Universidade do Minho
Escola de Engenharia

Fábio José Gonçalves Correia

Assessing the Hardness of SVP Algorithms on Multi-core CPUs

Outubro de 2014

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Fábio José Gonçalves Correia

Assessing the Hardness of SVP Algorithms on Multi-core CPUs

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor Alberto José Proença
Artur Miguel Matos Mariano

Outubro de 2014

Anexo 3

DECLARAÇÃO

Nome

Fábio José Gonçalves Correia

Endereço electrónico: pg22817@alunos.uminho.pt Telefone: 968586389 / ____________

Número do Bilhete de Identidade: 13881126

Título dissertação □/tese □

Assessing the Hardness of SVP Algorithms on Multi-core CPUs

Orientador(es):

Prof. Alberto José Proença e Artur Miguel Matos Mariano

Ano de conclusão: 2014

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Mestrado em Engenharia Informática

Universidade do Minho, 31/10/2014

Assinatura: Fábio José Gonçalves Correia

Acknowledgments

Ao meu orientador, Alberto Proença, quer pela disponibilidade total, quer pelo rigor exigido, assim como pelo apoio na escrita da dissertação. Ao meu co-orientador Artur Mariano, que sempre se mostrou disponível para discutir problemas relacionados com a tese e por me ter aberto a oportunidade de efectuar um estágio no âmbito da minha dissertação de mestrado na Alemanha.

Ao programa Erasmus Placements, que me possibilitou a experiência de estudar num país estrangeiro, a Alemanha, ajudando-me a evoluir tanto a nível pessoal, com muitas novas amizades, como a nível profissional, permitindo trabalhar num ambiente bastante diferente do que conhecia em Portugal.

I would like to thank the Institute for Scientific Computing for receiving me in Darmstadt. I would also like to thank Florian Göpfert, Özgür Dagdelen, Damien Stehlé and Erik Agrell for very useful discussions on lattices. A special thanks to Thomas Arneich for all the hours spent in discussions, which also lead to new ideas.

A todos os meus amigos de LEI que me suportaram durante o meu percurso académico e que me possibilitaram os melhores anos da minha vida, em especial ao Cristiano Sousa pelas várias discussões muito úteis durante o último ano.

Um agradecimento especial à minha família, pais e irmão, por todo o apoio que deram e que me ajudaram a chegar onde cheguei.

Abstract

Lattice-based cryptography has been a hot topic in the past decade, since it is believed that lattice-based cryptosystems are immune against attacks operated by quantum computers. The security of this type of cryptography is based on the hardness of algorithms that solve lattice-based problems, namely the Shortest Vector Problem (SVP). Therefore, it is important to assess the performance of such algorithms on High Performance Computing (HPC) systems.

This dissertation compares a wide range of algorithms that solve the SVP, the SVP-solvers, namely, the Voronoi cell-based algorithm and two enumeration-based solvers, SE++ and ENUM. We show that different techniques and optimizations used to significantly improve the performance of ENUM can also be applied to other enumeration algorithms, namely the extreme pruning technique and the optimization that avoids symmetric branches of the enumeration tree.

We present the first practical results of the Voronoi cell-based algorithm and compare its performance to the mentioned enumeration algorithms. The Voronoi cell-based algorithm performed considerably worse, although it displays potential for parallelization.

The optimization that avoids the computation of symmetric branches improved the performance of SE++ by almost 50%, thus outperforming ENUM by a factor of 3%, on the average. Parallel versions of the enumeration algorithms were implemented on a shared memory system based on a dual (8+8)-core device, which, in some instances, scale super-linearly for up to 8 threads and linearly for 16 threads. The parallel versions of the enumeration with extreme pruning algorithms were parallelized with MPI and achieved speedups of up to 12.96x with 16 processes with ENUM on a lattice in dimension 74.

We show that an efficient parallel implementation of ENUM can be integrated into BKZ, a lattice basis reduction algorithm, to parallelize it efficiently, since for high block-sizes almost all of the execution time of BKZ is spent on ENUM. This implementation of BKZ achieves speedups of up to 13.72x for a lattice in dimension 60, reduced with block-size 50. We also compared the quality of the output bases to other BKZ implementations, namely AC_BKZ, an implementation developed in colaboration with Thomas Arnreich, and G_BKZ_FP, an implementation publicly available in the NTL library. Our implementation showed to compute the bases with better quality, in the general case.

Finally, a novel parallel approach for the enumeration with extreme pruning is proposed. This approach promises to significantly improve the performance of the enumeration with extreme pruning, since much higher number of cores can be used to achieve higher speedups.

Resumo

A criptografia baseada em retículos tem vindo a tornar-se um tópico central ao longo da última década, dado que se acredita que criptosistemas baseados em retículos sejam resistentes a ataques infligidos com computadores quânticos. A segurança destes criptosistemas é medida pela eficácia dos algoritmos que resolvem problemas centrais em retículos, como o problema do vector mais curto. Por isso, é importante avaliar o desempenho destes algoritmos em arquitecturas computacionais de alto rendimento.

Esta dissertação compara uma grande variedade de algoritmos que resolvem o problema do vector mais curto, nomeadamente o algoritmo baseado em células de Voronoi e dois algoritmos de enumeração, o SE++ e o ENUM. Além disso, mostramos que é possível aplicar várias técnicas e optimizações do ENUM a outros algoritmos de enumeração, nomeadamente a técnica de poda extrema e a optimização que evita a computação de ramos simétricos da árvore de enumeração.

Foram ainda apresentados os primeiros resultados práticos do algoritmo de células de Voronoi, cujo desempenho foi comparado ao desempenho dos algoritmos de enumeração mencionados. O algoritmo baseado em células de Voronoi, apesar do apresentar potencial de paralelização, apresenta um desempenho bastante pior que as restantes implementações.

A optimização que evita a computação de ramos simétricos acelera o SE++ em quase 50%, o que lhe permite ultrapassar o ENUM em termos de desempenho por um factor de 3%, no caso médio. Além disso, foram implementadas versões paralelas do algoritmos de enumeração, quer num sistema de memória partilhada baseado num dispositivo com 8+8 núcleos computacionais, para as variantes sem poda extrema, quer em memória distribuída, para as variantes com poda extrema. Os resultados mostram que as implementações em memória partilhada atingem, em certos casos, acelerações super-lineares até 8 *threads* e lineares para 16 *threads*. As implementações em memória distribuída, por seu turno, são aceleradas em cerca de 13 vezes para 16 processos.

Também é mostrado que é possível integrar uma versão paralela eficiente do algoritmo ENUM no BKZ, um algoritmo de redução da base de um retículo, como forma de o paralelizar eficientemente, dado que a grande maioria do tempo de execução é gasto em chamadas ao ENUM para tamanhos de bloco grandes. Esta implementação alcança acelerações até 13.72 vezes para o retículo na dimensão 60, reduzido com um tamanho de bloco 50. A qualidade das bases dos retículos, computados por esta implementação, foi comparada a outros implementações do BKZ, nomeadamente o AC_BKZ, uma implementação desenvolvida em conjunto com Thomas Arnreich, e o G_BKZ_FP, uma implementação acessível publicamente na biblioteca NTL. A nossa implementação apresentou computar bases com uma qualidade superior às restantes, no caso geral.

Por fim, é proposta uma abordagem paralela inovadora que promete melhorar o desempenho dos algoritmos de enumeração com poda extrema significativamente, dado que poderá tirar partido de um maior número de núcleos computacionais.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Lattices are discrete and infinite subgroups of the m-dimensional Euclidean space $\mathbb{R}^m$. A lattice $\mathcal{L}$ is generated by all linear combinations of a basis **B**, a set of linearly independent vectors $\mathbf{b}_1$,...,$\mathbf{b}_n$ in $\mathbb{R}^m$, and is denoted by:

$$\mathcal{L}(\mathbf{B}) = \{\mathbf{x} \in \mathbb{R}^m : \mathbf{x} = \sum_{i=1}^{n} \mathbf{c}_i \mathbf{b}_i, \mathbf{c} \in \mathbb{Z}^n\} \tag{1.1}$$

where n is the *dimension* of the lattice.

Vectors and matrices are represented in this dissertation in bold face, vectors in lower-case and matrices in upper-case, as in vector **v** and matrix **M**. The auxiliary functions used by the algorithms are written as in function(). The dot product of two vectors **v** and **p** is denoted by $\langle \mathbf{v},\mathbf{p} \rangle$. The transpose of a matrix is given by $\mathbf{M}^T$. $\lceil a \rfloor$ rounds *a* to the nearest integer number. The absolute value of a is given by $|a|$.

The Euclidean norm of a vector $\mathbf{v} \in \mathbb{R}^n$, denoted by $||\mathbf{v}||_2$ or simply $||\mathbf{v}||$, is the distance spanned from the origin of the lattice to the point given by the vector **v**, i.e. $||\mathbf{v}|| = \sqrt{\sum_{i=1}^{n} \mathbf{v}_i^2}$, where $\mathbf{v}_i$ is the ith coordinate of **v**. The norm of the shortest vector of the lattice is given by $\lambda_1(\mathcal{L}(\mathbf{B}))$.

Lattices and basis reduction algorithms are used to solve relevant problems in mathematics and computer science. A reduced basis is a basis where the vectors are short and nearly orthogonal. In mathematics, an example is the study of the geometry of numbers (Cassels, 1997). In computer science, lattices has played a key role in the theory of integer programming (Lenstra, 1983; Kannan, 1987), in factoring polynomials over the rationals (Lenstra et al., 1982), in solving low density subset-sum problems (Coster et al., 1992), in checking the solvability by radicals (Landau and Miller, 1985) and in solving many cryptanalytic problems (Odlyzko, 1990; Bellare et al., 1997; Joux and Stern, 1998; Nguyen and Stern, 2001).

Lattice-based cryptography has been a hot topic in the past decade, since lattice-based cryptosystems are believed to be resistant against attacks operated with quantum computers. Cryptosystems based on lattices are said to be secure when specific lattice problems can not be solved in a timely manner.

Well known mathematical problems based on lattices are the shortest vector problem (SVP) and the closest vector problem (CVP). The SVP aims to find the shortest non-zero vector of the lattice, i.e., to find $\mathbf{x} \in \mathbb{Z}^n \backslash 0$ minimizing $||\mathbf{B} \cdot \mathbf{x}||$ in the Euclidean norm. The CVP aims to find the closest lattice vector to a given target vector $\mathbf{t} \in \mathbb{R}^m$, i.e., to find $\mathbf{x} \in \mathbb{Z}^n$ that minimizes $||\mathbf{B} \cdot \mathbf{x} - \mathbf{t}||$. Both have been proved to be NP-hard (Boas, 1981; Ajtai, 1998; Blömer and Seifert, 1999). These problems have been thoroughly studied to be exactly solved and also to efficiently find approximate solutions. Algorithms that solve the SVP are known as SVP-solvers, while algorithms that solve the CVP are known as CVP-solvers.

1.1 Motivation & Goals

Lattice-based cryptography has gained a renewed interest in the past decade, since it is believed that cryptosystems based on lattices can be more resistant to attacks, even when operated with quantum computers. Lattice-based cryptosystems are secure when specific lattice problems can not be solved in a timely manner. This dissertation focuses on two classes of algorithms that solve the SVP, one of the problems in question.

In the last decade, SVP-solvers have been thoroughly studied. New algorithms were proposed (e.g., (Micciancio and Voulgaris, 2009; Gama et al., 2010)) and some existing algorithms were optimized to achieve a higher performance (e.g., (Blömer and Naewe, 2009; Fitzpatrick et al., 2014)). Recently, some parallel implementations were also proposed (Dagdelen and Schneider, 2010; Mariano et al., 2014b).

Enumeration algorithms are currently the fastest in practice, mainly due to a specific technique, extreme pruning. This motivated the implementation of two enumeration algorithms and the application of the extreme pruning technique on them, followed by a performance assessment of the implementations of both algorithms, with and without extreme pruning. After this analysis, this dissertation also aimed to implement efficient parallel versions of the algorithms that would significantly speed them up.

Research in another algorithm, the Voronoi cell-based algorithm, has not been aimed as much as the others in the past. This algorithm seems to display potential for an efficient parallel implementation. This motivated us to implement this algorithm, showing the first practical results.

Finally, one of the most important components in every algorithm related to lattices is a lattice basis pre-processing, the lattice basis reduction mentioned earlier. Efficient lattice basis reduction algorithms could lead to stronger basis reductions, which could reduce the execution time of SVP-solvers considerably. Another goal of this work is to efficiently parallelize one of these lattice basis reduction algorithms, the BKZ, which is currently one of the most used in practice.

1.2 Contribution

The work developed during this dissertation lead to several scientific contributions in the field related to SVP-solvers. These include:

- Efficient implementations of several SVP-solvers and BKZ, from sequential to parallel versions, most of them not publicly available; List of algorithms:
  - Sequential Voronoi cell-based algorithm.
  - Enumeration, namely sequential and parallel SE++, ENUM, both with and without extreme pruning, and the improved SE++, a variant with an optimization that avoids the computation

  of symmetric branches of the enumeration tree.
  - Lattice basis reduction, namely sequential and parallel BKZ implementations, which internally use the LLL algorithm and the Gram-Schmidt orthogonalization.

- A comparative performance evaluation of several SVP-solver implementations, both sequential and parallel.
- A basis quality assessment of BKZ implementations.
- Proposal of a novel parallel implementation that merges ENUM (with extreme pruning) with the parallel BKZ reduction, on distributed memory systems.

1.3 Outline

The rest of this dissertation is organized as follows. Chapter 2 details the kind of computing platforms our implementations will target and their characteristics. Chapter 3 presents the current approaches to reduce the lattice basis and the key algorithms to solve the SVP. Chapter 4 discusses the performance results achieved with the implemented sequential and parallel versions of the algorithms. It also assesses the lattice basis quality, comparing our implementations with the ones available at the NTL[1] library. Chapter 5 concludes the dissertation with a critical analysis of the obtained results, leaving suggestions for future work on relevant topics that could not be fully covered on this dissertation.

[1]The Number Theory Library, a portable C++ library, developed by Victor Shoup; more details at http://www.shoup.net/ntl/

Chapter 2

Target Platforms

Computer clusters are currently the most used systems for High Performance Computing (HPC). A cluster is built of a set of interconnected computing nodes, where each node may have one or more multi-core CPU devices and eventually a computing accelerator, for example, a GPU (Graphics Processing Unit) or a many-core CPU device, such as the Intel Many Integrated Cores (MIC) device. When a node or a computing system has attached an accelerator, we consider this system to be heterogeneous.

A program can be executed on a single node or on multiple nodes of a cluster. If the program is prepared to explore parallelism it will, therefore, distribute its work among the available resources to reduce its execution time. There are two major paradigms in parallel computing: shared memory and distributed memory.

On the shared memory paradigm, the processors work on different tasks of a program, each implemented as a process or a thread. When tasks are implemented as threads, a single copy of the data can reside in a global memory, shared among all threads.

In the distributed memory model, the work and associated data is distributed by multiple computing nodes. This is particularly useful for programs where a large data domain can be partitioned by several independent memory banks, allowing each node to compute only on part of the data, increasing the performance of the program.

Another paradigm that is becoming more used is a hybrid paradigm, which joins the advantages of shared memory with distributed memory. In this approach, each node has a smaller data block and the different cores of a node work on the same block.

2.1 The Architecture

The parallel implementations of the SVP algorithms in this dissertation work, aimed the heterogeneous nodes as the main target platform, where the CPU devices were Intel Xeon, based on the Sandy Bridge micro-architecture.

