm l_t (line 4) is defined as $l_t = l_{t+1} + (u_t + c_t)^2 \|\mathbf{b}_t^*\|^2$. This is the norm part of Equation 2 that is predefined by the values u_i with $n \geq i \geq t$.

The algorithm enumerates the coefficients in reverse order, from u_n to u_1. This can be considered as finding a minimum in a weighted search tree. The height of the tree is uniquely determined by the dimension n. The root of the tree denotes the coefficient u_n. The coefficient values u_t for $1 \leq t \leq n$ determine the values of the vertices of depth $(n-t+1)$, leafs of the tree contain coefficients u_1. The inner nodes represent intermediate nodes, not complete coefficient vectors, i.e., a node on level t determines a subtree $(\perp, \ldots, \perp, u_t, u_{t+1}, \ldots, u_n)$, where the first $t-1$ coefficients are not yet set. l_t is the norm part predefined by this inner node on level t. We only enumerate parts of the tree with $l_t < A$. Therefore, the possible values for u_t on the next lower level are in an interval around c_t with $(u_t + c_t)^2 < (A - l_{t+1})/\|\mathbf{b}_t^*\|$, following the definition of l_t.

ENUM iterates over all possible values for u_t, as long as $l_t \leq A$, the current minimal value. If l_t exceeds A, enumeration of the corresponding subtree can be cut off, the intermediate norm l_t will only increase when stepping down in the tree, as $l_t \leq l_{t-1}$ always holds. The iteration over all possible coefficient values is (due to Schnorr and Euchner) performed in a zig-zag pattern. The values for u_t will be sequenced like either $c_t, c_t+1, c_t-1, c_t+2, c_t-2, \ldots$ or $c_t, c_t-1, c_t+1, c_t-2, c_t+2, \ldots$.

ENUM starts at the leaf $(1, 0, \ldots, 0)$ and gives the first possible solution for a shortest vector in the given lattice. The algorithm performs its search by moving up (when a subtree can be cut off due to $l_t \geq A$) and down in the tree (lines 13 and 7). The norm of leaf nodes is compared to A. If $l_1 \leq A$, it stores $A \leftarrow l_1$ and $\mathbf{u}_{min} \leftarrow \mathbf{u}$ (line 10), which define the current shortest vector and its size. When ENUM moves up to the root of the search tree it terminates and outputs the computed global minimum A and the corresponding shortest vector $\mathbf{u}_{min}$.

4 Algorithm for parallel enumeration of the shortest lattice vector

In this section we describe our parallel algorithm for enumeration of the shortest lattice vector. The algorithm is a parallel version of the algorithm presented in [14]. First we give the main idea of parallel enumeration. Secondly, we present a high level description. Algorithms 2 and 3 depict our parallel ENUM. Thirdly, we explain some improvements that speed up the parallelization in practice.

4.1 Parallel Lattice Enumeration

The main idea for parallelization is the following. Different subtrees of the complete search tree are enumerated in parallel independently from each other representing them as threads (Sub-ENUM threads). Using s processors, s subtrees can be enumerated at the same time. All threads ready for enumeration are stored in a list L, and each CPU core that has finished enumerating a subtree picks the next subtree from the list. Each of the subtrees is an instance of SVP in smaller dimension; the initial state of the sub-enumeration can be represented by a tuple $(\mathbf{u}, l, c, t)$.

When the ENUM algorithm increases the level in the search tree, the center (c_t) and the range $((A - l_{t+1})/\|\mathbf{b}_t^*\|)$ of possible values for the current index are calculated. Therefore, it is easy to open one thread for every value in this range.

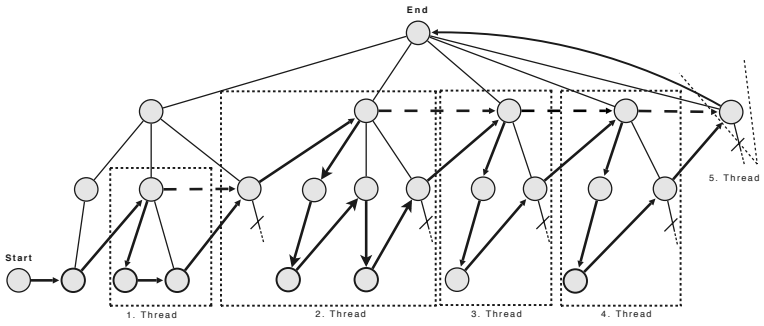

Fig. 1. Comparison of serial (solid line) and parallel (dashed line) processing of the search tree.

