

High performance algorithms for lattice-based cryptanalysis

Technische Universität Darmstadt

Artur Mariano, MSc
(geboren in Guarda, Portugal)

Tag der Einreichung: 03.08.2016
Tag der mündlichen Prüfung: 22.09.2016

Dissertation approved by the Fachbereich Informatik in fulfillment of the requirements for the degree of Dr.-Ing.

Referenten:
Prof. Dr. Christian Bischof
Prof. Dr. Keshav Pingali

Darmstadt, 2016
D17

Acknowledgements

"I am not the man I once was. I do not want to go back in time, to be the second son, the second man.",
Vasco da Gama, Portuguese explorer.

I was once told by an experienced researcher that *"it should come at no surprise that even the most persistent drop out of their PhD, because a PhD is more of a test of character than one of intellect. Going abroad increases these odds, because there you lose control of many variables."* I remember thinking that most of what I had achieved up until that point was due to my persistence and hard work, rather than my intellect, and I thought that would be enough to overcome any problem I might have down the road. Little did I know.

Living abroad for 4 years teaches you more than you can possibly imagine, but you must be prepared for adversities. If the road gets dark (and Germany lacks sunlight a lot!), you will have to pull it off. The good news is that you will not want to be the man you once were. Throughout this journey of mine, I encountered many people who I will remember forever. Without many of them, this thesis would not have been possible. I am extremely appreciative of all of them, who I humbly acknowledge in the following lines.

I thank my advisor, professor Christian Bischof, for giving me the opportunity to engage in a topic that fascinated me from day one, giving me complete independence to carry out my research, improving the readability of my papers, and reviewing this dissertation.

I want to especially thank the people with whom I worked the most during my PhD, Özgür Dagdelen and Fábio Correia, but not only for the great work we have done together. Özi for introducing me to the German and Turkish cultures, for going to bat for me whenever I ran low on motivation and battery, and for his invaluable friendship. Fábio for his devotion to the work I proposed, for engaging in discovering the most remote parts of Germany and German with me, and for preventing me from getting homesick many times.

I want to thank all my co-authors, including Christian Bischof, Özgür Dagdelen and Fábio Correia, for the possibility of working together. Roberto Ribeiro, Alberto Proença, Cristiano Sousa, Mattias Diener, Philippe Navaux, Erik Agrell, Bo-Yin Yang, Florian Göptert, Johannes Buchmann, Leonel Sousa and Paulo Garcia. I particularly thank Shahar Timnat, for his highly valuable contributions to my ideas, Robert Fitzpatrick, for sharing his genius with me, and Thijs Laarhoven, for embarking with me on such beautiful work and visiting me in Darmstadt.

I also thank my student Cristiano Sousa for the numerous conversations we had about my work and helping me out at many stages of my work. To my previous advisor, Alberto Proença, I thank his

support in starting my PhD and for the many possibilities of collaboration with graduate students from the University of Minho, some of whom I brought to Germany. He instilled in me a unique ability to critically question my own results, regardless of they were good or bad, and I am sure that this shaped the type of researcher I went on to be.

To my colleagues at the Institute of Scientific Computing; Johannes Willkomm for his endless availability, Alex Hück for the nice chats over coffee and keeping my mental sanity, and Jan Patrick for his effort and patience in teaching me German. To the people at the HRZ whom I interacted with, such as Sergey Boldyrev for the intellectual conversations and Geert Geurts for his friendship. Pal, *Dank u wel*. Señora Novato, for preventing my Spanish from getting rusty and for being so nice to me, *muchas gracias*, and Andreas Wolf for giving me access to hardware that was essential to my work. To Ilona from House of IT, for her motivational smile and good mood. To the guys from the third floor, especially Gorgia, Felix Günther, and Paul Bächer, for creating such a nice environment. To Giulia and Sabrina, for understanding my Latino heart so well. You prevented me from cracking up many times!

To the people who hosted me, including Georg Hager (RRZE, Erlangen, Germany) and Leonel Sousa (IST, Lisbon, Portugal). To my colleagues at the IST, especially Diego Souza and Aleksandar Ilic, for the formidable environment at the office, it felt like home to me, folks!

To Wolfgang from ProPhysio, for his ability to heal up my injuries from my workouts, in magical physiotherapy sessions. To the nice buddies who worked out with me at the gym, like Marcel, Alvaro, and the rest of the gang.

To people from CASED. To Ana, for helping me out when I first got in Germany, *muito obrigado*, to Rodrigo, for his friendship and opportunity to keep up with Spanish, *tambien muchas gracias amigo*.

To CROSSING, a DFG project that funded a great part of my PhD, numerous trips to conferences, visits of guests of mine and other things. To Stefanie Kettler, CROSSING's secretary.

To the University of Minho, which I graduated from, and LabCG, my former research group, as well as all its members, namely Luis Paulo Santos, Nuno, Ricardo, Roberto and Waldir.

To all the great (mainly Soul music) singers who eased the write up of this dissertation. James Brown, Marvin Gay, Nina Simone, Ben King, Otis Redding, Bob Neuwirth, Leela James, Alloe Blacc and Annie Lennox. To the Corner café in Grafenstrasse, DA, for being my office during a good chunk of the write up.