With the increasing demand on architectures that simultaneously compute massive amounts of data, 2 types of heterogeneous HPC systems have emerged: systems using GPUs as attached computing accelerators and systems using the Intel Xeon Phi (based on the Many Integrated Core architecture - MIC) as accelerators. Figure 2.1 shows an example of an heterogeneous platform, which uses multiple CPU devices and multiple accelerators to perform their computation.

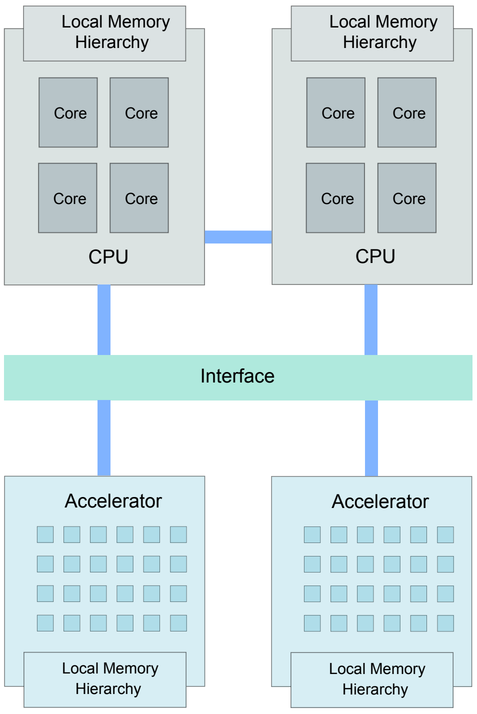


Figure 2.1: Example of an heterogeneous platform.

2.2 Computing Accelerators

The MIC is a multiprocessor computer architecture developed by Intel. It is possible to use programming languages, models and tools from traditional Intel Xeon processors on this architecture. Implementations for Intel Xeon processors will also work on Intel Xeon Phi coprocessors. But to get high performance out of this device, further improvements and optimizations have to be done. Currently, MIC coprocessors provide up to 61 cores and hardware support for 4 threads per core. Knights Corner, the current Xeon Phi implementation, uses for each core an updated version the old Pentium P54C, which only supports in-order execution and SIMD instructions on 512 bit data (from the previous Intel Larrabee, not compatible with SVX-512). Although each core has 2 private cache levels, while usual Xeons have a large third level cache (L3), the data on all all L2 caches can be accessed by all cores as a L3 cache. On the other hand, Knight Landing, the next generation MIC-based architecture, use Airmont (14-nm Atom) core and supports 384 GB of DDR4 RAM and AVX-512F (AVX3.1) SIMD instructions.

However, other heterogeneous platforms use GPUs as accelerators. Their optimization in programs that execute the same instructions over multiple data and their high bandwidth allows them to reach higher performances than CPUs in many problems.

Early GPU devices were specifically designed for computer graphics, where images are rendered from geometric objects. The color of the pixels that constitute the image are computed in parallel and sent to the screen. When GPUs started to increase their complexity and performance, the interest in using them in scientific computing also increased. A GPU is accessed and controlled by the CPU, but to allow concurrent execution and memory transfer, they work asynchronously.

NVidia GPUs used in HPC have the following organization:

- several SIMD (Single Instruction Multiple Data) cores, where NVidia refers to these as Streaming Multiprocessors (SMs) and on the latest Kepler architectures NVidia changed into next-generation SMs or in short, SMXs (up to 16 SMXs).
- each SIMD core in Kepler has 192 Streaming Processors (SPs), also known as CUDA processors, basically, a single precision Floating Point (FP) unit and an Integer unit; previous Fermi generation had 32 SPs per SM.
- additional functional units at each SM/SMX, including double precision FP units and load/store units.
- each SIMD core in Kepler has 64K 32-bit registers, 48 KB read-only data cache and 64KB of configurable L1 cache and local memory.
- 1.5MB shared L2 cache in Kepler, doubles the L2 in Fermi.
- SMT (Simultaneous Multi-threading) support for up to 64 warps per SMX (in Kepler); warps are described below.

Figure 2.2 shows a diagram of the Kepler architecture. Current NVidia GPUs can be programmed using CUDA (Compute Unified Device Architecture). CUDA allows to do this by using a familiar language, like C or Fortran, just by adding specific directives for the execution on GPUs. CUDA uses a large number of threads organized into blocks to operate in a SPMD model. All blocks run the same program, where threads of the same block can communicate between them using a local memory shared among all threads of a block. Blocks are also split into *warps*. Each *warp* consists of 32 threads and all threads of the same *warp* execute the same instruction (following the SIMD model). At runtime, blocks are scheduled to SMs/SMXs.

To abstract the programmer from these technical details and to aid in the development of efficient applications, specific frameworks for heterogeneous programming have been developed. This allows the programmer to focus more on algorithmic problems, instead of wasting time with details related to multi-platform implementations or data management.

2.3 Development Aids for Heterogeneous Platforms

Programming efficiently GPUs requires a great knowledge about the target architecture to get the maximum performance, for example memory and thread organization. So, the programmer has to take into account

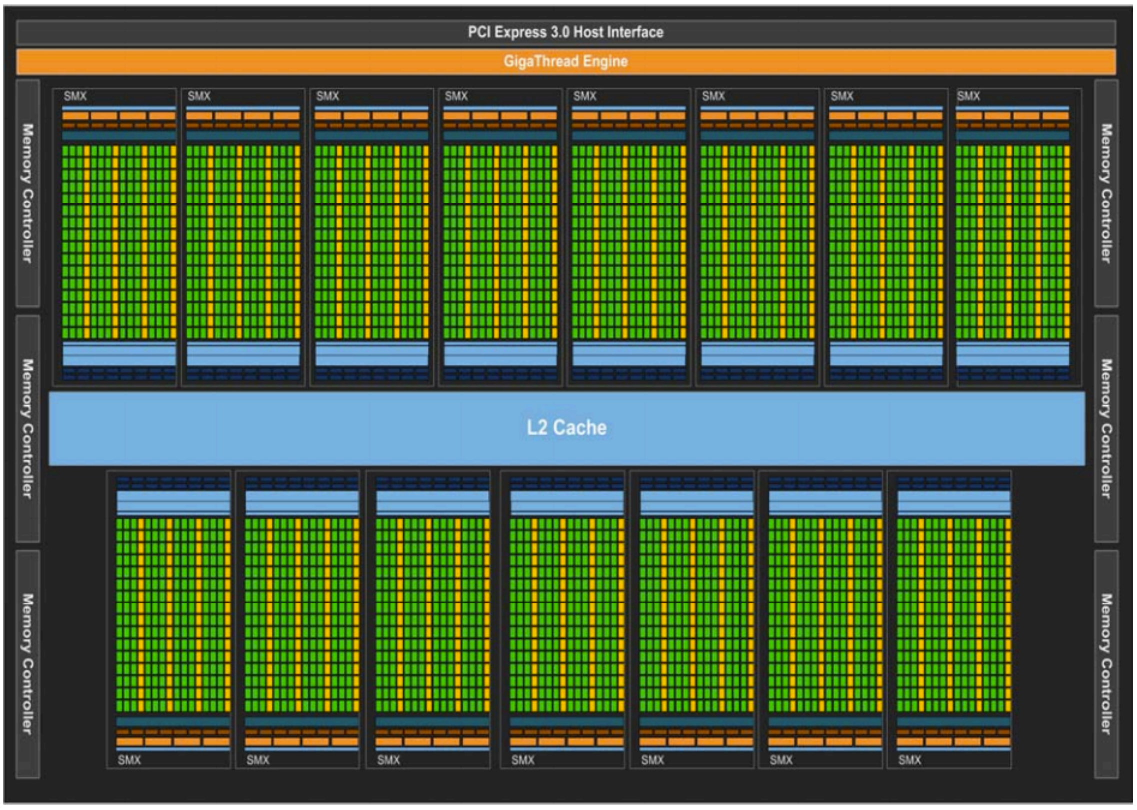


Figure 2.2: Diagram of the Kepler architecture (from NVidia documentation).

the way how data is organized in memory, thread scheduling,... There is also the risk of portability that when changing the system the performance might drop. To automate all this process of memory management, thread scheduling and portability, heterogeneous development frameworks have been created, e.g., StarPU (Augonnet et al., 2011) or GAMA(Barbosa, Sept. 2012). However, the main aim of heterogeneous development frameworks is to allow the simultaneous execution of a program on multi-core CPUs and GPUs.

StarPU, currently one of the most used heterogeneous frameworks, is a software tool that supports C extensions and aims to allow programmers to exploit the available resources, multi-core CPUs and GPUs, relieving them from the need to adapt their code to the target platforms. This framework is responsible to schedule tasks at runtime on CPU and/or GPU implementations and automatically manage data transfers on available CPUs and GPUs. This way, the programmer does not have to worry about scheduling issues and technical details associated with these data transfers.

Two of the most important data structures in StarPU are codelets and tasks. A codelet is a computational kernel that can be executed on distinct computational units, such as a multi-core CPU, a CUDA device or an OpenCL device. A task applies a codelet on a data set on the architecture where the codelet is implemented and controls how it is accessed (read and/or write). A task is an asynchronous operation and, therefore, submitting a task is also a non-blocking operation. A task can also define its priority as hint to the scheduler or a callback function, which is executed once the task is completed.

Tasks can also have data dependencies between them, which forces a sequential execution. A task might be identified by a unique number, called tag. Dependencies between tasks can be expressed by callback functions, by submitting other tasks or by expressing dependencies between tags.

Since tasks are scheduled in runtime, data has to be transferred automatically between processing units relieving the programmer from explicit data transfers. To avoid unnecessary data transfers, StarPU allows to keep multiple copies of the same data on different processing units as long as it is not modified and to store data where it was lastly needed even if it was modified.

Chapter 3

The Shortest Vector Problem on Lattices

The shortest vector problem aims to find the shortest non-zero vector of the lattice, i.e., to find $\mathbf{x} \in \mathbb{Z}^n \backslash 0$ that minimizes $||\mathbf{B} \cdot \mathbf{x}||$ in the Euclidean norm. This problem has been extensively studied during the last decades and three main families of SVP-solvers have emerged: sieving, enumeration and Voronoi cell-based algorithms. Sieving is a class of probabilistic and randomized algorithms that sieve a list of vectors, until a given stop criterion is met. Enumeration algorithms list all possible vectors within a given search radius and return the shortest among them. Voronoi cell-based algorithms compute the Voronoi relevant vectors and choose the shortest among them, which is also the shortest vector of the lattice.

The primary enumeration algorithm currently used is ENUM, which was proposed by Schnorr and Euchner (Schnorr and Euchner, 1994) and improved by Gama et al. (Gama et al., 2010). Another enumeration algorithm, SE++, based on ENUM, was proposed by Agrell et al. (Agrell et al., 2002) and improved by Ghasemmehdi and Agrell (Ghasemmehdi and Agrell, 2009). Although SE++ was originally proposed to solve the CVP, it was presented in (Agrell et al., 2002) how to transform it into an SVP-solver. Currently, the fastest probabilistic approach to solve the SVP, in practice, is an heuristic based on enumeration algorithms, the enumeration with extreme pruning (Gama et al., 2010).

The performance of SVP-solvers depend strongly on the quality of the basis. The more reduced the basis is, i.e., the shorter and more orthogonal the basis vectors, the faster they find a solution for the SVP. Such bases can be computed by performing a lattice basis reduction. Lattice basis reduction and SVP-solvers have a strong relation, since lattice basis reduction algorithms compute an approximate version of the SVP and use SVP-solvers as part of their logic to improve the quality of the computed basis.

3.1 Lattice Basis Reduction

Lattice basis reduction is the process of transforming a basis **B** into another basis **B**', whose vectors are reasonably short and nearly orthogonal. The quality of a basis depends on the shortness and orthogonality of the basis vectors. The shorter and more orthogonal the basis vectors, the better the quality of the basis, thus reducing the execution times of SVP- and CVP-solvers. The two most common lattice basis reduction algorithms in practice are the Lenstra-Lenstra-Lovász (LLL) and the Block Korkine-Zolotarev (BKZ).

The LLL algorithm was proposed in 1982 by Lenstra et al. (Lenstra et al., 1982). Despite of running in polynomial time, it only generates a basis of moderate quality. A basis is considered LLL-reduced if it is size-reduced and if $\|b_i^*\| \geq \|b_{i-1}^*\|/2, \forall i \in \{2..n\}$. A size-reduced basis is a basis where $\mu_{i,j} \leq 1/2$ for any $i > j$.

Another lattice basis reduction algorithm is the Hermite-Korkine-Zolotarev (HKZ). The basis vectors computed by this algorithm are more orthogonal, but it is also more expensive to compute them (Hanrot and Stehlé, 2008). A basis is HKZ-reduced if it is size-reduced, if $b_2, ..., b_n$ are also HKZ-reduced when b_1 is orthogonally projected, and if $\|b_1\| = \lambda_1(\mathcal{L}(\mathsf{B}))$.

The BKZ algorithm, proposed by Schnorr and Euchner (Schnorr and Euchner, 1994), combines the efficiency of the LLL with the quality of HKZ. This algorithm uses a sliding window on the basis and performs successively ENUM calls over the window as the basis is being traversed. If a shorter vector is found, it is added to the basis, which generates a dependency. This dependency is removed by an LLL reduction. The SVP is performed by the ENUM enumeration algorithm.

In 2011, another lattice basis reduction algorithm, BKZ 2.0, was proposed by Chen and Nguyen (Chen and Nguyen, 2011). This algorithm is an optimized version of the BKZ reduction algorithm that significantly reduces the execution time of BKZ by applying: (i) an early-abort, which stops the algorithm after a certain number of iterations instead of only stopping when it is no longer possible to get a better basis; (ii) a stronger lattice basis reduction on each block, by using BKZ instead of LLL; (iii) a maximum search radius on the ENUM to improve its performance; (iv) sound pruning, where specific bounding functions are used to significantly decrease the execution time of ENUM. Despite of being the best lattice basis reduction algorithm at the moment, there are some crucial details for its implementation that were omitted on the original paper, which are still not clear (e.g., the bounding functions used for the sound pruning).

Recently, Liu et al. proposed a new variant of the BKZ algorithm and parallelized it on distributed memory systems, using MPI (Liu et al., 2014). From here on, we will call one traversal of the sliding window over the basis, from the beginning to the end, one round. This implementation uses n processes to compute the ENUM calls of one round in parallel. Instead of performing an LLL-reduction after each ENUM call, it inserts the shorter vectors found by ENUM into the basis at their respective positions and reduces this new set of vectors at the end.

3.1.1 LLL Reduction

The LLL algorithm is fed with the coefficients and square norms of the Gram-Schmidt matrix. It reduces the basis iteratively. The index k indicates that, at any given moment, the basis vectors $(b_1, ..., b_{k-1})$ are LLL-reduced. The value of k can be either incremented or decremented at each loop iteration and stops when it reaches $n + 1$, when the entire basis $(b_1, ..., b_n)$ is guaranteed to be LLL-reduced.

On each iteration, the algorithm computes a size-reduction of the vector b_k, as in Algorithm 1. Afterwards, the Lovász condition is checked. If the Lovász condition holds true, nothing happens and the algorithm proceeds with the size-reduction of the next vector. Otherwise, the vector b_k is swapped with its predecessor b_{k-1}.