Figure 1 shows a 3-dimensional example and compares the flow of the serial ENUM with our parallel version. Beginning at the starting node the procession order of the serial ENUM algorithm follows the directed solid edges to the root. In the parallel version dashed edges represent the preparation of new Sub-ENUM threads which can be executed by a free processor unit. Crossed-out edges point out irrelevant subtrees. Threads terminate as soon as they reach either a node of another thread or the root node.

Extra Communication – Updating the Shortest Vector. Again, we denote the current minimum, the global minimum, as A. In our parallel version, it is the global minimum of all threads. As soon as a thread has found a new minimum, the Euclidean norm of this vector is written back to the shared memory, i.e. A is updated. At a certain point every thread checks the global minimum whether another thread has updated A and, if so, uses the updated one. The smaller A is, the faster a thread terminates, because subtrees that exceed the current minimum can be cut off in the enumeration. The memory access for this update operation is minimal, only one integer value has to be written back or read from shared memory. This is the only type of communication among threads, all other computations can be performed independently without communication overhead.

4.2 The Algorithm for Parallel Enumeration

Algorithm 2 shows the main thread for the parallel enumeration. It is responsible to initialize the first Sub-ENUM thread and manage the thread list L. A Sub-ENUM thread (SET) is represented by the tuple $(\mathbf{u}, l, c, t)$, where $\mathbf{u}$ is the coefficient vector, l the intermediate norm of the root to this subtree, c the search region center and t the lattice dimension minus the starting depth of the parent node in the search tree.

Algorithm 2: Main thread for parallel enumeration

Input: Gram-Schmidt coefficients $(\mu_{i,j})_{1\leq j\leq i\leq n}$, $\|\mathbf{b}_1^*\|^2 \dots \|\mathbf{b}_n^*\|^2$
Output: $\mathbf{u}_{min}$ such that $\left\|\sum_{i=1}^{n} u_i \mathbf{b}_i\right\| = \lambda_1(\mathcal{L}(\mathbf{B}))$

```
1  A ← ‖b*_1‖², u_min ← (1, 0, ..., 0)                          ▷ Global variables
2  u ← (1, 0, ..., 0), l ← 0, c ← 0, t ← 1                      ▷ Local variables
3  L ← {(u, l, c, t)}                                           ▷ Initialize list
4  while L ≠ ∅ or threads are running do
5  |  if L ≠ ∅ and cores available then
6  |  |  pick Δ = (u, l, c, t) from L
7  |  |  start Sub-ENUM thread Δ = (u, l, c, t) on new core
8  |  end
9  end
```

Algorithm 3: Sub-ENUM thread (SET)

Input: Gram-Schmidt coefficients $(\mu_{i,j})_{1\leq j\leq i\leq n}$, $\|\mathbf{b}_1^*\|^2 \dots \|\mathbf{b}_n^*\|^2$, $(\bar{\mathbf{u}}, \bar{l}, \bar{c}, \bar{t})$

```
1  u ← ū, l ← (0, ..., 0), c ← (0, ..., 0)
2  t ← t̄, l_{t+1} ← l̄, c_t ← c̄, bound ← n
3  while t ≤ bound do
4  |  l_t ← l_{t+1} + (u_t + c_t)² ‖b*_t‖²
5  |  if l_t < A then
6  |  |  if t > 1 then
7  |  |  |  t ← t − 1                                  ▷ move one layer down in the tree
8  |  |  |  c_t ← Σ_{i=t+1}^{n} u_i μ_{i,t},   u_t ← ⌈c_t⌋
9  |  |  |  if bound = n then
10 |  |  |  |  L ← L ∪ (u, l_{t+2}, c_{t+1}, t + 1)      ▷ insert new SET in list L
11 |  |  |  |  bound ← t
12 |  |  |  end
13 |  |  else
14 |  |  |  A ← l_t,   u_min ← u                        ▷ set new global minimum
15 |  |  end
16 |  else
17 |  |  t ← t + 1                                     ▷ move one layer up in the tree
18 |  |  choose next value for u_t using the zig-zag pattern
19 |  end
20 end
```