To my references and sources of inspiration, in science and out of it: Rogério Alves (Portuguese lawyer - exceptionally talented user of Portuguese), Bryan Stevenson (American lawyer and activist - inspiring activist against segregation and over-incarceration of black people and minorities), Warren Buffet (American investor - living proof that we don't have to cut corners in life in order to succeed), Gordon Ramsay (Scottish Chef - portrays passion and perfectionism in what we do), Kai Greene (American bodybuilder - portrays will, hard work and humility at its best), Bruno de Carvalho (Portuguese, President of Sporting Clube de Portugal - personifies courage and love) and Anderson 'The spider' Silva (Brazilian mixed martial artist - personifies humility, raw talent and faith). And to my role model and a person whom I relate to a lot, Percy Julian, American exceptional scientist, writer and activist, and proof that hard work and persistence always win, even when everybody want us to fail.

Last but not least, I want to thank my family - especially my beloved mother - and friends, who supported me from the very first day on. To my grandmother, who is not around any more to see her grandson graduating, I dedicate this thesis.

"Keep your eyes on the prize, hold on."

Folk song that became influential during the American civil rights movement of the 1950s and '60s.

Abstract

With quantum-computing, classical cryptosystems, such as RSA, can easily be broken. Today, lattice-based cryptography stands out as one of the most promising and prominent post-quantum type of cryptosystems. The cryptanalysis of new types of cryptography is a crucial part of their development, as it allows one to understand and improve the degree of security of these systems. The same way the security of RSA is deeply connected to the factorization of large integers, the security of lattice-based cryptography revolves around lattice problems such as the Shortest Vector Problem (SVP).

While the cryptography community has developed in-depth knowledge of the algorithms that solve these problems (which we also refer to as attacks), from a theoretical point of view, the practical performance of these algorithms is commonly not well understood. In particular, the practical performance of many classes of attacks is not congruent with theoretical expectations. This gap in knowledge is problematic because the security parameters of cryptosystems are selected based on the asymptotic complexity of the available attacks, but only those that are proven to be practical and scalable are considered. Therefore, if some theoretically strong algorithms are erroneously ruled out from this process, because they are believed to be impractical or not scalable, the security parameters of cryptosystems may not be appropriate. In particular, overly strong parameters lead to inefficient cryptosystems, while overly weak parameters render cryptosystems insecure. This is the reason why one must determine the real potential of attacks in practice. The key to understanding is to consider the underlying computer architecture and its influence on the performance of these algorithms, so an effective map between the algorithm and the architecture can be done. This means in particular, to develop appropriate parallelization methods for these algorithms, as all modern computer architectures employ parallel units of various flavours.

This thesis aims to fill this gap in knowledge, by describing computational analyses and techniques to parallelize and optimize attacks, with focus on sieving algorithms, in modern, parallel computer architectures. In particular, we show that (i) lattice basis reduction algorithms can benefit largely from cache friendly data structures and scale well, if the right parameters are used, (ii) enumeration algorithms can scale linearly and super-linearly if appropriate mechanisms are employed and (iii) sieving algorithms can be implemented in such a way that very good scalability is achieved, even for high core counts, if the properties of the algorithms are slightly relaxed. Throughout the thesis, we also provide heuristics to enhance the practical performance of specific algorithms, and various optimizations in practice, especially related to memory access.

Zusammenfassung

Vor ungefähr drei Jahrzehnten hat die Kryptographiegemeinschaft begonnen gegen Angriffe mittels Quantencomputern resistente Kryptosysteme zu finden, aufgrund der Verletzlichkeit von klassischen Kryptosystemen durch diese Angriffe, wie beispielsweise RSA. Heutzutage ist die gitterbasierte Kryptographie eines der vielversprechendsten und prominentesten Post-Quantumkryptosysteme für eine Vielzahl von Gründen. Die Kryptoanalyse neuer Arten von Kryptographie ist ein wichtiger Teil ihrer Entwicklung, da es den Sicherheitsgrad dieser Systeme zu verstehen und zu verbessern ermöglicht. Auf die gleiche Weise wie die Sicherheit von RSA tief mit der Primfaktorzerlegung großer Ganzzahlen verbunden ist, dreht sich die Sicherheit der gitterbasierte Kryptographie um einige Gitterprobleme, einschließlich des “Shortest Vector Problem” (SVP).

Während die Kryptographiegemeinschaft vertiefte Kenntnisse aus theoretischer Sicht der Algorithmen, die diese Probleme lösen (die wir auch als Angriffe bezeichnen), entwickelt hat, ist die praktische Performanz dieser Algorithmen häufig nicht ausreichend untersucht. Insbesondere ist die praktische Leistung vieler Klassen von Angriffen nicht kongruent mit unseren theoretischen Erwartungen. Diese Wissenslücke ist problematisch, weil die Sicherheitsparameter von Kryptosystemen basierend auf der asymptotischen Komplexität der verfügbaren Angriffe ausgewählt werden, aber nur diejenigen, die praktisch und skalierbar sind, werden berücksichtigt. Wenn deshalb theoretisch starke Algorithmen fälschlicherweise aus diesem Prozess ausgeschlossen werden, sind die Sicherheitsparameter von Kryptosystemen nicht angemessen. Insbesondere führen zu starke Parameter zu ineffizienten Kryptosystemen, während übermäßig schwache Parameter Kryptosysteme unsicher machen. Dies ist der Grund, warum man das maximale Potenzial aller Angriffe in der Praxis bestimmen muss. Der Schlüssel um das maximale Potenzial zu erreichen, ist (i) die darunterliegende Computerarchitektur und deren Einfluss auf die Leistung dieser Algorithmen zu betrachten, so dass eine effektive Abbildung zwischen dem Algorithmus und der Architektur durchgeführt werden kann, und (ii) die Entwicklung geeigneter Parallelisierungsmethoden für diese Algorithmen, da moderne Computerarchitekturen überwiegend parallel sind.