Schnorr and Euchner proposed a variant of the LLL algorithm that is called LLL with deep insertions

Algorithm 1: The size-reduction algorithm, as in Figure 2, presented in (Nguyen and Stehlé, 2006).

```
Input: A basis $(b_1, ..., b_n)$, its Gram-Schmidt orthogonalization and an index $k$.
Output: The basis where $b_k$ is size-reduced and the updated Gram-Schmidt orthogonalization.
1  for i=k-1 down to 1 do
2      $b_k = b_k - \lceil \mu_{k,i} \rfloor b_i$;
3      for j = 1 to i do
4          $\mu_{k,j} = \mu_{k,j} - \lceil \mu_{k,i} \rfloor \mu_{i,j}$;
5  Update the Gram-Schmidt orthogonalization ;
```

(Schnorr and Euchner, 1994). This algorithm replaces the swapping step by a "deep insertion", which improves the quality of the output basis significantly. When performing a deep insertion, the algorithm, instead of swapping the vector b_k with its predecessor, computes the index where to insert the vector b_k by consecutively performing Lovász tests with the preceding vectors, until it holds true. The vector b_k is then inserted right before the last vector where the Lovász condition is violated. The pseudo-code of the algorithm is presented in Algorithm 2.

Algorithm 2: The LLL algorithm, as in Figure 1, presented in (Nguyen and Stehlé, 2006).

```
Input: A basis $(b_1, ..., b_n)$ and $\delta \in (1/4, 1)$.
Output: An LLL-reduced basis with factor $(\delta, 1/2)$.
1  Compute the Gram-Schmidt orthogonalization, i.e., all $\mu_{i,j}$'s and $c_i$'s ;
2  $k = 2$;
3  while $k \leq n$ do
4      Size-reduce $b_k$ using Algorithm 1 ;
5      k' = k ;
6      while $k \geq 2$ and $\delta c_{k-1} > c_{k'} + \sum_{i=k-1}^{k'-1} \mu_{k',i}^2 c_i$ do
7          $k = k - 1$ ;
8      for i = 1 to k - 1 do
9          $\mu_{k,i} = \mu_{k',i}$ ;
10     Insert $b_{k'}$ right before $b_k$ ;
11     $k = k + 1$ ;
12 return $(b_1, ..., b_n)$ ;
```

LLL with Floating Point Arithmetic

The original LLL reduction algorithm was created to reduce a lattice using rational arithmetic. Floating-point arithmetic can speed up the lattice reduction process, but it can also introduce floating-point errors. These errors can not only cause the algorithm to produce incorrect results, but they also might prevent the algorithm to terminate.

The first theoretically provable floating-point variant of the LLL was proposed by Schnorr (Schnorr, 1988). A more efficient provable variant, L^2, was proposed by Nguyen and Stehlé (Nguyen and Stehlé, 2005). On the other side, heuristics are sometimes sufficient if they they are more efficient and have a high success probability. One important heuristic algorithm was proposed by Schnorr and Euchner (Schnorr and Euchner, 1994). It has been the inspiration for more recent algorithms, like the L^2 mentioned above.

3.1.2 BKZ Reduction

The BKZ reduction is a compromise between the LLL and the HKZ reduction and receives the block-size β as parameter. For $\beta = 2$, the algorithm outputs an LLL-reduced basis.When $\beta = n$, where n is the dimension of the lattice, the algorithm outputs an HKZ-reduced basis. For higher values of β, the basis is more reduced but it also takes more time to perform the reduction. The pseudo-code of the algorithm is presented in Algorithm 3.

Algorithm 3: The BKZ algorithm, as in Algorithm 1, presented in (Chen and Nguyen, 2011).

```
Input: A basis (b$_1, ..., $b$_n$), its Gram-Schmidt orthogonalization, i.e., $\mu$ and c$_i$, a block-size
       $\beta \in 2, ..., n$ and $\delta \in (1/4, 1)$.
Output: A BKZ $\beta$-reduced basis.
$z = 0$ ;
$j = 0$ ;
LLL((b$_1, ..., $b$_n), \delta$) ;
while $z < n - 1$ do
    $j = (j$ mod$(n-1)) + 1$ ;
    $k =$ min$(j + \beta - 1, n)$ ;
    $h =$ min$(k + 1, n)$ ;
    v = ENUM$(\mu_{[j,k]}, c_{j,k}$;
    if v $\neq (1, 0, ..., 0)$ then
        $z = 0$ ;
        LLL((b$_1, ..., \sum_{i=j}^{k}$ v$_i$b$_i$, b$_j, ..., $b$_h), \delta$) ;
    else
        $z = z + 1$;
        LLL((b$_1, ..., $b$_h), \delta$) ;
return (b$_1, ..., $b$_n$) ;
```

The algorithm starts by LLL-reducing the basis. Then it uses a sliding window, which contains at most β vectors and traverses the basis from the beginning until the end. Each window ranges from j to $k =$ min$(j + \beta - 1, n)$, where j is the index of the first basis vector of the window and k the index of the last basis vector of the window. For each block B$_{[j,k]}$, the enumeration algorithm ENUM is called. Whenever a shorter vector b^{new} is found, it is inserted into the basis on the position right before the beginning of the block, between the vectors b$_{j-1}$ and b$_j$. This insertion generates a linear dependency between the basis vectors. The linear dependency is removed by calling the LLL algorithm on the set of vectors (b$_1, ..., b_{j-1}$, b^{new}, b$_j, ..., b_h$), where $h =$ min$(k+1, n)$. If a better vector is not found, the LLL reduction is computed over B$_{[1,h]}$ to make sure that the next ENUM call is performed over an LLL-reduced window.

When the last window is computed, i.e., when $j = n - 1$ and $k = n$, it jumps back to the beginning of the basis by setting j back to 1. This process is repeated until the enumeration call does not output shorter vectors $n-1$ times consecutively, which means that the basis did not change for all indices $1 \leq j \leq n-1$.

3.2 Sieving Algorithms

A key issue in sieving algorithms is vector reduction, which consists in subtracting one vector by another, in order to make it short. The first sieving algorithm to solve the shortest vector problem, was proposed in 2001 by Ajtai, Kumar and Sivakumar, also known as AKS (Ajtai et al., 2001). This algorithm takes a large list of vectors and reduces them by one another, until two vectors are obtained such that their difference is the shortest non-zero vector of the lattice.

The AKS was simplified by Nguyen and Vidick (Nguyen and Vidick, 2008), who also introduced an heuristic variant of the algorithm with much smaller asymptotic complexity. Further improvements on AKS were proposed in 2009 (Blömer and Naewe, 2009) and on Nguyen and Vidick's heuristc in 2011 (Wang et al., 2011).

Other researchers proposed an efficient variant of the AKS, ListSieve, and its heuristic counterpart, GaussSieve (Micciancio and Voulgaris, 2009). These algorithms built a list of vectors instead of starting with a large lists. Then, the vectors are used to reduce the vectors that are incrementally sampled and added to the list. While ListSieve only reduce the sampled vectors against the elements in the list, GaussSieve, on the other hand, also reduces the list with the sampled vectors. Recently, a comprehensive comparison between ListSieve and GaussSieve has been perfomed (Mariano et al., 2014a) and further improvements on GaussSieve were also proposed (Fitzpatrick et al., 2014).

In 2011, Milde and Schneider (Milde and Schneider, 2011) proposed a parallel implementation of GaussSieve. However, the implementation changes the workflow of the algorithm and requires more iterations to converge as more threads are used, therefore, limiting its scalability. In 2013, a new parallel approach was presented that scales much better than Milde's and Schneider's, since it does not increase the time for the algorithm to converge at such a high pace as the first implementation (Ishiguro et al., 2013). However, this version extends the computation done by the original algorithm. Currently, the most efficient and scalable parallel implementation of GaussSieve consists in replacing the list of vectors by a lock-free list, atomically making any modification of this list (Mariano et al., 2014b).

3.3 Enumeration Algorithms

P. van Emde Boas showed, in 1981, that the general closest vector problem as a function of the dimension n is NP-hard (Boas, 1981). 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 (Pohst, 1981), and to 1983, when Kannan showed a different approach using a rectangular parallelepiped (Kannan, 1983). Extensions of these two approaches were published later on, by Fincke and Pohst, in 1985 (Fincke and Pohst, 1985), and by Kannan (following Helfrich's work (Helfrich, 1985)), in 1987 (Kannan, 1987). In 1994, Schnorr and Euchner proposed a significant improvement of Pohst's method (Schnorr and Euchner, 1994), that was later on found to be substantially faster than Pohst's and Kannan's approaches (Agrell et al., 2002). 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 target vector (Babai, 1986). Recently, Ghasemmehdi and Agrell showed that there are some redundant operations in the algorithm that can be eliminated, accelerating it substantially (Ghasemmehdi and Agrell, 2009).

In 2010, a new technique to solve the SVP, the enumeration with extreme pruning, which reduces the probability of finding the shortest vector, but also reduces the execution time, and by a much higher pace (Gama et al., 2010). This technique was applied to ENUM, the enumeration algorithm proposed by Schnorr and Euchner (Schnorr and Euchner, 1994).

Recently, Micciancio and Walter presented a new enumeration algorithm that outperforms other enumeration algorithms that do not use the extreme pruning technique by changing the basis pre-processing (Micciancio and Walter, 2014). However, it is still not known yet if the extreme pruning technique can be applied to this algorithm.

There is also work published on the parallelization of enumeration algorithms. On multi-core CPUs, Dagdelen and Schneider proposed an approach that distributes work among threads using a list, where only one thread may add work if there is any idle thread was proposed in (Dagdelen and Schneider, 2010). For GPUs, an implementation, where a CPU is used to create tasks that the GPU will compute when they get assigned, was proposed in 2010 (Hermans et al., 2010). After a given number of iterations the algorithm stops and might keep computing the same task or start computing a different, depending on whether the previous task did already finish or not. Recently, (Kuo et al., 2011) proposed an approach, based on (Hermans et al., 2010), for the enumeration with extreme pruning was proposed. This approach performs multiple calls to an extreme pruned enumeration algoritthm, running simultaneously on CPU and GPU, and uses a map-reduce to join the vectors at the end and to select the shortest.

3.3.1 SE++

The SE++ was initially proposed as a CVP-solver, although a variant to solve the SVP was also proposed in (Agrell et al., 2002). Ghasemmehdi and Agrell proposed an improvement for the CVP-solver that consists in avoiding redundant operations.

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 (Agrell et al., 2002) 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**. Finally, the sphere decoding is fed with the dimension of the lattice n 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 pro-

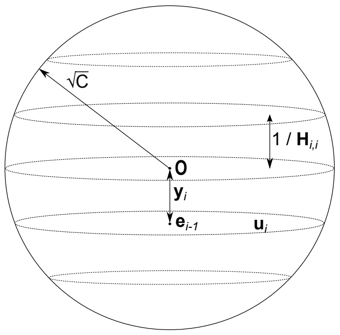


Figure 3.1: Representation of the hipersphere in dimension i, which is divided into $(i-1)$-dimensional layers.

cess of enumerating lattice points inside a hypersphere, which is presented in Figure 3.1. Algorithmically, this can also be described as a traversal of a tree.

As mentioned before, the sphere decoding consists in finding the shortest vector of the lattice inside a hypersphere, i.e., the closest vector to the origin of the lattice, which also denotes the center of the hypersphere. $\sqrt{C}$ is the norm of the best vector found so far, denoted by $\hat{\mathbf{u}}$. $\sqrt{C}$ also represents the radius of the hypersphere. The implementation of the sphere decoding process is shown in Algorithm 4, where sign(a) returns -1 if $a \leq 0$ and 1 otherwise, and round(a) rounds a to the nearest integer number.

Lattices can be divided into lower-dimensional layers, where the dimension of the layer that is currently being examined, is, from here on, denoted by i. Each i-dimensional layer can be divided into $(i-1)$-dimensional parallel layers, where $1/\mathbf{H}_{i,i}$ represents the distance between the layers.

The algorithm iterates over all the layers. It starts in the n-dimensional layer and stops when this layer is reached again. We will refer to the process of visiting a lower-dimensional layer (decrementing i) as *moving down* and the process of visiting a higher-dimensional layer (incrementing i) as *moving up*.

For each dimension, the coefficients $\mathbf{e}_i$ of the projection onto the basis vectors, where $\mathbf{e}_i = (\mathbf{E}_{i,1}, \mathbf{E}_{i,2}, \ldots, \mathbf{E}_{i,n}) \in \mathbb{R}^n$, and the orthogonal displacement $\mathbf{y}_i$ from the projected vector to the current layer are calculated. The $(i-1)$-dimensional layer that is being examined is determined by $\mathbf{u}_i$ as a coefficient. The $(i-1)$ parallel layers are visited in a zigzag pattern, based on the Schnorr-Euchner refinement (Schnorr and Euchner, 1994). Δ_i contains the step that has to be taken to visit the next $(i-1)$-dimensional layer and is used to update $\mathbf{u}_i$. The squared distance from the origin of the lattice to the layer that is being analyzed is denoted by $\mathbf{dist}_i$. If $\mathbf{dist}_i < C$ the algorithm will *move down*, otherwise it will *move up* again.

When the zero-dimensional layer is reached, the coefficient vector $\mathbf{u}$, is stored in $\hat{\mathbf{u}}$ and the value of C is updated. The algorithm stops when the n-dimensional layer is reached again and returns the vector $\hat{\mathbf{u}}$. The shortest vector of the lattice is determined by transforming the output vector $\hat{\mathbf{u}}$ back to the original coordinate system, i.e., $\mathbf{x} = \hat{\mathbf{u}}\mathbf{B}$, where $\mathbf{x}$ denotes the shortest vector of the lattice.

Conceptually, this algorithm can also be seen as a depth-first traversal on a weighted tree, whose height is n. Each node of the tree represents a layer of the hypersphere, where the child nodes are the $(i-1)$-dimensional layers. The algorithm iterates over all the nodes, until it reaches the root node again. Every time a leaf is reached, C is updated, which reduces the number of nodes that still have to be visited.

Algorithm 4: SE++ sphere decoder with the improvement proposed in (Ghasemmehdi and Agrell, 2009) with some changes to solve the SVP.

```
Input: $n$, H
Output: $\hat{u} \in \mathbb{Z}^n$
$C = \infty$;
$i = n + 1$;
$d_j = n,\ j = 1, \dots, n$;
$dist_{n+1} = 0$;
$E_{n,j} = 0,\ j = 1, \dots, n$;
LOOP
do
    if $i \neq 1$ then
        $//move\ down$
        $i = i - 1$;
        $E_{j-1,i} = E_{j,i} - y_j H_{j,i},\ j = d_i, d_i - 1, \dots, i + 1$;
        $u_i = round(E_{i,i})$;
        $y_i = (E_{i,i} - u_i)/H_{i,i}$;
        $\Delta_i = sign(y_i)$;
        $dist_i = dist_{i+1} + y_i^2$;
    else
        if $dist_i \neq 0$ then
            $//update\ best\ vector$
            $\hat{u} = u$;
            $C = dist_1$;
            $i = i + 1$ ;
        moveToSibling($i, last\_nonzero$, u, $\Delta$, y, E, dist, H);
while $dist_i < C$;
$m = i$;
do
    if $i = n$ then
        return $\hat{u}$;
    else
        $//move\ up$
        $i = i + 1$;
        moveToSibling($i, last\_nonzero$, u, $\Delta$, y, E, dist, H);
while $dist_i \geq C$;
$d_j = i, j = m, m + 1, \dots, i - 1$;
for $j = m - 1, m - 2, \dots, 1$ do
    if $d_j < i$ then
        $d_j = i$;
    else
        break;
goto LOOP
```

```
Function moveToSibling($i, last\_nonzero$, u, $\Delta$, y, E, dist, H)
$u_i = u_i + \Delta_i$;
$\Delta_i = -\Delta_i - sign(\Delta_i)$;
$y_i = (E_{i,i} - u_i)/H_{i,i}$;
$dist_i = dist_{i+1} + y_i^2$;
```

As proposed by Ghasemmehdi and Agrell (Ghasemmehdi and Agrell, 2009), 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 $\mathbf{E}_{i,i}$, 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.