Whenever the list contains a SET and free processor units exist, the first SET of the list is executed. The execution of SETs is performed by Algorithm 3. We process the search tree in the same manner as the serial algorithm (Algorithm 1), except the introduction of the loop bound *bound* and the handling of new SETs (lines $9-11$). First, the loop bound controls the termination of the subtree and prohibits that nodes are visited twice. Second, only the SET whose bound is set to the lattice dimension is allowed to create new SETs. Otherwise, if we

allow each SET to create new SETs by itself, this would lead to an explosion of the number of threads and each thread has too few computations to perform. We denote the SET with bound set to n by *unbounded SET* (USET). At any time, there exists only one USET that might be stored in the thread list L.

As soon as an USET has the chance to find a new minimum within the current subtree (lines $5-6$), its bound is set to the current t value. Thereby, it is transformed to a SET and the recent created SET becomes the USET.

4.3 Improvements

We presented a first solution for the parallelization of the ENUM algorithm providing a runtime speed-up by a divide and conquer technique. We distribute subtrees to several processor units to search for the minimum. Our improvements deal with the creation of SETs and result in significantly shorter running time. Recall the definitions of Sub-ENUM thread (SET) and unbounded Sub-ENUM thread (USET). By now we call a node, where a new SET can be created, a candidate. Note that a candidate can only be found in an USET.

The following paragraphs show worst cases of the presented parallel ENUM algorithm and present possible solutions to overcome the existing drawbacks.

Threads within threads. Our parallel ENUM algorithm allows to create new SETs only by an USET. The avoidance of producing immense overhead which happens by permitting the creation of new SETs by any SET, backs our decision that it suffices to let only USET create new instances. However, if an USET creates a new SET at a node of depth 1, then this new SET is executed by a single processor sequentially. Note that this SET solves the SVP problem in dimension $n-1$. It turns out that in the case the depth of a current analyzed node in ENUM is sufficient far away from the depth t of the starting node, the creation of a new SET is advantageous according to the overall running time and the number of simultaneously occupied processors. Therefore, we introduce a bound s_{deep} which expresses what we consider to be sufficient far away, i.e. if a SET visits a node with depth k fulfilling the equation $k-t \geq s_{deep}$ where t stands for the depth of the starting node and it is not an USET, then this SET is permitted to create a new SET once.

Thread Bound. Although we avoid the execution of SETs where the dimension of the subtree is too big, we are still able to optimize the parallel ENUM algorithm by considering execution bounds. We achieve additional performance improvements by the following idea. Instead of generating SETs in each possible candidate, we consider the depth of the node. This enables us to avoid big subtrees for new SETs by introducing an upper bound s_{up} representing the minimum distance of a node to the root to become a candidate. If ENUM visits a node with depth t fulfilling $n-t > s_{up}$ and this node is a candidate, we no longer make a subtree ready for a new SET. We rather prefer to behave in that situation like the serial ENUM algorithm. Good choices for the above bounds s_{deep} and s_{up} are evaluated in Section 5.

5 Experiments

We performed numerous experiments to test our parallel enumeration algorithm. We created 5 different random lattices of each dimension $n \in \{42, \ldots, 56\}$ in the sense of Goldstein and Mayer [6]. The bitsize of the entries of the basis matrices were in the order of magnitude of $10n$. We started with bases in Hermite normal form, then LLL-reduced the bases (using LLL parameter $\delta = 0.99$). The experiments were performed on a compute server equipped with four AMD Opteron (2.3GHz) quad core processors. We compare our results to the highly optimized, serial version of fpLLL in version 3.0.12, the fastest ENUM implementation known, on the same platform. The programs were compiled using gcc version 4.3.2. For handling parallel processes, we used the Boost-Thread-sublibrary in version 1.40. Our C++ implementation uses double precision to store the Gram-Schmidt coefficients $\mu_{i,j}$ and the $\|\mathbf{b}_1^*\|^2, \ldots, \|\mathbf{b}_n^*\|^2$. Due to [12], this is suitable up to dimension 90, which seems to be out of the range of today's enumeration algorithms.