Diese Arbeit zielt darauf ab diese Wissenslücke zu füllen mittels der Beschreibungen von rechnerischen Analysen und Parallelisierungs- und Optimierungstechniken von Angriffen, mit Fokus auf “Sieving” Algorithmen, in modernen, parallelen Rechnerarchitekturen. Insbesondere zeigen wir, dass (i) Gitterbasis Reduktionsalgorithmen weitgehend von “Cache”-freundlichen Datenstrukturen profitieren können und dass sie gut skalieren, wenn die richtigen Parameter verwendet werden, (ii) “Enumeration” Algorithmen können linear und super-linear skalieren, wenn geeignete Mechanismen eingesetzt werden und (iii) “Sieving” Algorithmen können in einer Weise implementiert werden, die sehr gute Skalierbarkeit erreicht,

selbst für eine hohe Anzahl von Kernen, wenn die Eigenschaften der Algorithmen leicht entspannt und ausgenutzt werden. Im Lauf der These beschreiben wir auch Heuristiken, um die praktische Leistung von spezifischen Algorithmen zu verbessern, und verschiedenen praktischen Optimierungen, vor allem Verbesserungen im Zusammenhang mit dem Speicherzugriff.

Theses. This dissertation states and validates several theses, including:

- One may have to explore low-level threading mechanisms in order to extract full potential of certain algorithms for lattice-based cryptanalysis (Validated in Chapter 4).
- The mathematical properties of several SVP-solvers can be relaxed to improve their practical performance (Validated in Chapters 4 and 6).
- The performance of many of the algorithms to reduce lattice bases and solve the SVP is not well understood, and more efficient implementations can be achieved if one has simultaneously enough knowledge of the underlying computer architecture and the algorithm (Validated in Chapter 5).
- Many SVP-solvers are perfectly suited for parallel computation and some of them can achieve super-linear speedups without sacrificing the solution (Validated in Chapters 4 and 6).
- The performance of sieving algorithms varies upon how we interpret them; despite the algorithmic descriptions do not define the order to reduce vectors, different orders impact performance differently (Validated in Chapter 6).
- The practicability of sieving algorithms is considerably higher than believed and scalable implementations can be developed if one has enough knowledge of the underlying computer architecture and the algorithms (Validated in Chapter 6).

Publications pertaining to this dissertation, in the period 2014-2016:

- 1) Artur Mariano, Thijs Laarhoven and Christian Bischof: "A Parallel Variant of LDSieve for the SVP on Lattices" @ Submitted to SBAC-PAD'16.
- 2) Paulo Garcia, Artur Mariano and Leonel Sousa: "A Survey on Fully Homomorphic Encryption: an Engineering Perspective", @ Submitted to ACM Computing Surveys.
- 3) Fábio Correia, Artur Mariano, Alberto Proenca, Christian Bischof and Erik Agrell: "Parallel Improved Schnorr-Euchner Enumeration SE++ on Shared and Distributed Shared Memory, With and Without Extreme Pruning" @ Submitted to JoWUA.
- 4) Artur Mariano^{*}, Fábio Correia and Christian Bischof: "A vectorized, cache efficient LLL implementation" @ VECPAR'16 - 12th International Meeting on High Performance Computing for Computational Science, Porto, Portugal, June 28-30, 2016. (Acceptance Rate: 50%)
- 5) Artur Mariano^{*}, Matthias Diener, Christian Bischof and Philippe Navaux: "Analyzing and Improving Memory Access Patterns of Large Irregular Applications on NUMA Machines" @ PDP'16 - 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Heraklion, Greece, February 17-19, 2016. Short paper (Acceptance Rate: 31.5%)
- 6) Fábio Correia, Artur Mariano, Alberto Proenca, Christian Bischof and Erik Agrell: "Parallel Improved Schnorr-Euchner Enumeration SE++ for the CVP and SVP" @ PDP'16 - 24th Euromicro International

Conference on Parallel, Distributed, and Network-Based Processing, Heraklion, Greece, February 17-19, 2016. (Acceptance Rate: 31.5%)

7) Artur Mariano* and Christian Bischof: "Enhancing the scalability and memory usage of HashSieve on multi-core CPUs", @ PDP'16 - 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Heraklion, Greece, February 17-19, 2016. (Acceptance Rate: 31.5%)

8) Artur Mariano*, Thijs Laarhoven and Christian Bischof: "Parallel (probable) lock-free HashSieve: a practical sieving algorithm for the SVP" @ ICPP'15 - 44th International Conference on Parallel Processing, Beijing, China, September 1-4, 2015. (Acceptance Rate: 32%)

9) Artur Mariano*, Shahar Timnat and Christian Bischof: "Lock-free GaussSieve for Linear Speedups in Parallel High Performance SVP Calculation" @ SBAC-PAD'14 - 26th International Symposium on Computer Architecture and High Performance Computing, Paris, France, October 22-24, 2014. (Acceptance Rate: 32%)

10) Robert Fitzpatrick, Christian Bischof, Johannes Buchmann, Özgür Dagdelen, Florian Göpfert, Artur Mariano* and Bo-Yin Yang: "Tuning GaussSieve for Speed" @ LATINCRYPT'14 - 3rd International Conference on Cryptology and Information Security in Latin America, Florianópolis, Brazil, September 17-19, 2014. (Acceptance Rate: 39.6%)

11) Christian Bischof, Johannes Buchmann, Özgür Dagdelen, Robert Fitzpatrick, Florian Göpfert, and Artur Mariano: "Nearest Planes in Practice" @ BALCANCRYPT'14 - International Conference on Cryptography and Information security, Istanbul, Turkey, October 16-17, 2014.