3.3.2 ENUM

ENUM is an enumeration algorithm originally proposed by Schnorr and Euchner (Schnorr and Euchner, 1994) as a subroutine of the BKZ-reduction algorithm. Gama et al. (Gama et al., 2010) proposed, for ENUM, an improvement, similar to the improvement proposed by Ghasemmehdi and Agrell (Ghasemmehdi and Agrell, 2009). The implementation of ENUM is shown in Algorithm 5, where max(a, b) returns a if $a \geq b$ and b, otherwise.

ENUM, as well as SE++, enumerates all lattice points inside a hypersphere. It also examines the layers of the hypersphere in a zigzag pattern and can be mapped onto a depth-first tree traversal. However, there are some differences between these algorithms, regarding the pre-processing, the starting point of the computation and the way how some layers are traversed.

The pre-processing of ENUM also includes the basis reduction (e.g., with BKZ or LLL), but differs in the second part. The Gram-Schmidt orthogonalization is used instead of the Cholesky or the QR decomposition. This orthogonalization generates a lower-triangular matrix μ and a vector that contains the squared norms of the orthogonalization $||\mathbf{b}_1^*||^2, ..., ||\mathbf{b}_n^*||^2$. ENUM receives both as input.

Since the first vector that is found is always the first vector of the basis, the starting point of this algorithm is defined as one of the leaves and not the root of the tree. Although the impact of this improvement is very low, it allows to avoid the computation of the n first nodes. Therefore, C is initialized as $||\mathbf{b}_1^*||^2$ and the variable that contains the current shortest vector as $\hat{\mathbf{u}} = (1, 0, ..., 0)$.

The biggest difference between both algorithms lies in the fact that ENUM avoids the computation of symmetric subtrees by using a variable called $last\_nonzero$. For each vector **v** that can be found in the tree, there is a symmetric vector **-v** with the same norm. Since it is sufficient to find only one of them, the computation of the symmetric subtree can be avoided. This optimization can also be applied to the SE++ algorithm as shown in Section 4.1.2.

3.3.3 Enumeration with Extreme Pruning

In 2010, a new technique, called enumeration with extreme pruning, was proposed (Gama et al., 2010). This technique modifies the ENUM enumeration algorithm to considerably reduce its execution time. Despite of transforming it into an heuristic, the number of nodes of the enumeration tree that has to be visited is considerably reduced, thus reducing the execution time of the enumeration.

The branches of the tree that have to be traversed are bounded by a polynomial bounding function, based on the Gaussian heuristic. This heuristic provides a good estimate for the norm of the shortest vector of a lattice $\mathcal{L}(\mathbf{B})$ and is given by $GH(\mathcal{L}(\mathbf{B})) = \frac{\Gamma(n/2+1)^{1/n}}{\sqrt{\pi}} \times \det(\mathcal{L}(\mathbf{B}))^{1/n}$, where $\Gamma(x)$ is the

Algorithm 5: ENUM algorithm with the improvement proposed in (Gama et al., 2010)

```
Input: $n, \mu, \|b_1^*\|^2, \ldots, \|b_n^*\|^2$
Output: $\hat{u} \in \mathbb{Z}^n$
$C = \|b_1^*\|^2$;
$i = 1$;
$\mathrm{dist}_j = 0,\ j = 1, \ldots, n+1$;
$u_1 = \hat{u}_1 = 1$;
$u_j = \hat{u}_j = 0,\ j = 2, \ldots, n$;
$c_j = 0,\ j = 1, \ldots, n$;
$\Delta_j = 0,\ j = 1, \ldots, n$;
$E \leftarrow (0)_{(n+1)\times n}$;
$d_j = j,\ j = 1, \ldots, n$;
$last\_nonzero = 1$;
LOOP
$\mathrm{dist}_i = \mathrm{dist}_{i+1} + (u_i - c_i)^2 \cdot \|b_i^*\|^2$;
if $\mathrm{dist}_i < C$ then
    if $i \neq 1$ then
        $//move\ down$
        $i = i - 1$;
        $d_{i-1} = \max(d_{i-1}, d_i)$;
        $E_{j,i} = E_{j+1,i} + u_j \mu_{j,i},\ j = d_i, d_i - 1, \ldots, i+1$;
        $c_i = -E_{i+1,i}$;
        $u_i = \mathrm{round}(c_i)$;
        $\Delta_i = 1$;
    else
        $//update\ best\ vector$
        $C = \mathrm{dist}_i$;
        $\hat{u} = u$;
else
    if $i = n$ then
        return $\hat{u}$;
    $//move\ up$
    $i = i + 1$;
    $d_{i-1} = i$;
    if $i \geq last\_nonzero$ then
        $last\_nonzero = i$;
        $u_i = u_i + 1$;
    else
        if $u_i > c_i$ then
            $u_i = u_i - \Delta_i$;
        else
            $u_i = u_i + \Delta_i$;
        $\Delta_i = \Delta_i + 1$;
goto LOOP
```

gamma function, given by $\Gamma(x) = (x-1)!$, and det$(\mathcal{L}(\mathbf{B}))$ is the determinant of the lattice $\mathcal{L}(\mathbf{B})$. The success probability is increased by multiplying the estimated norm by 1.05, as described in (Kuo et al., 2011).

Although the bounding function for dimension 110 in (Kuo et al., 2011) is rounded, we obtained the non-rounded function from Michael Schneider, which is also publicly available in [1] and is presented in Equation 3.1.

$$p(x) = \sum_{i=0}^{8} \mathbf{v}_i x^i, \tag{3.1}$$

where $\mathbf{v} = (9.14465 \times 10^{-4}, 4.00812 \times 10^{-2}, -4.24356 \times 10^{-3}, 2.2931 \times 10^{-4}, -6.91288 \times 10^{-6}, 1.21218 \times 10^{-7}, -1.20165 \times 10^{-9}, 6.20066 \times 10^{-12}, -1.29185 \times 10^{-14})$ and x is the level of the tree for which the bound is being computed. For dimension n, $p(x \cdot 110/n)$ is used. The bounds for each of the levels of the tree are then multiplied by $1.05 \cdot GS(\mathcal{L}(\mathbf{B}))$.

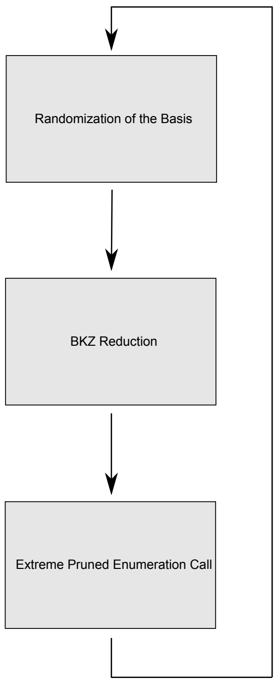


Figure 3.2: Workflow of the enumeration with extreme pruning.

Each enumeration with extreme pruning call has a success probability, i.e., the probability of a single extreme pruned ENUM call to find a short vector, of only 10% for the aforementioned bounding function, based on empirical tests (Kuo et al., 2011). However, the success probability increases when performing multiple calls to the algorithm on different bases of the same lattice. Therefore, 2 steps have to be performed, before each extreme pruned ENUM call: (1) randomize the basis and (2) pre-process the randomized basis. The first step generates a different basis for the same lattice, while the second, first

[1] http://homes.esat.kuleuven.be/ jhermans/gpuenum/

reduces the randomized basis and then computes its Gram-Schmidt orthogonalization. Figure 3.2 shows the workflow of the extreme pruning technique.

Kuo et al. (Kuo et al., 2011) also proved that 44 extreme pruned enumeration calls guarantee a success probability over 99% for the aforementioned bounding function. Therefore, this is the value used from here on for the number of extreme pruned calls.

3.4 The Voronoi Cell-Based Algorithm

The Voronoi cell of a lattice is the set of all points that are closer to the origin than to any other lattice point. The Voronoi relevant vectors are lattice vectors **v** such that $\mathbf{v}/2$ is a facet of the Voronoi cell, which is a convex and symmetric structure that is defined by the intersection of all facets. A Voronoi cell can have at most $2(2^n - 1)$ relevant vectors and, therefore, at most $2(2^n - 1)$ facets.

In 2002, was shown that it is possible to use the Schnorr-Euchner sphere decoding algorithm, described above, to compute the relevant vectors, in (Agrell et al., 2002). The relevant vectors are obtained by computing the closest vector to the midpoints of all sums of one or more basis vectors, i.e., the vectors obtained by $\mathbf{s} = \mathbf{zB}$, where $\mathbf{z} \in \{0, 1/2\}^n \backslash 0$. This generates a set of $2^n - 1$ vectors. The closest vector **v** to each of the midpoints is computed by a call to the SE++ (for the CVP) algorithm. Since the Voronoi cell is symmetric -**v** is also a relevant vector. The vectors computed by SE++ represent the set of relevant vectors. The shortest relevant vector is the solution of the SVP.

Chapter 4

Towards Efficient Implementations

This chapter focuses on the experimental results that were achieved with the sequential and parallel implementations. The tests were performed on a dual-socket cluster node equipped with 2 Sandy Bridge Intel Xeon E5-2670 @ 2.6 GHz processors, with 8 cores each and 2-way simultaneous multi-threading (SMT) technology. The node has a total of 128 GB of RAM. Table 4.1 shows further details of the node. The codes were written in C and compiled with the GNU g++ 4.6.1 compiler, with the -O2 optimization flag. Additionally, the NTL (BKZ basis reduction and for the Gram-Schmidt orthogonalization, except when said otherwise) and Eigen[1] (for the QR decomposition, inverse and transpose matrix computations) libraries were also used. We have used Goldstein-Mayer bases for random lattices, available from the SVP Challenge[2], all of which were generated with the random seed 0. Although the execution times of the programs were fairly stable, each program was executed three times and the best sample was selected.

	Intel Xeon E5-2670
Micro-architecture	Sandy Bridge
Clock Frequency	2.6 GHz
#Sockets	2
#Cores per Socket	8
SMT	2-way
L1 Cache	32 KB I. + 32KB D. per core
L2 Cache	256 KB per core
L3 Cache	20 MB shared

Table 4.1: Specification of the CPU that was used for the tests.

We recall the definition of speedup S_p

$$S_p = \frac{T_1}{T_p}, \tag{4.1}$$

where T_p denotes the execution time of the application with p processors, and efficiency E_p

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p}. \tag{4.2}$$

[1]The Eigen library, a C++ template library for linear algebra; more details at http://eigen.tuxfamily.org/

[2]SVP Challenge, which helps assessing the strength of SVP algorithms, and serves to compare different types of algorithms; more details at http://www.latticechallenge.org/svp-challenge/

4.1 Sequential Implementations

4.1.1 Enumeration and Voronoi cell-based algorithms

To solve the SVP, we implemented the SVP-solver proposed in (Ghasemmehdi and Agrell, 2009), SE++, ENUM, improved in (Gama et al., 2010), and the Voronoi cell-based algorithm also proposed in (Agrell et al., 2002). The bases were reduced with BKZ with block-size 20. This block-size was chosen because its execution time is very low, compared to the total execution time, and computes a basis of good quality. We omitted the execution time of the basis reduction in the presented results since it is a very common practice in lattice basis cryptography, due to its negligible execution time when compared to the execution time of the SVP-solvers. The comparison of the execution times of these implementations is shown in Section 4.1.4.

Additionally, the SE++ algorithm was optimized to improve its data locality and reduce redundant operations. Both the columns and rows, of matrices **E** and **H** are accessed in a reverse order (i.e., from the bottom to the top and from right to left), which considerably hurts the data locality. Measurements of the computation of matrix **E** (line 10 of Algorithm 4) have shown that up to 25% of the total execution time is spent on this operation. Therefore, matrix **H** was transposed and reflected to allow accesses to both matrices from the top to the bottom and from left to right. The reflection is computed by a swap$(\mathbf{H}_{i,j}, \mathbf{H}_{i,j-n}), \forall 1 \leq i \leq n, 1 \leq j < n/2$ operation.

Since matrix **H** is not changed during the execution of the algorithm, the value $1/\mathbf{H}$ (e.g., in line 12 of Algorithm 4) is also never changed. Therefore, this computation can be done before entering the cycle and its results sent as input to the algorithm. This optimization can also improve the usage of pipelining, since division operations cannot be pipelined.

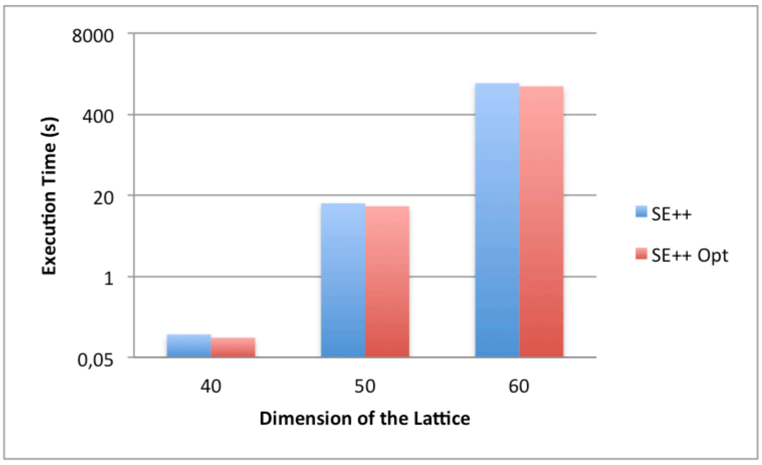


Figure 4.1: Comparison of SE++ with the optimized version of the algorithm.

Figure 4.1 shows the comparison between SE++ and the optimized SE++ algorithm (SE++ Opt). With these optimizations the execution time of SE++ is decreased by a factor of 11%. Despite of this improvement of the performance of SE++, we opted to compare the different algorithms (SE, SE++, ENUM and the Voronoi cell-based algorithm) without any optimization to avoid comparing versions of the algorithms with

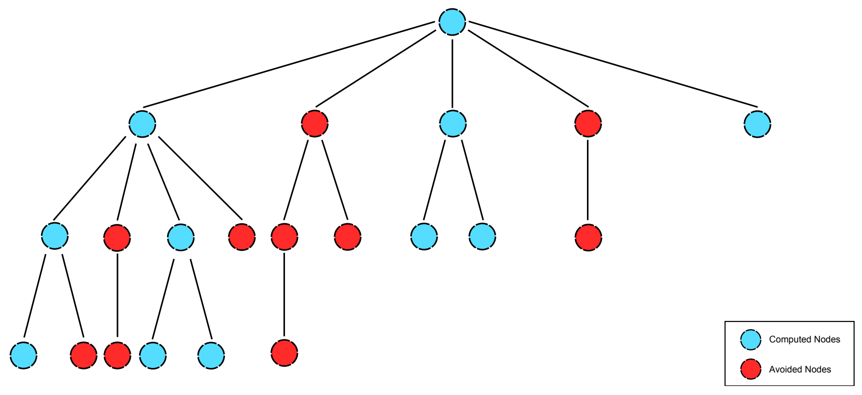


Figure 4.2: Representation of the symmetric subtrees whose computation can be avoided.

different levels of optimization, since it was not possible to optimize the others due to time restrictions.