We tested the parallel ENUM algorithm for several s_{deep} values and concluded that $s_{deep} = \frac{25}{36}(n-t)$ seems to be a good choice, where t is the depth of the starting node in a SET instance. Further, we use $s_{up} = \frac{5}{6}n$.

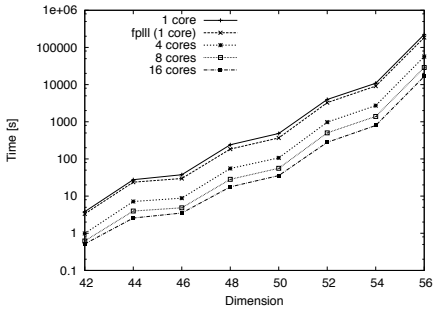

Fig. 2. Average runtimes of enumeration of 5 random lattices in each dimension, comparing our multi-core implementation to fpLLL's and our own single-core version.

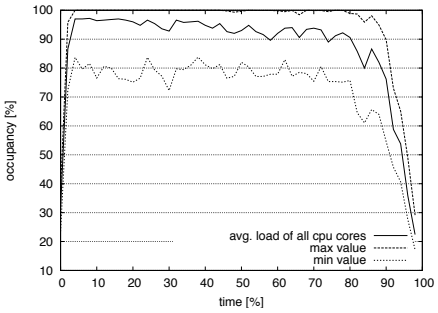

Fig. 3. Occupancy of the cores. The x-axis marks the percentage of the complete runtime, the y-axis shows the average occupancy of all CPU cores over 5 lattices.

n	42	44	46	48	50	52	54	56
1 core	3.81	27.7	37.6	241	484	3974	10900	223679
4 cores	0.99	7.2	8.8	55	107	976	2727	56947
8 cores	0.62	4.0	4.8	28	56	504	1390	28813
16 cores	0.52	2.6	3.5	18	36	280	794	16583
fpLLL 1 core	3.32	23.7	29.7	184	367	3274	9116	184730

Table 1. Average time in seconds for enumeration of lattices in dimension n.

Table 1 and Figure 2 present the experimental results that compare our parallel version to our serial algorithm and to the fpLLL library. We only present the timings, as the output of the algorithms is in all cases the same, namely a shortest non-zero vector of the input lattice. The corresponding speed-ups are shown in Figure 4.

To show the strength of parallelization of the lattice enumeration, we first compare our multi-core versions to our single-core version. The best speed-ups are 4.5 ($n = 50$) for 4 cores, 8.6 ($n = 50$) for 8 cores, and 14.2 ($n = 52$) for 16 cores. This shows that, using s processor cores, we sometimes gain speed-ups of more than s, which corresponds to an efficiency of more than 1. This is a very untypical behavior for (standard) parallel algorithms, but understandable for graph search algorithms as our lattice enumeration. It is caused by the extra communication for the write-back of the current minimum A.

The highly optimized enumeration of fpLLL is around 10% faster than our serial version. Compared to the fpLLL algorithm, we gain a speed-up of up to 6.6 ($n = 48$) using 8 CPU cores and up to 11.7 ($n = 52$) using 16 cores. This corresponds to an efficiency of 0.825 (8 cores) and 0.73 (16 cores), respectively.

Figure 3 shows the average, the maximum, and the minimum occupancy of all CPU cores during the runtime of 5 lattices in dimension $n = 52$. The average occupancy of more than 90% points out that all cores are nearly optimally loaded; even the minimum load values are around 80%. These facts show a good balanced behaviour of our parallel algorithm.