12) Artur Mariano*, Özgür Dagdelen and Christian Bischof: "A comprehensive empirical comparison of parallel ListSieve and GaussSieve" @ APCI&E'14 - Workshop on Applications of Parallel Computation in Industry and Engineering (in conjunction with Euro-Par), Porto, Portugal, August 25-29, 2014.

(★ included presentation)

Contents

1	Introduction	15
1.1	Ancient cryptography	15
1.1.1	Ciphers	16
1.1.2	War driven cryptography and the enigma machine	16
1.2	Modern cryptography	18
1.2.1	Symmetric and asymmetric cryptography	18
1.2.2	Cryptanalysis of modern cryptography	20
1.2.3	Somewhat/fully homomorphic encryption	21
1.3	Quantum-immune cryptography	22
1.3.1	Lattice-based cryptography	22
1.3.2	Lattice-based cryptanalysis	24
1.4	Contributions, structure and technical details of the thesis	25
2	Preliminaries	27
2.1	Lattices	27
2.1.1	Notation and definitions	27
2.1.2	Lattice problems	30
2.2	Parallel computing	31
2.2.1	Shared-memory	31
2.2.2	Distributed-memory	32
2.3	Test environment	33
3	State of the Art of Lattice-based Cryptanalysis	35
3.1	The landscape of lattice-based cryptanalysis	35
3.2	Algorithms	36
3.2.1	Lattice basis reduction	36
3.2.2	The shortest vector problem	37
3.3	Implementations	39
3.3.1	Lattice basis reduction algorithms	40
3.3.2	CVP- and SVP-solvers	41
3.4	Contributions of this thesis to the state of the art	42

4	Parallel Vector Enumeration	45
4.1	Enumeration algorithms	45
4.2	Parallel ENUM and SE++ on shared-memory systems	47
4.2.1	Tasking mechanism on SE++	48
4.2.2	An ad hoc demand-driven mechanism (on ENUM)	53
4.2.3	Summary	58
5	Parallel Efficient Lattice Basis Reduction	61
5.1	Lattice basis reduction	61
5.2	The LLL algorithm	62
5.2.1	Floating-point LLL	62
5.2.2	Performance	65
5.2.3	Data structures re-organization and SIMD vectorization	69
5.3	The Block Korkine-Zolotarev (BKZ) algorithm	73
5.3.1	State of the art of BKZ	73
5.3.2	Parallel BKZ on shared-memory machines	74
5.3.3	Summary	75
6	Scalable and Efficient Sieving Implementations	79
6.1	Sieving algorithms	79
6.2	Heuristics to speed up sieving algorithms	84
6.3	Parallelization techniques for shared-memory machines	86
6.3.1	Relevant sieving implementations	86
6.3.2	Loss-tolerant linked lists for ListSieve	87
6.3.3	Lock-free linked lists and optimizations for GaussSieve	89
6.3.4	Probable lock-free hash tables for HashSieve	95
6.3.5	A parallel variant of LDSieve	101
6.4	The memory problem	106
6.4.1	Improving memory efficiency through code optimizations	106
6.4.2	Improving memory access for NUMA machines	108
6.4.3	Summary	113
7	Conclusions and Outline	115
	Bibliography	119

Introduction

Synopsis. This introductory chapter describes briefly the evolution of cryptography over the past thousand years until present. After that, we present lattice-based cryptography, a promising candidate for quantum-immune cryptography. We then present the main problems underpinning the security of lattice-based cryptosystems, including lattice basis reduction, the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP). At the end of the chapter, we give an overview of this dissertation, with regard to its structure, goals and contents of each chapter.

“The key to growth is the introduction of higher dimensions of consciousness into our awareness.”

Lao Tzu, philosopher of ancient China

Cryptology is the science that focuses on the study of cryptography and cryptanalysis. The Cambridge Dictionary Online¹ defines "cryptography" as *the practice of creating and understanding codes that keep information secret*. Usually, “codes” are referred to as “schemes” or “cryptosystems”, terms we will use throughout this dissertation interchangeably. According to Dictionary.com², “cryptanalysis” is defined as *the science that studies the procedures, processes and methods used to translate or interpret secret writings, as codes and ciphers, for which the key is unknown*. Cryptanalysis, on one hand, enables the analysis of existing cryptosystems, thereby providing confidence in them. On the other hand, it serves as a tool to define the parameters of new cryptosystems, so that they are secure. The following sections describe the evolution of cryptography. We then expand on a particular type of cryptography, based on lattices and referred to as lattice-based cryptography, presenting the main problems which lattice-based cryptanalysis revolves around.

1.1 Ancient cryptography

Cryptography was not always mathematical science. While it is difficult to track down the exact roots of the use of cryptography, we can identify some of its less sophisticated instances throughout history. One can find many situations where making messages unintelligible was important, and even crucial for survival. For instance, military communication has largely contributed to shape and advance cryptography.

¹<http://dictionary.cambridge.org/>

²<http://dictionary.reference.com/>

Although considered primitive today, some of these types of cryptography were in fact very effective at the time, since intruders were very unlikely to attack them with success. This was primarily due to the fact that attacks were carried out with resort to manual labour, as no strong notions of security existed. Today, with powerful computers readily available, these types of cryptography are easily breakable. In the following subsections, we illustrate some of what we consider particularly relevant forms of older cryptography.