On the Voronoi cell-based algorithm we implemented, instead of storing the whole set of relevant vectors as described in (Agrell et al., 2002), we only store the shortest vector found at any given moment. Therefore, after each CVP call, the computed vector is compared to the best vector that was already found and updates it, if it is better. This optimization was used on the comparison with the other algorithms, since the amounts of memory used would be huge even for lattices in low dimensions.

4.1.2 Improved SE++

The SE++ algorithm, presented in (Agrell et al., 2002) and improved in (Ghasemmehdi and Agrell, 2009), computes 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. Figure 4.2 shows a representation of the symmetric subtrees that can be avoided.

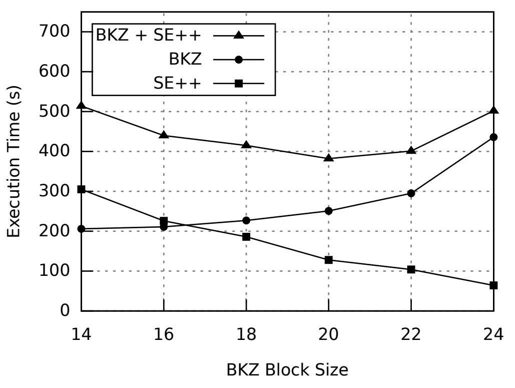
The ENUM algorithm avoids these computations, by using a variable, called $last\_nonzero$, which stores the largest index i of the coefficient vector **u** for which $\mathbf{u}_i \neq 0$. For example, if $\mathbf{u}_i = 0$, but its parent $\mathbf{u}_{i+1} \neq 0$, then all its subtrees have to be computed. On the other hand, there are only symmetric subtrees on nodes where $\mathbf{u}_j = 0, j = i, ..., n$. As shown in Figure 4.2, there are only subtrees whose computation can be avoided on the left-most 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 the same strategy can be applied to SE++ exactly the same way. From here on, we will call improved SE++

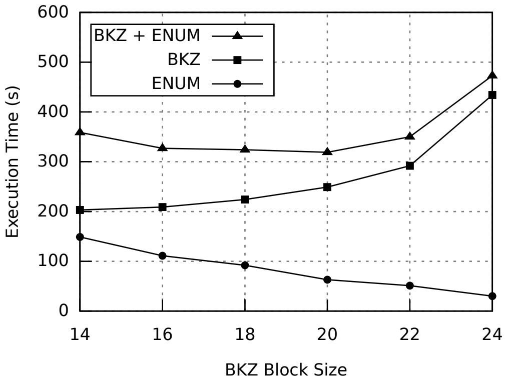
Algorithm 6: Improved SE++ algorithm

```
Input: $n, H$
Output: $\hat{u} \in \mathbb{Z}^n$
2  $C = \infty$;
3  $i = n + 1$;
4  $d_j = n, \ j = 1, \ldots, n$;
5  $dist_{n+1} = 0$;
6  $E_{n,j} = 0, \ j = 1, \ldots, n$;
7  $last\_nonzero = 1$;
8  LOOP
   do
9      if $i \neq 1$ then
10         $//move\ down$
           $i = i - 1$;
11         $E_{j-1,i} = E_{j,i} - y_j H_{j,i}, \ j = d_i, d_i - 1, \ldots, i + 1$;
12         $u_i = round(E_{i,i})$;
13         $y_i = (E_{i,i} - u_i)/H_{i,i}$;
14         $\Delta_i = sign(y_i)$;
15         $dist_i = dist_{i+1} + y_i^2$;
16     else
17         $//update\ best\ vector$
           if $dist_i \neq 0$ then
18             $\hat{u} = u$;
19             $C = dist_1$;
20             $i = i + 1$;
21         moveToSibling($i, last\_nonzero$, $u, \Delta, y, E, dist, H$);
22 while $dist_i < C$;
23 $m = i$;
24 do
25     if $i = n$ then
26         return $\hat{u}$;
27     else
28         $//move\ up$
           $i = i + 1$;
29         moveToSibling($i, last\_nonzero$, $u, \Delta, y, E, dist, H$);
30 while $dist_i \geq C$;
31 $d_j = i, j = m, m + 1, \ldots, i - 1$;
32 for $j = m - 1, m - 2, \ldots, 1$ do
33     if $d_j < i$ then
34         $d_j = i$;
35     else
36         break;
37 goto LOOP
```

```
Function moveToSibling($i, last\_nonzero$, $u, \Delta, y, E, dist, H$)
if $i \geq last\_nonzero$ then
    $last\_nonzero = i$;
    $u_i = u_i + 1$;
else
    $u_i = u_i + \Delta_i$;
    $\Delta_i = -\Delta_i - sign(\Delta_i)$;
$y_i = (E_{i,i} - u_i)/H_{i,i}$;
$dist_i = dist_{i+1} + y_i^2$;
```



(a) SE++



(b) ENUM

Figure 4.3: Execution time BKZ call, the extreme pruned enumeration call and the total execution time of our parallel implementation of both algorithms with extreme pruning running with 32 (+1) processes for a lattice in dimension 80.

to the SE++ algorithm with the optimization that avoids the computation of symmetric branches.

4.1.3 Enumeration with Extreme Pruning

The extreme pruning technique was originally applied to the ENUM enumeration algorithm. However, we show that it is possible to apply this technique to other enumeration algorithms, namely the SE++, and that it can significantly increase its performance.

Since the execution time of the lattice basis reduction takes a significant portion of the overall execution, in this case, it is included in the results. The execution time of the enumeration with extreme pruning is a compromise between the running time of the BKZ reduction and the running time of the enumeration with extreme pruning. When increasing the block-size of the BKZ reduction, the execution time of BKZ increases, but the execution time of the extreme pruned enumeration call decreases. Therefore, it is necessary to find a balance between these execution times to minimize the total execution time decreases. Figures 4.3(a) and 4.3(b) show that, for a lattice in dimension 80, the total execution time reaches its minimum for the block-size 20, for both algorithms. Therefore, from here on, the results presented for the implementations with extreme pruning use the block-size 20. Since, the purpose of this test is only to find the ideal block-size, we performed the tests with our parallel implementation running with 32 (+1 master) processes, which is described in Section 4.2.2.

4.1.4 Comparative Performance Evaluation

Figure 4.4 compares the execution times of all 6 implementations: the Voronoi cell-based algorithm (VC), SE++, ENUM, improved SE++, SE++ with extreme pruning and ENUM with extreme pruning. All implementations were BKZ-reduced with block-size 20. While for the first four algorithms the execution of the BKZ is not significant and was not included in the presented results, for the extreme pruned enumeration algorithms, it is a crucial part of the algorithm and, therefore, its execution time was included in the results.

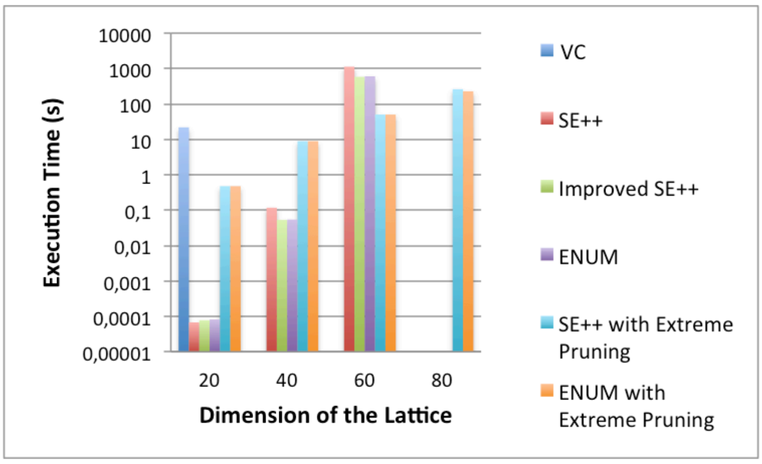


Figure 4.4: Performance of different SVP-solver running in sequential.

The execution time of each algorithm increases with the dimension of the lattice. The Voronoi cell-based algorithm performs significantly worse than the others and becomes impractical for dimensions 40 and above, since the number of CVP calls is exponential ($2^n - 1$). ENUM performs better than SE++, with and without extreme pruning. However, the optimization that avoids symmetric branches improved the performance of SE++ by a factor of almost 50%, slightly outperforming ENUM by a factor of 3%.

On the other hand, the performance of enumeration algorithms is significantly improved by the extreme pruning technique for a lattice in dimension 60. Since the chosen block-size is optimal for the lattice in dimension 80, the performance of solving the SVP on a lattice in dimension 60 could still be improved. The enumeration with extreme pruning technique also allows to solve the lattice in dimension 80, which would be impractical for the enumeration without extreme pruning. For lattices in dimensions 20 and 40, the extreme pruned enumeration performed worse than the implementations without extreme pruning, because the block-size is not optimal in these cases, thus taking much longer than with an optimal block-size. For these dimensions, the execution times of both algorithms become almost identical, since almost all the time is spent on the BKZ reduction.

4.1.5 A New BKZ Reduction Approach

An approach was studied in collaboration with Thomas Arnreich (Arnreich, 2014), which changes the workflow of the algorithm similarly to the implementation of Liu et al., from here on called AC_BKZ. It consists in computing the shortest vector on multiple blocks simultaneously. Instead of calling ENUM on one block and performing an LLL reduction afterwards, this approach performs the ENUM calls on all blocks of one round first and only in the end performs the LLL reduction. The vectors found by ENUM are inserted in the position right before the respective block. However, the ENUM algorithm is not called over all blocks. When the sliding window reaches the bottom of the basis, i.e, when $j > n-\beta+1$ and $k = n$, the shorter vectors that can be possibly found by ENUM are always greater or equal than the shorter vector that can be possibly found in the window $j = n-\beta+1$ and $k = n$. This happens because the difference between the consecutive windows for $j > n - \beta + 1$ is one additional vector that is not analyzed, which leads to having one less vector to perform linear combinations. Therefore, the norm of the shorter vectors that can be possibly found by ENUM is greater or equal to the shorter vector that can be possibly found in the window $j = n - \beta + 1$. The vectors that are shorter than the first vector of the respective block are inserted in the position right before the beginning of the block. An LLL reduction is performed over the basis at the end. This process is repeated until the enumeration algorithm no longer finds shorter vectors.

4.1.6 Basis Quality Assessment

Although it is crucial to have efficient lattice reduction implementations, it is even more relevant to guarantee that the computed bases have a good quality, since this can significantly speedup the performance of other algorithms (e.g., SVP-solvers). To measure the quality of the bases that the algorithms output, we assessed the following criteria, all important to the quality of a basis:

1. The Hermite factor of the basis.
2. The norm of the shortest vector of the basis.
3. The average of the norms of the basis vectors.
4. The product of the norms of the basis vectors.
5. The norm of the last Gram-Schmidt vector.
6. The execution time of an SVP-solver.

The quality of the basis is better if all the mentioned criteria, except the fourth, are smaller. The Hermite factor can be defined as

$$HF = \frac{\|\mathbf{b}_1\|}{vol(\mathcal{L}(\mathbf{B}))} = \frac{\|\mathbf{b}_1\|}{det(\mathcal{L}(\mathbf{B}))^{1/n}} \tag{4.3}$$

where $vol(\mathcal{L}(\mathbf{B}))$ is the volume of the lattice. Since the lattices are unimodular, the volume of the lattices, used for normalization, is always 1 and, therefore, the Hermit Factor is equal to the norm of the first basis vector, which for a BKZ-reduced basis is also the shortest vector of the basis.

Each of these criteria was analyzed on the three sequential implementations, Orig_BKZ, AC_BKZ and G_BKZ_FP, for lattices in dimensions 60, reduced with block-sizes 30, 35 and 40, and 70 and 80, reduced with block-size 40. The Orig_BKZ implementation is the original BKZ as described in Section 3.1.2. The AC_BKZ implementation was described in Section 4.1.5. The G_BKZ_FP implementation is publicly available at NTL and is the implementation, provided by this library, that produces the bases with best quality and with the shortest execution time (*G* stands for Givens rotations, used internally for the Gram-Schmidt orthogonalization, and *FP* stands for double precision). The fplll[3] library was not used in the comparison

[3]Floating point LLL library, a library that contains several algorithms on lattices that rely on floating point computations; more details at http://perso.ens-lyon.fr/damien.stehle/fplll/fplll-doc.html

tests because it uses a different LLL implementation, which, despite of being faster, also outputs a basis of lower quality.

The results are presented in Table 4.2 and 4.3. The selected SVP-solver for the sixth criteria was ENUM, since the class of algorithms it belongs to is the most dependent on the quality of the basis. However, this is also the reason why we only performed this test on the lattice in dimension 60, since, on the presented lattices, ENUM only runs in reasonable times for this dimension.

	Criteria	Block-size		
		30	35	40
Orig_BKZ	1. & 2.	1943	1943	1943
	3.	2656	2576	2653
	4.	1.53×10^{205}	2.65×10^{204}	1.35×10^{205}
	5.	444.2	443.8	487.8
	6.	192.4s	198.5s	239.1s
AC_BKZ	1. & 2.	1943	1956	1943
	3.	2638	2607	2636
	4.	9.63×10^{204}	5.28×10^{204}	9.10×10^{204}
	5.	529,8	485.1	437.9
	6.	209.6s	229.1s	241.4s
G_BKZ_FP	1. & 2.	2116	1943	1943
	3.	2658	2590	2575
	4.	2.19×10^{205}	4.38×10^{204}	3.09×10^{204}
	5.	446.9	485.6	454.2
	6.	266.3s	228.0s	253.0

Table 4.2: Quality results for the lattice in dimension 60, reduced with block-sizes 30, 35 and 40, for the different criteria.

Dimension	Criteria	Orig_BKZ	AC_BKZ	G_BKZ_FP
70	1. & 2.	2143	2294	2251
	3.	2851	2971	2917
	4.	4.68×10^{241}	8.31×10^{242}	2.40×10^{242}
	5.	432.0	393.9	401.6
80	1. & 2.	2432	2395	2511
	3.	3258	3244	3244
	4.	6.28×10^{280}	4.54×10^{280}	3.17×10^{280}
	5.	372.6	319.6	374.9

Table 4.3: Quality results for the lattices in dimensions 70 and 80, reduced with block-size 40, for the different criteria.

In the general case, the Orig_BKZ implementation, showed to produce bases with better quality. The shortest vector found by this implementation was always shorter or equal to the remaining implementations, except for the lattice in dimension 80. Additionally, ENUM (sixth criteria) performs better on lattices reduced by Orig_BKZ. The third, fourth and fifth criteria are not very conclusive, since the values are very close to each other. On the other hand, the AC_BKZ seems unstable, since the quality of the basis widely varies when comparing it with the other implementations.

4.1.7 BKZ Performance Analysis

The lattices provided by the SVP Challenge require high precision data structures, since their bases, before reduction, are composed by large numbers, which do not fit into the primitive data types provided by the C programming language. This problem was overcome by using the NTL library to read the input files and to perform a first LLL reduction. With this reduction, the values of the coordinates of the vectors become small enough to fit into an int type matrix.

Since the BKZ algorithm performs multiple LLL reductions internally, the LLL algorithm and the Gram-Schmidt orthogonalization had also to be implemented to achieve an efficient implementation of BKZ. For the internal LLL calls, NTL's LLL had to be replaced because it only outputs the reduced matrix and not the updated Gram-Schmidt orthogonalization, which means that it would be necessary to compute it twice.