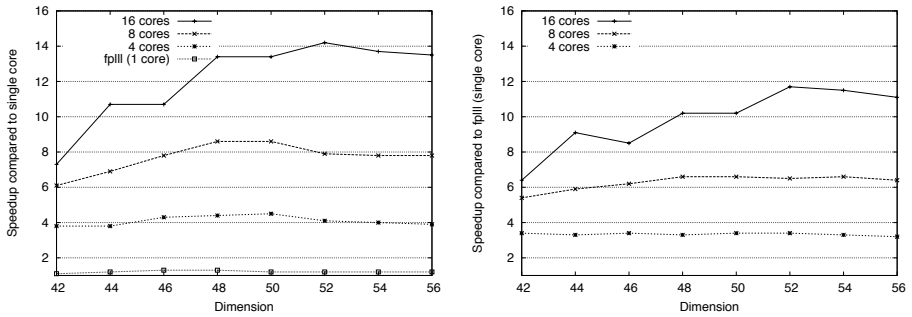

Fig. 4. Average speed-up of parallel ENUM compared to our single-core version (left) and compared to fpLLL single-core version (right).

6 Conclusion and Further Work

In this paper we have presented a parallel version of the most common algorithm for solving the shortest vector problem in lattices, the ENUM algorithm. We have shown that a huge speed-up and a high efficiency is reachable using multi-core processors. As parallel versions of LLL are already known, with our parallel ENUM we have given evidence that both parts of the BKZ reduction algorithm can be parallelized. It remains to combine both, parallel LLL and parallel ENUM, to a parallel version of BKZ. Our experience with BKZ shows that

in higher blocksizes of ≈ 50 ENUM takes more than 99% of the complete runtime. Therefore, the speed-up of ENUM will directly speed up BKZ reduction, which in turn influences the security of lattice based cryptosystems. Furthermore, to enhance scalability further, an extension of our algorithm to parallel systems with multiple multicore nodes is considered as future work.

Acknowledgments

We thank Jens Hermans, Richard Lindner, Markus Rückert, and Damien Stehlé for helpful discussions and their valuable comments. We thank Michael Zohner for performing parts of the experiments. We thank the anonymous reviewers for their comments.

References

1. Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: STOC 2001. pp. 601–610. ACM (2001)
2. Backes, W., Wetzel, S.: Parallel lattice basis reduction using a multi-threaded Schnorr-Euchner LLL algorithm. In: Euro-Par. LNCS, vol. 5704, pp. 960–973. Springer (2009)
3. Fincke, U., Pohst, M.: A procedure for determining algebraic integers of given norm. In: European Computer Algebra Conference. LNCS, vol. 162, pp. 194–202. Springer (1983)
4. Gama, N., Nguyen, P.Q.: Predicting lattice reduction. In: Eurocrypt 2008. LNCS, vol. 4965, pp. 31–51. Springer (2008)
5. Gama, N., Nguyen, P.Q., Regev, O.: Lattice enumeration using extreme pruning (2010), to appear in Eurocrypt 2010
6. Goldstein, D., Mayer, A.: On the equidistribution of Hecke points. Forum Mathematicum 2003, 15:2 pp. 165–189 (2003)
7. Hermans, J., Schneider, M., Buchmann, J., Vercauteren, F., Preneel, B.: Parallel shortest lattice vector enumeration on graphics cards. In: Africacrypt. LNCS, vol. 6055, pp. 52–68. Springer (2010)
8. Kannan, R.: Improved algorithms for integer programming and related lattice problems. In: STOC 1983. pp. 193–206. ACM (1983)
9. Lenstra, A., Lenstra, H., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen 4, 515–534 (1982)
10. Micciancio, D., Voulgaris, P.: Faster exponential time algorithms for the shortest vector problem. In: SODA 2010 (2010)
11. Pujol, X.: Recherche efficace de vecteur court dans un réseau euclidien. Masters thesis, ENS Lyon (2008)
12. Pujol, X., Stehlé, D.: Rigorous and efficient short lattice vectors enumeration. In: Asiacrypt 2008. LNCS, vol. 5350, pp. 390–405. Springer (2008)
13. Pujol, X., Stehlé, D.: Accelerating lattice reduction with FPGAs (2010), to appear in Latincrypt 2010
14. Schnorr, C.P., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. Mathematical Programming 66, 181–199 (1994)
15. Villard, G.: Parallel lattice basis reduction. In: ISSAC 1992. pp. 269–277. ACM (1992)