1.1.1 Ciphers

A cipher is a process used to encrypt and decrypt information, wherein a given plaintext is converted into a cryptogram, i.e., the encrypted text, and vice versa. Ciphers are usually implemented with algorithms that depend on an encryption key (also called cipher key), which can be seen as one parameter of the ciphering algorithm that determines its behaviour. One of the first known cryptosystems was the so-called Caesar's substitution cipher, named after Julius Caesar, commander of the Roman army. The cipher was created so that the Caesar's troops were not exposed, even if the messages exchanged were obtained by the enemy. With the cipher, each letter of the message was shifted by an encryption key that corresponded to the number of places in the alphabet (e.g. with three places *a* would become *d*), and the alphabet was seen as a circular sequence (for the same three places, e.g. *x* would become *a*). Empowered to communicate safely, the Roman army obtained the upper hand on the enemies during war.

After Caesar's cipher, more and more sophisticated ciphers followed. Vigenere's cipher serves as a good example. Created in the 1500's, Vigenere's cipher works like Caesar's, except that the cipher key is changed throughout the process. Instead of a pre-determined shift, as in Caesar's cipher, Vigenere's cipher uses a grid of letters that serves as the substitution method. A cipher key is chosen and repeated until the length of the plaintext is matched. For example, for the plaintext "*portuguese*", and the original cipher key "*empire*", the cipher key would become "*empirempi*", so both the plaintext and the cipher key have 10 letters. Then, the plaintext is lined up on the x-axis of the grid, and the cipher key is lined up in the y-axis. Using this process, each letter of the plaintext is then replaced with the corresponding letter in the grid. The decryption algorithm is identical, except that the cipher key and the cipher text are used to decode the plaintext.

Another cipher of major relevance is due to Thomas Jefferson, who, in the 1700's, came up with a system based on 26 physical wheels whereon the alphabet was randomly scattered. To encode a message, the wheels are lined up in such a way that the plaintext is formed. Then, the cipher text is any of the other lines. Decryption is done the same way, except that the cipher text is lined up on the wheels instead. The order of the wheels, which needs to be the same for encryption and decryption, can be seen as the cipher key. Curiously, the United States Army created the exact same system as Jefferson, in 1923, without knowing about Jefferson's invention, of which a prototype was never built.

1.1.2 War driven cryptography and the enigma machine

World War I (WWI) and World War II (WWII) were the ultimate steps towards the rise of modern cryptography, laying the foundation for an era known as war-driven cryptography [78]. A remarkable event in the context of WWI was a German telegram decrypted by British cryptographers, in early 1917, which would eventually become known as the Zimmerman telegram³. The Zimmerman telegram was an

³https://en.wikipedia.org/wiki/Zimmermann_Telegram

encrypted communication between Arthur Zimmerman, German minister for foreign issues, and Heinrich von Eckardt, the German ambassador in Mexico, where Zimmerman suggested Mexico to “*Make war together*”, thus joining the German cause. In exchange, Zimmerman declared that Mexico would have, among other things, “*An understanding from our part [Germany] that Mexico is [was] to reconquer the lost territory in Texas, New Mexico and Arizona*”. The decryption of this telegram would change not only the history of cryptanalysis, but also impact the history of the world, because the telegram was instrumental in convincing the United States to declare war against Germany and its allies.

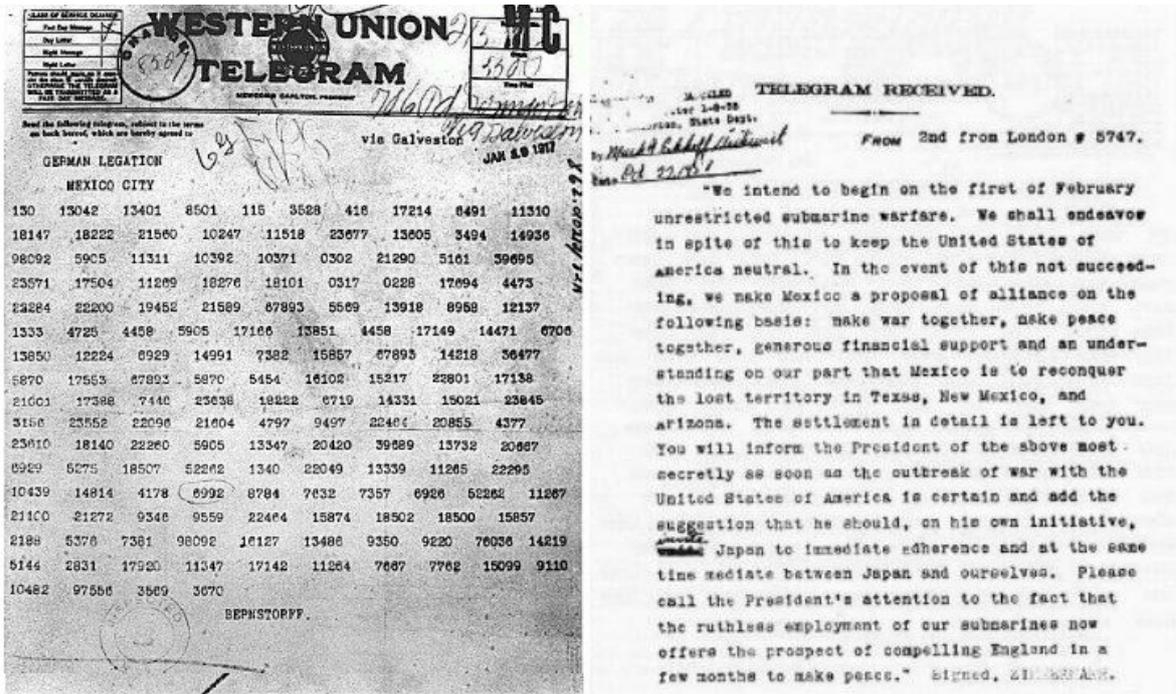


Figure 1.1: The Zimmerman telegram, coded on the left and decoded on the right. Images retrieved from Wikipedia³.