For the LLL calls in BKZ, we implemented the LLL with deep insertions with exact arithmetic, as described in Section 5.1. The LLL with deep insertions is also used in NTL's implementation of BKZ, although it is implemented with floating point arithmetic. The classical Gram-Schmidt orthogonalization was implemented, since it is used by LLL. To prevent precision losses, which occurred with float precision, double precision was used.

Three sequential BKZ implementations were compared, namely Orig_BKZ, AC_BKZ and G_BKZ_FP, the fastest implementation provided by the NTL library. The first two implementations used NTL's LLL_XD for the first LLL reduction before calling the BKZ reduction and the developed LLL implementation for the remaining.

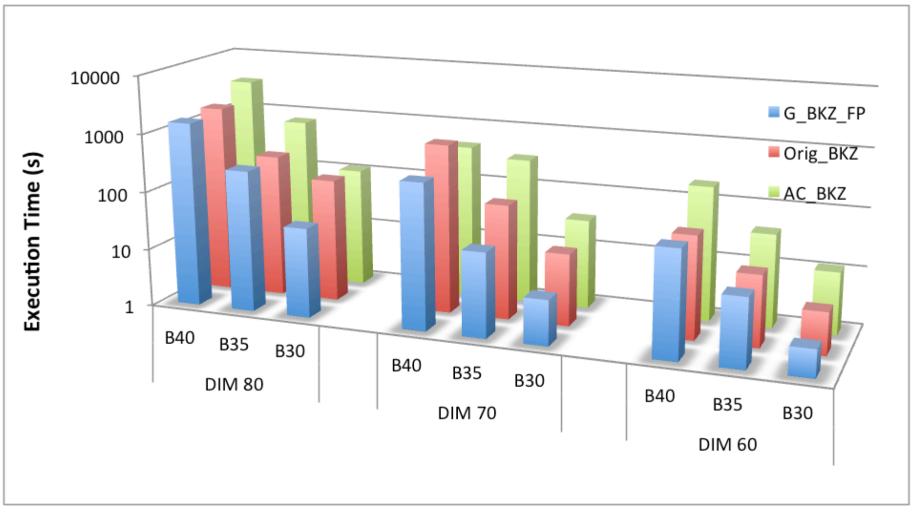


Figure 4.5: Execution time of the sequential versions of three BKZ implementations for lattices in dimensions 60, 70 and 80, reduced with block-sizes 30, 35 and 40.

We ran the three implementations with lattices in dimensions 60, 70 and 80 with block-sizes 30, 35 and 40. Figure 4.5 shows their execution times for the mentioned dimensions and block-sizes. In all implementations, the execution time increases with the block-size and with the lattice dimension. By changing the workflow of the algorithm, the execution time of the AC_BKZ almost doubles in comparison to the original algorithm. Since LLL reductions are not performed on each iteration, the algorithm takes much longer to converge. The two algorithms that performed the best were the Orig_BKZ implementation and

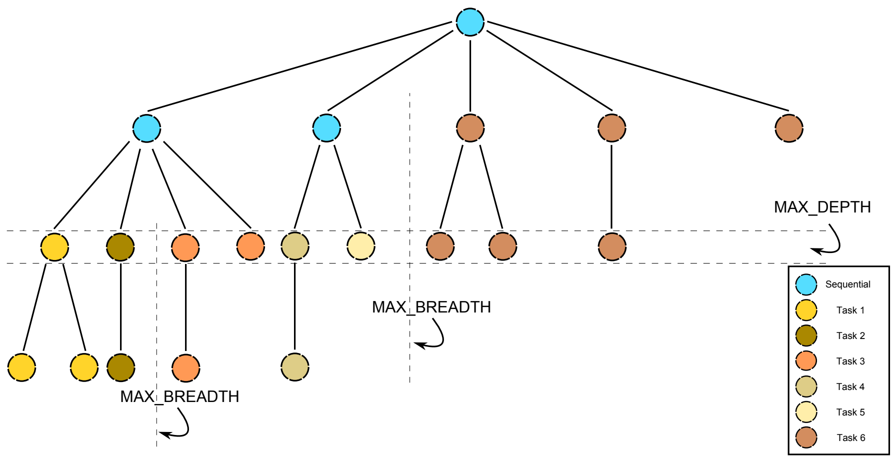


Figure 4.6: Map of the parallel enumeration workflow on a tree, partitioned into tasks, according to the parameters. In this example the parameters were set as MAX_BREADTH = 3 and MAX_DEPTH $= n - \log_2(4Threads)$.

the G_BKZ_FP from NTL. In some instances, the Orig_BKZ performed better (e.g., the lattice in dimension 60, reduced with block-size 40), while in others G_BKZ_FP was faster (e.g., the lattice in dimension 62, reduced with block-size 40).

4.2 Parallel Implementations

4.2.1 ENUM, SE++ and Improved SE++

As previously mentioned, the workflow of the algorithms can be naturally mapped onto a tree traversal, where different branches can be computed in parallel, by different threads. Synchronization is seldom, since it only updates the best vector found at any given instant (the coefficients of the shortest vector, at a given moment), as explained below. The implementations of the parallel approach of ENUM, SE++ and the improved SE++ were written in C, and create OpenMP tasks to distribute work among threads. Once tasks are created, they are added to a queue of tasks, and are scheduled by the OpenMP runtime system among the running threads. This system also defines the order of execution of the created tasks, in runtime. Figure 4.6 exemplifies the workflow of the parallel enumeration for the parameters MAX_BREADTH = 3 and MAX_DEPTH $= n - \log_2(4Threads)$.

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. However, when creating the task number MAX_BREADTH, i.e. $|\Delta_i|$ = 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. Once tasks are created, they are (either promptly or after some time) assigned to one of the threads within the team, by the OpenMP runtime system. As there is an implicit barrier at the end of the single region, all the created tasks will be, at that point, entirely processed.

The tree is considerably unbalanced. $|\Delta_i|$ 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 the $(i-1)$-dimensional layer that is being analyzed and the projected vector. Therefore, $|\Delta_i|$ is used to identify heavier and lighter subtrees: the lower $|\Delta_i|$ is, the heavier tree. As mentioned, MAX_BREADTH determines the value of $|\Delta_i|$ from which subtrees are grouped together, thus preventing the creation of too fine-grained tasks. We set MAX_BREADTH = 6, based on empirical tests described in Section 4.2.3.

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 dimension 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 $|\Delta_i|$ is also sent to the task, allowing to distinguish subtrees that were grouped together from single subtrees.

At the beginning of the execution of the task, the variable $last\_non\_zero$, based on the already computed values of **u** given by **u_Aux**, is also recomputed to identify if the subtree assigned to the task contains symmetric branches. It is not sufficient to initialize it with the value 1, because the left-most subtree is not computed by every thread and, therefore, some threads would identify symmetric subtrees where they are not existent. Since $last\_nonzero$ is the largest value of i for which $\mathbf{u}_i \neq 0$, this has to be checked at the beginning of the execution of each task. If all the coordinates of **u_Aux** are 0, $last\_nonzero$ is initialized to 1, as in the sequential implementation.

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**, **dist**, Δ and **d** (matrix **E** and vectors **u**, **c**, **dist**, Δ and **d**, for ENUM), with the create_node() function, 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 as

$\hat{\mathbf{u}} = (1, 0, \ldots, 0)$. 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 if $\mathbf{dist}_1 < C$, since other threads might update those values in the meantime.

Additionally, the starting point of ENUM was altered to the root, instead of one of the leaves. This allowed to apply the same parallelization strategy to ENUM.

4.2.2 ENUM/SE++ with Extreme Pruning

Both algorithms with extreme pruning, SE++ and ENUM, were implemented in C, using MPI to distribute work among multiple processes. OpenMP was not used, since NTL, the library used to perform the basis reduction and for ENUM also the Gram-Schmidt orthogonalization, is not thread safe.

As aforementioned, to guarantee a high success probability, it is necessary to perform multiple extreme pruned enumeration calls. Each call requires a randomization of the basis, the pre-processing of the randomized basis and and extreme pruned enumeration call. Each of these iterations (randomization, pre-processing and the extreme pruned enumeration call) does not depend on one another. Therefore, it is possible to distribute them among different processes without any synchronization.

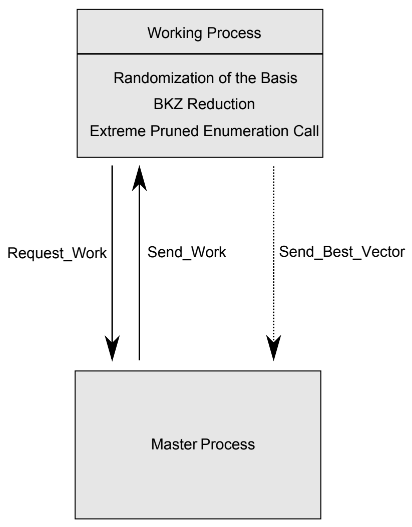
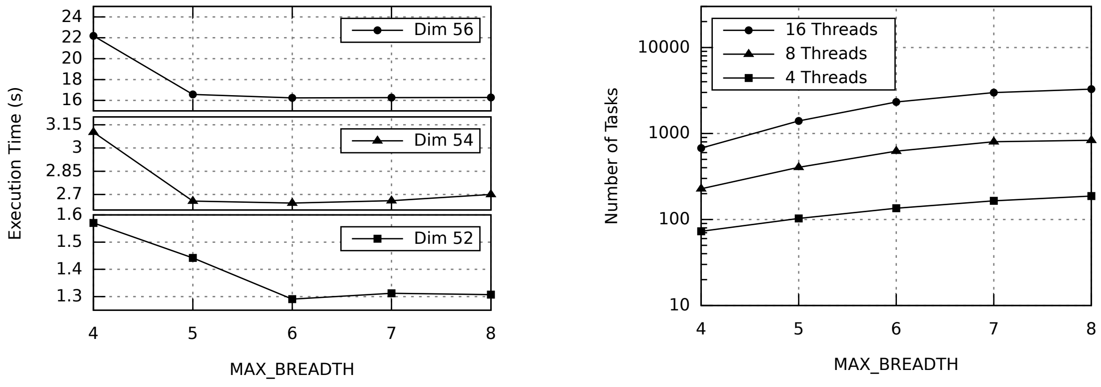


Figure 4.7: Workflow of the parallel enumeration with extreme pruning.

Since the reduction and the extreme pruned enumeration depend on the quality of the basis, the execution time of each iteration will also differ. Therefore, an additional process was used to distribute the iteration among processes and to collect the best vectors found at the end of each iteration. This allows a better work distribution, since each process only requests an additional iteration when the previous is finished. Therefore, each process only needs to request the number of the iteration that it will compute, which is used as random seed to generate the randomized basis. Figure 4.7 shows the workflow of the parallel enumeration with extreme pruning.

4.2.3 Comparative Performance Evaluation



(a) Execution time

(b) Number of tasks

Figure 4.8: Performance of our improved SE++ implementation with 16 threads solving the CVP on random lattices in dimensions 52, 54 and 56, in (a), and number of tasks created for 4, 8, and 16 threads for a lattice in dimension 56, in (b). BKZ-reduced bases with block-size 20.

As mentioned before, our parallel implementations of ENUM, SE++ and the improved SE++ use two parameters 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 simplicity, Figures 4.8(a) and 4.8(b) show the results of only some of them. Figure 4.8(a) shows the execution time of our improved SE++ implementation, for different values of MAX_BREADTH, when running with 16 threads (for other numbers of threads the results were very similar). The variation of the execution time and the number of tasks where the same for the other algorithms, which leads to the same conclusions. Figure 4.8(b) shows the number of tasks that are created, for 4, 8 and 16 threads. The results showed that the higher the value of MAX_BREADTH the higher the number of tasks that are created. Performance-wise, MAX_BREADTH delivered the best results when set to 6. The number of created tasks represents a good compromise between the size and number of tasks, which results in good load balance. 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 guarantee 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 to these values, an ideal trade-off between load balancing and granularity of the tasks is guaranteed.

The parallel versions of SE++, ENUM and the improved SE++ were tested for 1 to 32 threads. Figure 4.9 shows the performance of the implementations. All three implementations scale linearly for up to 8 threads and almost linearly for 16 threads. The implementations also benefit from SMT.

Tables 4.4 ,4.5 and 4.6 show the speedup and efficiency of all trials, for SE++, ENUM and the improved SE++, respectively. Since the compute node has 16 physical cores, the row with 32 threads concerns

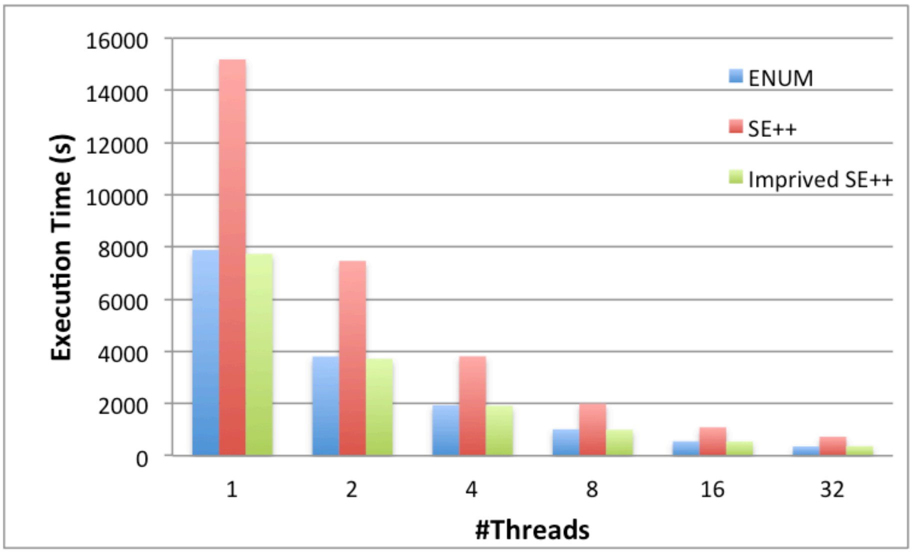


Figure 4.9: Performance of the SE++, ENUM and improved SE++ parallel implementations on a random lattice in dimensions 62. BKZ-reduced bases with block-size 20.

	Dimension of the lattice											
	52		54		56		58		60		62	
Threads	S	E	S	E	S	E	S	E	S	E	S	E
2	1.97x	98%	1.99x	100%	2.19x	109%	2.05x	103%	1.74x	87%	2.03x	102%
4	3.86x	97%	4.00x	100%	4.36x	109%	3.98x	99%	3.32x	83%	3.99x	100%
8	7.18x	90%	7.65x	96%	8.61x	108%	7.35x	92%	5.30x	66%	7.61x	95%
16	13.79x	86%	14.36x	90%	15.53x	97%	14.01x	88%	12.53x	78%	13.98x	87%
32	21.05x	66%	22.01x	69%	22.77x	71%	22.06x	69%	21.01x	66%	20.92x	65%

Table 4.4: Speedup (S) and efficiency (E) of SE++ for 6 lattices, whose bases were reduced with BKZ with block-size 20.

a special case: the use of SMT. The parallel version might possibly have a smaller workload than the sequential execution, since some threads might find, at any given point, vectors that are strictly shorter than those that would be analyzed in a sequential execution. This justifies the super-linear speedups that are achieved for some cases, such as for the lattice in dimension 56 for 1 to 8 threads.

For the remaining cases, efficiency levels of $>$90% are attained for the majority of the instances of up to 8 threads, except for the lattice in dimension 60, presumably due to the quality of the basis. With 16 threads, lower efficiency levels are attained than for up to 8 threads, 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. The lattice in dimension 56 was also a special case, since it reached linear speedups for up to 16 threads. This is explained by the high reduction of workload for this lattice, as we can see on the high efficiency values for 2 to 8 threads.