Having remained neutral for most of the war, the United States were not as developed as other countries in terms of secure communication. Eventually, Army commander Captain Lewis decided to use code talkers, soldiers who knew native American languages and used them to transmit messages. Initially, Cherokee and Choctaw Indians were used, but later on Indians from other tribes, including Lakota, Meskwaki, and Comanche were deployed by the army, also during WWII.

At the end of WWI, the German engineer Arthur Scherbius invented the Enigma encryption machine, a rotor machine for commercial and military usage [119]. Rotor machines are electro-mechanical stream cipher devices which encrypt and decrypt messages. These devices have rotors, i.e. rotating disks with the alphabet on the top and electrical contacts on either side. The wires between the rotors define a given substitution of letters, which is used both to encrypt and decrypt messages. A major security feature of these machines is that rotors advance positions whenever an encryption or decryption is done (or a key is pressed). To decrypt messages, rotor machines have to be set up with the same rotor position and order they had when encrypting the message.

It was not until WWII that the Enigma machine became famous. The first commercially available

versions were released in the 1920's, and the machine was soon adopted by the military of several countries, most notably Nazi Germany, in 1926. In 1932, exploiting both small built-in weaknesses of the machine and, especially, errors of German operators, a team of mathematicians of the Polish military intelligence, including Marian Rejewski, decoded, for the first time, German military messages enciphered on the Enigma machine. In order to communicate, not only two (or more) machines were necessary but they would also have to be set up in the same exact manner. This was extremely error prone, because battalions were often physically apart from one another, and the setup instructions would have to be carried for long distances. After 1938, the Enigma machines were made considerably more complex, and decryption became progressively harder and more time-consuming.

In 1939, Alan Turing, a mathematician born in London, took up a full-time position at Bletchley Park, the Britain's codebreaking centre, where work was carried out to break the Enigma machine. Turing led Hut 8, the section responsible for German naval cryptanalysis. Building upon Rejewski and his colleagues work, Turing devised the British bombe machine, which helped tremendously with the cryptanalysis of the newest versions of the Enigma machine. Turing also made important contributions to decrypt the more complex German naval communications. In particular, Turing developed the 'Banburismus' machine, which could read Enigma naval messages. This was of extreme importance during the war, as this helped Allies shipping to stay away from the U-boat German submarines. Despite his numerous contributions to the cryptanalysis of the Enigma machine, Turing remains most known today for his influence in the development of theoretical computer science, in particular by proposing the Turing machine, which can be considered a model of a general purpose computer. Turing is also considered to be the father of theoretical computer science.

1.2 Modern cryptography

The aftermath of WWII led to significant changes in the world. In the early 60s, computer networking became a primary focus of interest. Although initially used for military purposes, its potential for enhancing day-to-day processes (e.g. bank transactions) was soon understood. The ever-increasing number of computer communication networks and the growth of the internet accentuated the need for efficient, secure communication models.

Today, cryptography is an integral part of virtually every computer system. Accessing secured websites and operating systems are examples where users rely (oftentimes unknowingly) on cryptography. Cryptography has evolved to serve ordinary people across many computing systems, thus becoming a central (and complex) topic in computer science. In this section, we provide a very brief overview of current⁴ types of cryptography and cryptanalysis. For a more comprehensive overview, see, for example [53, 12, 120].

1.2.1 Symmetric and asymmetric cryptography

Up until the 1970s, cryptography revolved around symmetric cryptography, where a secret key is shared by the parties involved in the communication, and which they use both to encrypt and decrypt messages. Thus, all parties agree on a secret key before starting to communicate. In addition, the key should not be

⁴The term "modern cryptography" is typically used to distinguish from "classical cryptography". However, in this thesis, the term is used to refer to current (forms of) cryptography.

disclosed to third parties or be obtained by eavesdroppers, as this would compromise the communication. The key advantage of symmetric cryptography is that it is typically efficient to implement. In particular, the ciphertext does not become significantly larger than the plaintext.

The Diffie-Hellman protocol

As the internet grew, a new problem emerged: How could two people who have never met, agree on a secret key in a secure way? In 1976, Diffie and Hellman published a breakthrough concept, nowadays referred to as the Diffie-Hellman (key exchange) protocol, which solves this problem [32]. We use an analogy with colours to explain their solution⁵. Given the use of colours, the initial question is now: “How can Bob and Alice agree on a secret colour without Eve finding it out?” The answer is based on two simple facts: (1) it is easy to mix two different colours to arrive at a third colour, and (2) given a colour that is the result of mixing two other colours, it is hard to reverse it to find the original colours.

Given these facts, two different parties agree on a starting colour, mix their private colours with the starting colour and send the mixture to the other party. Then, each party adds its private colour to the mixture, thus arriving at the same colour. Note that an eavesdropper could only listen to the communication and get a copy of the mixture, but it will need a private colour to arrive at the final colour. The analogy with colours is in fact an intuitive way to explain a one-way function: easy to compute, hard to reverse. To build an actual public-key cryptosystem, one needs *good* one-way functions: the harder it is to reverse it, the better it is. Section 1.2.2 expands on this topic.

In essence, the Diffie-Hellman protocol avoids the problem of secure key exchange because the private keys are never transmitted. This is also known as asymmetric or public-key cryptography. In asymmetric cryptography, users generate a public and private key-pair and make the former publicly available. This type of cryptography is secure because it is computationally impractical to determine a private key based on its corresponding public key. Thus, security comes down to keeping the private key in secrecy. Messages are then encrypted using the public key and can only be decrypted with the matching private key and the same algorithm. Compared to symmetric cryptography, both encryption and decryption require far more processing time. For instance, ciphertexts are usually considerably larger than plaintexts in these systems. It is thus common to use asymmetric systems in conjunction with symmetric systems, especially for long-lasting communications. In this set-up, an asymmetric system is first used to share the secret key of the symmetric system, which is then used for the rest of the communication.