The scalability of the algorithms is almost the same. This shows that the same parallelization technique can be applied to different enumeration algorithms without influencing the scalability. Additionally, with the improvement that avoids the computation of symmetric branches, SE++ becomes a competitive SVP-solver.

	Dimension of the lattice											
	52		54		56		58		60		62	
Threads	S	E	S	E	S	E	S	E	S	E	S	E
2	1.99x	99%	1.94x	97%	2.20x	110%	2.05x	102%	1.74x	87%	2.07x	104%
4	3.93x	98%	3.97x	99%	4.56x	114%	4.11x	103%	3.07x	77%	4.07x	102%
8	7.05x	88%	7.72x	97%	8.39x	105%	7.75x	97%	4.84x	60%	7.78x	97%
16	13.47x	84%	14.29x	89%	15.62x	98%	13.93x	87%	11.17x	70%	14.18x	89%
32	20.11x	63%	22.24x	70%	24.20x	76%	22.72x	71%	21.83x	68%	22.03x	69%

Table 4.5: Speedup (S) and efficiency (E) of ENUM for 6 lattices, whose bases were reduced with BKZ with block-size 20.

	Dimension of the lattice											
	52		54		56		58		60		62	
Threads	S	E	S	E	S	E	S	E	S	E	S	E
2	1.97x	98%	2.01x	101%	2.19x	110%	2.04x	102%	1.70x	85%	2.08x	104%
4	3.81x	95%	3.90x	97%	4.53x	113%	4.09x	102%	3.01x	75%	4.04x	109%
8	6.93x	87%	7.61x	95%	8.36x	104%	7.71x	96%	4.75x	59%	7.75x	97%
16	13.17x	82%	13.89x	87%	15.52x	97%	13.87x	87%	10.98x	69%	14.16	89%
32	19.24x	60%	20.84x	65%	23.35x	73%	21.84x	69%	20.48x	64%	20.92x	65%

Table 4.6: Speedup (S) and efficiency (E) of the improved SE++ for 6 lattices, whose bases were reduced with BKZ with block-size 20.

We tested the SE++ and ENUM implementations with extreme pruning for lattices in dimensions 60, 70 and 80. In Section 4.1.3 we showed that the block-size 20 is ideal for a lattice in dimension 80 and, therefore, we will use $\beta = 20$, from here on. In this case, the execution time of the pre-processing is also included in the presented running times.

The parallel versions of both implementations were tested for 1-32 processes that do work and one additional process that only sends work to the other processes and receives the best vectors that were found at the end. Since the execution time of the latter is negligible when comparing with the total execution time, we omitted it in the presented results, i.e., the results show only the number of processes that effectively do work.

Tables 4.7 and 4.8 show the speedups and efficiency levels of our implementations. For up to 4 processes we have $> 90\%$ efficiency levels and almost 90% for up to 8 processes. For higher number of processes the scalability is hurt by the load imbalance. For 8 and 16 processes the scalability is hurt by the load imbalance. The reason for this is that the execution time of BKZ, which is used on the basis pre-processing and which uses an enumeration algorithm as sub-routine, and the enumeration with extreme pruning call strongly depend on the quality of the basis. Despite of being always the same lattice that is analyzed, the randomization of the basis generates a different basis for the same lattice, which leads to different execution times of BKZ and the extreme pruned enumeration call. These implementations also profit from SMT, since they keep scaling for 32 processes.

	Lattice Dimension											
	70		72		74		76		78		80	
Processes	S	E	S	E	S	E	S	E	S	E	S	E
2	1.99.x	99%	1.96x	98%	1.99x	100%	1.98x	99%	1.99x	99%	1.99x	99%
4	3.89x	97%	3.78x	95%	3.79x	95%	3.76x	94%	3.76x	94%	3.82x	96%
8	7.06x	88%	7.00x	87%	7.07x	88%	7.19x	90%	7.12x	89%	7.11x	89%
16	12.42x	78%	12.63x	79%	12.89x	81%	12.18x	76%	12.70x	79%	12.73x	80%
32	16.39x	51%	16.57x	52%	17.02x	53%	15.66x	49%	16.46x	51%	17.09x	53%

Table 4.7: Speedup (S) and Efficiency (E) of the parallel SE++ algorithm with extreme pruning for 6 lattices, whose bases were reduced with BKZ with block-size 20.

	Lattice Dimension											
	70		72		74		76		78		80	
Processes	S	E	S	E	S	E	S	E	S	E	S	E
2	1.99x	99%	1.96x	98%	2.00x	100%	1.98x	99%	1.98x	99%	1.99x	99%
4	3.85x	96%	3.81x	95%	3.78x	95%	3.75x	94%	3.81x	95%	3.89x	97%
8	7.07x	88%	6.97x	87%	7.05x	88%	7.18x	90%	7.15x	89%	7.03x	88%
16	12.50x	78%	12.72x	79%	12.96x	81%	12.10x	76%	12.78x	80%	12.71x	79%
32	16.27x	51%	16.54x	52%	17.04x	53%	15.98x	50%	16.56x	52%	15.88x	50%

Table 4.8: Speedup (S) and Efficiency (E) of the parallel ENUM algorithm with extreme pruning for 6 lattices, whose bases were reduced with BKZ with block-size 20.

4.2.4 Further Improvements using BKZ

The execution time of the BKZ algorithm and the quality of the output basis strongly depend on the block-size. For high dimensional block-sizes, the ENUM call takes almost as much time as the total execution time of the BKZ reduction. The reason for this is the exponential complexity of the ENUM algorithm, while the LLL algorithm runs only in polynomial time. Therefore, to efficiently parallelize the algorithm, it is sufficient to parallelize ENUM. This is done by replacing the ENUM call by the parallel ENUM call, described in Section 4.2.1. Since this approach used the implementation of the parallel ENUM, it used OpenMP to spawn threads and distribute work among them. It aims to explore parallelism without changing the workflow of BKZ on a more fine grained level. This implementation will be, from here on, called ENUM_BKZ.

The AC_BKZ implementation was parallelized with OpenMP. It consisted in calling multiple sequential ENUM calls of the same round, in parallel. At the end, the vectors found by ENUM are inserted into the lattice and an LLL-reduction is performed over this new set of vectors sequentially. This approach allows to parallelize BKZ at a more coarse grained level by changing its workflow.

Scalability tests were performed for both implementations, ENUM_BKZ and AC_BKZ. The implementations were tested for lattices in dimensions 60 and 70 for block-sizes 40, 45 and 50. Each test was executed three times and the best sample was selected. Unfortunately, we did not have access to the implementation of Liu et al. to compare it to our implementations. Figures 4.10 and 4.11 show the execution time of our two parallel implementations for 1-32 threads.

The ENUM_BKZ implementation scales linearly for up to 8 threads and almost linearly for 16 threads for block-size 50 for both tested lattices. In fact, the ENUM calls are executed over a basis whose dimension

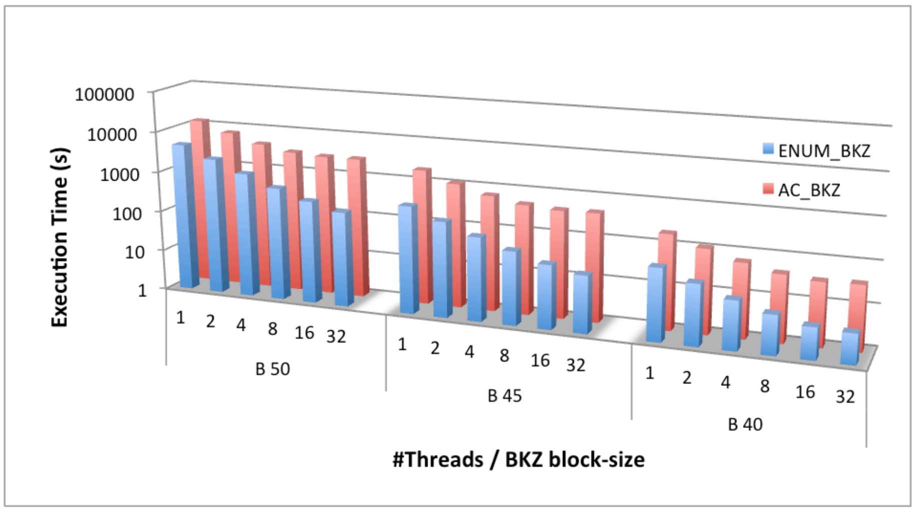


Figure 4.10: Performance of both parallel versions of BKZ for block-sizes 40, 45 and 50 for a lattice in dimension 60.

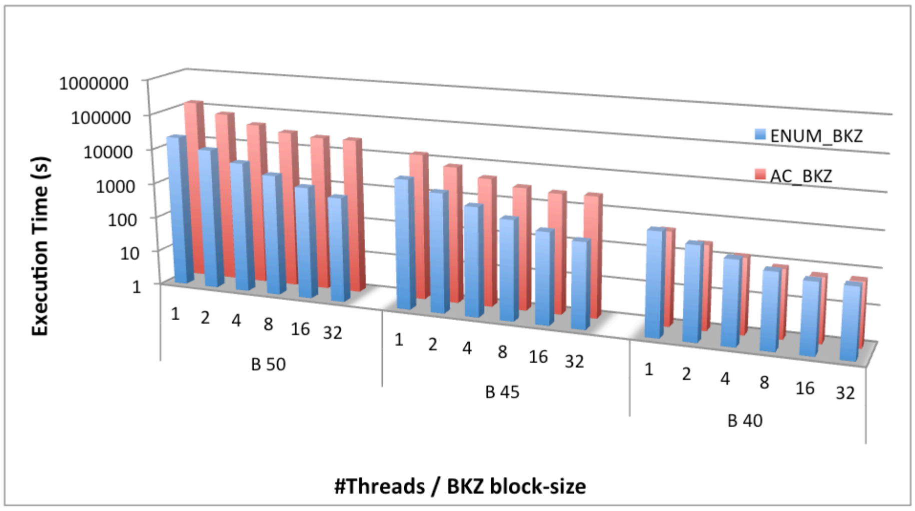


Figure 4.11: Performance of both parallel versions of BKZ for block-sizes 40, 45 and 50 for a lattice in dimension 70.

is the size of the corresponding sliding window. This explains why the scalability for the presented block-sizes are very similar to results of the parallel ENUM presented in Section 4.2.3. This implementation also benefits from the SMT technology, since it keeps scaling for 32 threads. On the other hand, the AC_BKZ implementation does not scale linearly. Since the ENUM calls over different blocks have different execution times, load imbalance occurs, which hurts the scalability. This implementation does not benefit from SMT, because the load imbalance increases for 32 threads, which hinders it from scaling more.

Tables 4.9 and 4.10 show the speedup and efficiency of both implementations. For the ENUM_BKZ implementation, efficiency levels of $> 90\%$ were attained for the majority of the instances of up to 8 threads, except for block-sizes of 40, where the workload of each enumeration call is too small to allow it to scale more. With 16 threads, the efficiency decreases, presumably due to the NUMA organization of the RAM. The scalability of this approach increases with the block-size, similarly to the parallel ENUM with the dimension, and attains a speedup of up to 13.72x for a lattice in dimension 60, reduced with block-size 50,

	Threads	Block-size					
		40		45		50	
ENUM_BKZ	2	1.90x	95%	1.98x	99%	2.00x	100%
	4	3.71x	93%	3.86x	97%	3.95x	99%
	8	6.37x	80%	6.96x	87%	7.67x	96%
	16	9.69x	61%	11.99x	75%	13.72x	86%
	32	10.68x	33%	17.37x	54%	21.62x	68%
AC_BKZ	2	1.81x	91%	1.83x	93%	1.78x	89%
	4	3.17x	79%	3.03x	76%	2.98x	75%
	8	4.81x	60%	4.24x	53%	4.14x	52%
	16	5.84x	37%	4.84x	75%	4.57x	29%
	32	5.51x	17%	4.69x	15%	4.55x	14%

Table 4.9: Speedup (S) and Efficiency (E) of both parallel BKZ implementations for a lattice in dimension 60 for block-sizes 40, 45 and 50.

		Block-size					
		40		45		50	
	Threads	S	E	S	E	S	E
ENUM_BKZ	2	1.87x	93%	1.97x	98%	1.97x	99%
	4	3.58x	89%	3.86x	97%	3.93x	98%
	8	5.88x	73%	7.04x	88%	7.41x	93%
	16	8.36x	52%	12.20x	76%	13.49x	84%
	32	8.69x	27%	18.20x	57%	21.76x	68%
AC_BKZ	2	1.88x	95%	1.86x	93%	1.87x	94%
	4	3.30x	83%	3.25x	81%	3.28x	82%
	8	5.23x	65%	4.82x	60%	4.66x	58%
	16	6.79x	42%	5.70x	36%	5.51x	34%
	32	6.87x	21%	5.51x	17%	5.49x	17%

Table 4.10: Speedup (S) and Efficiency (E) of both parallel BKZ implementations for a lattice in dimension 70 for block-sizes 40, 45 and 50.

using 16 threads. It is expected that for bigger block-sizes a better speedup would be obtained, but BKZ becomes impractical for such block-sizes. Another important fact is that the scalability is almost the same for different dimensions and, therefore, it would presumably be similar for lattices in higher dimensions.

The AC_BKZ implementation attains speedups that are bounded by the load imbalance of up to 6.87x on a lattice in dimension 70, reduced with block-size 40, using 32 threads. Efficiency levels of 92%, for 2 threads, and 81%, for 4 threads, were achieved. Since only $n-\beta+1$ enumeration calls can be performed at the same time (see Section 4.1.5), the scalability of this implementation increases with the dimension, but decreases with the block-size.

The comparison of both implementations showed that the approach with the more fine grained level of parallelism, the BKZ with the parallel ENUM, performed better, since the workload is much better distributed among all threads and the load imbalance is much lower. This allowed it to achieve higher speedups than on the BKZ with parallel blocks, which used a more coarse grained approach.

Due to time restrictions, we could not invest further effort in joining the ENUM_BKZ implementation with the work developed on the enumeration and on the enumeration with extreme pruning. A novel parallel approach of the enumeration with extreme pruning would consist in, inside each iteration, running the parallel ENUM_BKZ implementation together with a parallel extreme pruned enumeration call, using the parallelization technique described in Section 4.2.1. Multiple extreme pruning iterations would run on multiple processes, distributed among multiple nodes of a computing cluster, in a distributed computing mudel. This hybrid approach could, therefore, be used to achieve higher speedups by using much more CPU cores.

4.3 Implementations on CUDA/GPU

In 2010, a parallel implementation for GPUs of ENUM was proposed (Hermans et al., 2010). Their approach consisted in creating a large amount of tasks on the CPU and sending them to the GPU. The implementation uses a parameter M to stop the computation of the tasks every M iterations. When this occurs, each thread either keeps computing the task that it was computing before or, if the previous task already finished, it receives a new task. On each iteration one of the following operations can be performed: *move up* on the enumeration tree, *move down* or update the shortest vector found so far, from here on called *update best*. Since each of these operations has different instructions, a considerable thread divergency can occur, therefore, hurting the performance of the implementation.

Our CUDA implementation of ENUM uses shared copies of the input variables (n, μ and $||b_1^*||^2, ..., ||b_n^*||^2$), the best vector found so far and its norm ($\hat{\mathbf{u}}$ and C, respectively) and private copies of the structures used for computation (**E**, **u**, **c**, **dist**, Δ, **d**). A large amount of tasks is created and sent to the GPU. Each task is represented by the **u** coefficients that were already computed and the value of $last\_nonzero$.