RSA

RSA, named after its authors Ron Rivest, Adi Shamir, and Leonard Adleman, is also a public-key cryptosystem [107]. It was proposed in 1977, right after the Diffie-Hellman protocol, together with the groundbreaking concept of asymmetric public-key cryptosystems, but left open the problem of realizing a one-way function. Rivest, Shamir and Adleman made this approach practical by using the factorization of a product of two large primes as the one-way function behind RSA. RSA is currently widely used in internet protocols such as SSH, OpenPGP, S/MIME, and SSL/TLS. It is often found in many software programs and physical devices, such as key fobs. In fact, RSA signature verification is one of the most

⁵This analogy is based on information available from www.khanacademy.org

performed operations in IT⁶. In the following, we provide a very simplistic explanation of how RSA works. For detailed explanations of RSA, we refer the reader to further literature e.g. [120, 107, 53].

The foundation of RSA is that while it is easy to multiply numbers, it is hard to factor composite numbers. The system works with one public key, which can be shared with everyone, and one private key. The private key consists of two large secret primes p and q , and a secret exponent d . The public key is the composite number $N = pq$ and a public exponent e . Messages m can be encrypted by exponentiating e , i.e. the ciphertexts $c = m^e \bmod N$. Due to some mathematical properties, the receiver of the ciphertext can use its private key to compute $c^d \bmod N = m$. In fact, looking at these together, in the form of $(m^e \bmod N)^d \bmod N = m$, we can see that one is undoing the effect of the other. If an adversary is able to factor N , he can retrieve the secret primes p and q , and break the system. N is difficult to find by adversaries, as it requires prime factorization. This gets back to the fundamental theorem of arithmetic, which holds that any number greater than 1 can be written in one (and only one) way as a product of prime numbers. While it is easy to find the product of two prime numbers, it is very difficult to factor a number and find its prime factors. In Section 1.2.2, we briefly revisit this problem.

Other cryptosystems

Apart from the Diffie-Hellman protocol and the RSA cryptosystem, other cryptosystems, including the Cramer-Shoup cryptosystem, the ElGamal cryptosystem, and various cryptosystems based on elliptic curves, are also widely employed in practice. It is not the purpose of this thesis to overview all the existing schemes. To that end, the reader is referred to Chapter 7 of [53] for an explanation of El Gamal, and to [28] for an explanation of the Cramer-Shoup cryptosystem, the first efficient public-key encryption scheme secure against chosen-ciphertext attacks.

1.2.2 Cryptanalysis of modern cryptography

Modern cryptography is based on the assumption that some problems (on which cryptosystems base their security on) cannot be solved in polynomial time [53]. In Section 1.2.1, it was mentioned that the core of the Diffie-Hellman protocol is a *good* one-way function, which is to say a mathematical function that is easy to compute but hard to invert. Modular arithmetic provides us with numerical procedures that are easy to solve in one direction but hard in the other. For example, computing $5^x \bmod 27$ is easy for any given x . However, deducing x from the equation $5^x \bmod 27 = 22$ is harder and although it is easy with small bases, such as 5, it becomes impractical for prime moduli that are hundreds of digits long. This is called the discrete logarithm problem [95], and it underpins the security of the Diffie-Hellman protocol and the ElGamal public key cryptosystem.

A specific instance of the discrete logarithm problem occurs in Elliptic Curve Cryptography (ECC), whose security is based on the Elliptic Curve Discrete Logarithm Problem (ECDLP). ECC is widely accepted in the industry as a highly secure, yet relatively efficient (in terms of key-size) type of cryptography. For comprehensive instructions to elliptic curve cryptography, we refer the reader to [56, 87]. The security of ECC is based on the hardness of the ECDLP. To this day, the best methods to solve the ECDLP are variants of Pollard's rho method [100]. In 1997, Certicom⁷ introduced ECC challenges to foster investigation around elliptic curve cryptanalysis and boost the acceptance of elliptic curve cryptosystems.

⁶<http://searchsecurity.techtarget.com/definition/RSA>

⁷<https://www.certicom.com/index.php/the-certicom-ecc-challenge>

The hardest Certicom challenge ever solved was the ECCp-109 challenge (prime-field based elliptic curves), for which a cluster of 10,000 computers (mostly PCs) was used for 549 days⁸, in 1997. Other challenges were also solved at the cost of massive compute power, including an ECDLP over a 109-bit Koblitz-curve, which required 9,500 computers for 126 days. Bos et al. solved a discrete logarithm defined over a 112-bit prime-field elliptic curve, with a cluster composed of 215 Cell processors (1290 cores), in 6 months time [17]. Wenger et al. solved a 113-bit binary-field Koblitz curve with a 18-core Virtex-6 FPGA in approximately 24 days if all cores were simultaneously active [130]. GPUs were also used to accelerate the computation of discrete logarithms on special classes of moduli [47].