To tackle the thread divergency problem, we implemented worklists for each of the operations (*move up*, *move down* and *update best*). The maximum number of tasks that are computed at the same time (size of all worklists together) is set at the beginning and is based on the number of SMXs of the device (#$SMXs$), the maximum number of resident blocks per SMX (max_blocks) and the maximum number of threads per block ($max\_threads$) and is given by #$SMXs \times max\_blocks \times max\_threads$. From here on, we will call the tasks that are being computed *active* tasks. The implementation starts by computing the first node of each active task, which is a *move down* operation, and adds each active task to its corresponding worklist. Afterwards, the operation of the respective worklist (*move up*, *move down* and *update best*) is computed for each task. The *move up* and *move down* worklist is computed by different kernels. Both kernels use *streams* to potentially run in parallel.

The *update best* operation consists in performing a *min reduction* primitive if more than one vectors were found that are shorter than the current shortest vector. Otherwise, if only one vector was found, the shortest vector and its norm are updated with cudaMemcpy operations. In the first case, a kernel is called to fill a vector with the norms of the vectors that were found. This vector is then sent to the *min reduction* primitive, which is performed by the CUDPP library[4].

[4]CUDA Data Parallel Primitives Library, a library of data-parallel algorithm primitives for devices that support CUDA; more details at http://cudpp.github.io/

After all kernels finished their computation, a kernel is called that adds all active tasks to their respective worklists for the next iteration, with the aid of atomic operations (atomicInc). In our implementation, instead of stopping the algorithm every M iterations, we use a parameter MT that defines the maximum number of tasks that can be already finished. When MT or more tasks finished, a kernel is executed that re-initializes the structures of the tasks that finished to reuse memory. All this process is repeated until all three worklists are empty and there are no more tasks awaiting their computation.

Although this preliminary implementation seemed to solve the thread divergency problem, its performance is still behind our expectations. The problem with this preliminary implementation is the load imbalance, since some tasks finish much faster than others. At some point the number of tasks in the worklists is not anymore sufficient to use all the computational capacity of the GPU. Unfortunately, due to time restrictions, it was not possible to further optimize this implementation.

Chapter 5

Conclusions

This dissertation presents a comparison between multiple sequential implementations of SVP-solvers, namely the Voronoi cell-based algorithm, presented in (Agrell et al., 2002) and two enumeration-based algorithms, SE++ and ENUM, proposed in (Ghasemmehdi and Agrell, 2009) and (Gama et al., 2010), respectively. We also show that different techniques and optimizations that significantly improve the performance of ENUM can also be applied to other enumeration-based algorithms, namely the extreme pruning technique and the optimization that avoids symmetric branches of the enumeration tree.

We present the first practical results of the performance of the Voronoi cell-based algorithm and compare this algorithm to enumeration algorithms. Despite of displaying potential for an efficient parallel implementation, the sequential implementation of the Voronoi cell-based algorithm performed considerably worse than the enumeration-based solvers.

The computation of symmetric subtrees in enumeration-based SVP-solvers is redundant, because the resultant vectors of those subtrees are always identical in terms of norm. While ENUM, the fastest SVP-solver known to this day, avoids this computation, the SE++ solver, another important enumeration-based SVP-solver does not. We show that, by ignoring the computation of symmetric subtrees, the SE++ SVP-solver is accelerated in almost 50%. As a result, it outperforms ENUM by a factor of 3% (on average).

We developed, implemented and assessed multi-core CPU parallel implementations of the enumeration-based SVP-solvers, (1) for shared memory systems, without extreme pruning, and (2) for distributed memory systems with extreme pruning. Extreme pruning is an very important technique that transforms enumeration algorithms into efficient heuristics. Our approaches can be applied to both algorithms, since both have very similar workflows.

Our parallel implementations of the solvers without extreme pruning can be efficiently parallelized, achieving linear speedups for up to 8 threads and almost linear speedups for 16 threads. The use of 16 threads implies the use of two CPU sockets, which can explain the loss of linear scalability. The implementation achieves super-linear speedups in some instances on up to 8 cores (see, for instance, the results for lattices in dimension 56), due to workload savings with the parallel algorithm, as explained in Section 4.2.3. 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.

The parallel approach of our extreme pruned enumeration can also be applied to both algorithms. However, the scalability of this approach is lower than the scalability of the versions without extreme pruning, due to load imbalance. This problems occurs since the number of iterations is fixed to 44 and does not depend on the number of processes that are used. Nevertheless, speedups of almost 13x were achieved with 16 processes, with efficiency levels of up to 81%.

We also developed a new BKZ implementation, AC_BKZ, that changes the workflow of the algorithm with the intention of achieving a better scalability, since it could be parallelized at a more coarse-grained level. However, in practice, when comparing this implementation to the version that follows the original workflow of the algorithm and performs calls to a parallel ENUM, ENUM_BKZ, not only was the later faster in sequential, but also achieved a much better scalability in parallel. While the AC_BKZ implementation achieved speedups of up to 6.79x on a lattice in dimension 70, reduced with block-size 40, ENUM_BKZ achieved speedups of up to 13.72% for a lattice in dimension 60, reduced with block-size 50. It is also relevant to say that the scalability of AC_BKZ increases with the dimension, but decreases with the bock-size. On the other hand, the scalability of ENUM_BKZ is constant for different dimensions and increases with the block-size.

We also assessed the quality of bases computed by different BKZ reduction implementations. The results show that, in the general case, ENUM_BKZ computes the bases with the best quality. On the other hand, AC_BKZ and G_BKZ_FP, an implementation of the NTL library, show very distinct basis qualities. In some instances AC_BKZ computes the bases with better quality, while on others it is G_BKZ_FP.

Therefore, it is possible to conclude that the scalability of enumeration algorithms without extreme pruning can be linear, while the scalability of these algorithms without extreme pruning is slightly lower on systems with at least 32 logical cores. Sequentially ENUM is more efficient than SE++, but SE++ can outperform ENUM by a factor of 3% with the optimization that avoids symmetric branches. Finally, we also show that a parallel ENUM implementation can significantly improve the performance of the BKZ reduction algorithm.

5.1 Future Work

The results presented in this dissertation can motivate further research in this field. Since lattice-based cryptography is still an expanding field, lattice-based algorithms can still be improved, either by improving the exact variants of the algorithms or by finding heuristic counterparts, which have shown to considerably outperform exact solvers. be studied.

One such example is the Voronoi cell-based algorithm. Only the algorithm that computes the exact SVP is known yet, which is substantially slower than other SVP-solvers. However, with further research in this algorithm, it could be possible to find more efficient variants of this algorithm and an heuristic that solves the SVP approximately, which could even outperform the currently known SVP-solvers. Since this algorithm displays potential for parallelization, parallel approaches could also developed on a practical variant of this algorithm.

On the other hand, the bounding functions of enumeration-based solvers with extreme pruning could also be targeted to prune the enumeration tree even more. The suitability of the extreme pruning technique on the algorithm of Micciancio and Walter (Micciancio and Walter, 2014) also still has to be explored. Parallel approaches of the extreme pruning technique can still be improved as shown by the novel approach that we propose, which combines a shared memory model with a distributed memory model, therefore, increasing the number of cores that can be used to increase its performance.

Bibliography

Erik Agrell, Thomas Eriksson, Alexander Vardy, and Kenneth Zeger. Closest Point Search in Lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.

Miklós Ajtai. The Shortest Vector Problem in L_2 is *NP*-hard for Randomized Reductions (Extended Abstract). In *STOC*, pages 10–19, 1998.

Miklós Ajtai, Ravi Kumar, and D. Sivakumar. A Sieve Algorithm for the Shortest Lattice Vector Problem. In *STOC*, pages 601–610, 2001.

Thomas Arnreich. A comprehensive comparison of BKZ implementations on multi-core CPUs, 2014.

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011. doi: 10.1002/cpe.1631.

László Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1): 1–13, 1986.

João Barbosa. GAMA framework: Hardware Aware Scheduling in Heterogeneous Environments. *Tech. rep. Computer Science Dept., University of Texas at Austin*, Sept. 2012.

Mihir Bellare, Shafi Goldwasser, and Daniele Micciancio. "Pseudo-Random" Number Generation Within Cryptographic Algorithms: The DDS Case. In *CRYPTO*, pages 277–291, 1997.

Johannes Blömer and Stefanie Naewe. Sampling Methods for Shortest Vectors, Closest Vectors and Successive Minima. *Theor. Comput. Sci.*, 410(18):1648–1665, April 2009. ISSN 0304-3975. doi: 10.1016/j.tcs.2008.12.045.

Johannes Blömer and Jean-Pierre Seifert. On the Complexity of Computing Short Linearly Independent Vectors and Short Bases in a Lattice. In *STOC*, pages 711–720, 1999.

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.

John W. Cassels. *An Introduction to the Geometry of Numbers (Reprint)*. Classics in mathematics. Springer, 1997. ISBN 978-3-540-61788-4.

Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better Lattice Security Estimates. In *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2011. doi: 10.1007/978-3-642-25385-0_1.

Matthijs J. Coster, Antoine Joux, Brian A. LaMacchia, Andrew M. Odlyzko, Claus-Peter Schnorr, and Jacques Stern. Improved Low-Density Subset Sum Algorithms. *Computational Complexity*, 2:111–128, 1992.

Özgür Dagdelen and Michael Schneider. Parallel Enumeration of Shortest Lattice Vectors. In *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II*, pages 211–222, 2010. doi: 10.1007/978-3-642-15291-7_21.

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. doi: 10.2307/2007966.

Robert Fitzpatrick, Christian Bischof, Johannes Buchmann, Ozgur Dagdelen, Florian Gopfert, Artur Mariano, and Bo-Yin Yang. Tuning GaussSieve for Speed. *Third International Conference on Cryptology and Information Security in Latin America (Latincrypt), Florianopolis, Brazil*, 2014.

Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice Enumeration Using Extreme Pruning. In *EUROCRYPT*, pages 257–278, 2010.

Arash Ghasemmehdi and Erik Agrell. Optimal Projection Method in Sphere Decoding. *CoRR*, abs/0906.0249, 2009.

Guillaume Hanrot and Damien Stehlé. Worst-Case Hermite-Korkine-Zolotarev Reduced Lattice Bases. *CoRR*, abs/0801.3331, 2008.

Bettina Helfrich. Algorithms to Construct Minkowski Reduced an Hermite Reduced Lattice Bases. *Theor. Comput. Sci.*, 41:125–139, 1985.

Jens Hermans, Michael Schneider, Johannes Buchmann, Frederik Vercauteren, and Bart Preneel. Parallel Shortest Lattice Vector Enumeration on Graphics Cards. In *Progress in Cryptology - AFRICACRYPT 2010, Third International Conference on Cryptology in Africa, Stellenbosch, South Africa, May 3-6, 2010. Proceedings*, pages 52–68, 2010. doi: 10.1007/978-3-642-12678-9_4.

Tsukasa Ishiguro, Shinsaku Kiyomoto, Yutaka Miyake, and Tsuyoshi Takagi. Parallel Gauss Sieve Algorithm: Solving the SVP in the Ideal Lattice of 128 dimensions. *IACR Cryptology ePrint Archive*, 2013:388, 2013.

Antoine Joux and Jacques Stern. Lattice Reduction: A Toolbox for the Cryptanalyst. *J. Cryptology*, 11(3): 161–185, 1998.

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. ISBN 0-89791-099-0. doi: 10.1145/800061.808749.

Ravi Kannan. Minkowski's Convex Body Theorem and Integer Programming. *Math. Oper. Res.*, 12(3): 415–440, August 1987. ISSN 0364-765X. doi: 10.1287/moor.12.3.415.

Po-Chun Kuo, Michael Schneider, Özgür Dagdelen, Jan Reichelt, Johannes Buchmann, Chen-Mou Cheng, and Bo-Yin Yang. Extreme Enumeration on GPU and in Clouds - - How Many Dollars You Need to Break SVP Challenges -. In *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*, pages 176–191, 2011. doi: 10.1007/978-3-642-23951-9_12.

Susan Landau and Gary L. Miller. Solvability by Radicals is in Polynomial Time. *J. Comput. Syst. Sci.*, 30 (2):179–208, 1985.

A.K. Lenstra, H.W.jun. Lenstra, and Lászlo Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.

H.W.jun. Lenstra. Integer programming with a fixed number of variables. *Math. Oper. Res.*, 8:538–548, 1983.

Xianghui Liu, Xing Fang, Zheng Wang, and Xianghui Xie. A new parallel lattice reduction algorithm for BKZ reduced bases. *SCIENCE CHINA Information Sciences*, 57(9):1–10, 2014. doi: 10.1007/s11432-013-4967-6.

Artur Mariano, Özgür Dagdelen, and Christian Bischof. A comprehensive empirical comparison of parallel ListSieve and GaussSieve. In *APCI&E - Workshop on Applications of Parallel Computation in Industry and Engineering*, 2014a.

Artur Mariano, Shahar Timnat, and Christian Bischof. Lock-free GaussSieve for Linear Speedups in Parallel High Performance SVP Calculation. In *SBAC-PAD - 26th International Symposium on Computer Architecture and High Performance Computing*, 2014b.

Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. *Electronic Colloquium on Computational Complexity (ECCC)*, 16:65, 2009.

Daniele Micciancio and Michael Walter. Fast Lattice Point Enumeration with Minimal Overhead. *IACR Cryptology ePrint Archive*, 2014:569, 2014.

Benjamin Milde and Michael Schneider. A Parallel Implementation of GaussSieve for the Shortest Vector Problem in Lattices. In *Parallel Computing Technologies - 11th International Conference, PaCT 2011, Kazan, Russia, September 19-23, 2011. Proceedings*, pages 452–458, 2011. doi: 10.1007/978-3-642-23178-0_40.

Phong Q. Nguyen and Damien Stehlé. Floating-Point LLL Revisited. In *Advances in Cryptology - EUROCRYPT 2005, 24th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Aarhus, Denmark, May 22-26, 2005, Proceedings*, pages 215–233, 2005. doi: 10.1007/11426639_13.

Phong Q. Nguyen and Damien Stehlé. LLL on the Average. In *Algorithmic Number Theory, 7th International Symposium, ANTS-VII, Berlin, Germany, July 23-28, 2006, Proceedings*, pages 238–256, 2006. doi: 10.1007/11792086_18.

Phong Q. Nguyen and Jacques Stern. The Two Faces of Lattices in Cryptology. In *CaLC*, pages 146–180, 2001.

Phong Q. Nguyen and Thomas Vidick. Sieve Algorithms for the Shortest Vector Problem are Practical. *J. Mathematical Cryptology*, 2(2):181–207, 2008.

A. M. Odlyzko. The Rise and Fall of Knapsack Cryptosystems. In *In Cryptology and Computational Number Theory*, pages 75–88. A.M.S, 1990.

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. ISSN 0163-5824. doi: 10.1145/1089242.1089247.

Claus-Peter Schnorr. A More Efficient Algorithm for Lattice Basis Reduction. *J. Algorithms*, 9(1):47–62, 1988. doi: 10.1016/0196-6774(88)90004-1.

Claus-Peter Schnorr and M. Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. *Math. Program.*, 66:181–199, 1994.

Xiaoyun Wang, Mingjie Liu, Chengliang Tian, and Jingguo Bi. Improved Nguyen-Vidick Heuristic Sieve Algorithm for Shortest Vector Problem. In Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong, editors, *ASIACCS*, pages 1–9. ACM, 2011. ISBN 978-1-4503-0564-8.