As mentioned before, the security of the RSA cryptosystem is based on the hardness of factoring large integers. While factoring large integers and calculating a discrete logarithm are different problems, they share some properties and can actually be solved with variants of the same algorithms. The factorization of large integers is usually referred to as a brute-force attack on RSA [14]. The most efficient algorithm to date is Pollard's General Number Field Sieve (GNFS), which is more than 20 years old. In the last 20 years, no breakthrough results were published on GNFS, except mostly for polynomial time factoring on quantum computers (cf. [116, 117]), which unveiled the vulnerability of RSA against quantum attacks. The hardness of factorization has been put to its limits over the last years. By the end of 2009, the conclusion of a GNFS factorization of a 768-bit RSA modulus was announced. It remains as the highest solved RSA modulus, and took more than two and a half years of computation on hundreds of machines [55]. GPUs were later on shown to be effective at solving the co-factorization step, a compute-intensive kernel of the GNFS algorithm, delivering speedups of around 50% [85].

As it was shown, cryptanalysis of modern cryptosystems revolves around two core problems: discrete logarithms and integer factorization. Considerable efforts have been devoted to solve these problems with as many bits as possible, which is to say the hardest case scenarios, on high end modern computer architectures. There are two important facts that should be observed. First, the actual security of cryptosystems is usually determined empirically, to which end the best known attacks are implemented and put to the test. In this process, it is also common to use as many and as powerful hardware resources as possible e.g. [55]. When it is only possible to use limited computing resources or limited parameters, it is common that extrapolations are made to larger computing units and larger parameters e.g. [27, 16]. The empirical benchmarks are used to extrapolate the power of the attacks when implemented on high end computer architectures. So, progress on cryptanalysis evolves both with algorithmic improvements and efficient implementations on high-end architectures.

1.2.3 Somewhat/fully homomorphic encryption

This section is based on [77].

Oftentimes, it is desirable to perform operations on encrypted data. For example, imagine a scenario with sensors that collect data from a patient, and due to its volume, the data has to be processed by a third party, say a cloud provider. In addition, the third party is either untrusted or might be attacked, exposing the data therein. In such a scenario, it is desirable that the data is sent encrypted to the third party that will process it without the need to decrypt it first. The solution to this problem is Homomorphic Encryption (HE). Systems that implement HE partially (e.g. they are capable of summing encrypting data but not

⁸<https://www.certicom.com/news-releases/340-notre-dame-mathematician-solves-eccp-109-encryption-key-problem-issued-in-1997>

multiplying it) are said to enjoy Somewhat Homomorphic Encryption (SHE) or Partially Homomorphic Encryption (PHE), while Fully Homomorphic Encryption (FHE) schemes can implement any operation on encrypted data.

The idea of creating a homomorphic encryption scheme was first suggested by Rivest, Adleman and Dertouzos [106], back in 1978. In a truly breakthrough work, in 2009, Gentry described the first plausible construction for a FHE scheme [39]. Using ideal lattices, Gentry showed that it is possible to construct a FHE scheme by first producing a somewhat homomorphic encryption scheme and then applying a bootstrapping process to obtain a complete FHE scheme. Multiple FHE schemes were presented since then, most of them focusing on improving efficiency. Although Gentry's result represented one of the most significant theoretical progresses in cryptography in the last decades, FHE schemes are only practical in restricted scenarios. However, this is bound to change in the near future. FHE is especially relevant in cloud computing, which has been massively adopted, for both business and private use. Thus, it is expected that FHE takes off, driven by the growth of cloud computing and its wide adoption. The purpose of this section is to briefly report on the importance of FHE, which is also enabled by (ideal) lattices. For a comprehensive review of (fully) homomorphic encryption schemes or their applications, the reader is referred to [77].

1.3 Quantum-immune cryptography

In the mid-nineties, the news broke that several classical cryptosystems, such as RSA, were insecure against quantum computers [114]. This was due to a breakthrough work of Shor on quantum algorithms for problems that underpin the security of classical cryptosystems, such as factoring large numbers, which become practical in the presence of large-scale quantum computers [116, 117]. This was a landmark in the history of cryptography, as it marks the beginning of a new era: the post-quantum (in this thesis referred to as quantum-immune) era. Soon after, the cryptography community sprang to the challenge of finding cryptosystems resistant against attacks operated with quantum computers, or quantum-immune cryptosystems. In the late 90s, Ajtai discovered that certain lattice problems have interesting properties for cryptography, such as worst/average-case hardness, and that lattices can be used for building cryptosystems [3]. Due to Ajtai's seminal work, lattice-based cryptography broke out in the mid-nineties as a very promising candidate to post-quantum cryptography, as many researchers set out to investigate lattice-based alternatives for classical schemes. This thesis also revolves around this topic, contributing to strengthen the knowledge one has of lattice-based cryptography.

1.3.1 Lattice-based cryptography

Lattices are discrete subgroups of the n -dimensional Euclidean space \mathbb{R}^n , with a strong periodicity property. A lattice \mathcal{L} generated by a basis \mathbf{B} , a set of linearly independent vectors $\mathbf{b}_1, \dots, \mathbf{b}_m$ in \mathbb{R}^n . In Chapter 2, lattices as well as lattice problems are presented in detail, along with the necessary notation. The roots of lattices trace back to the eighteenth century, when they were studied by renowned mathematicians such as Lagrange and Gauss.

Lattice-based cryptography stands out as the most prominent and rapidly growing field of quantum-immune cryptography, for several reasons. First, lattice-based cryptosystems are quite efficient and typically very simple to implement. Second, it is known that lattice-based cryptosystems enjoy worst-

case hardness, a powerful property for cryptosystems. Roughly speaking, this means that breaking the cryptosystem is (provably) at least as hard as solving several lattice problems in the worst case. In other words, breaking the cryptosystem is equal to solving a lattice problem that underpins the security of the cryptosystem, which is covered in detail in the next subsection. The reason why worst-case hardness is a distinguishing feature of lattice-based cryptography, is because most cryptographic constructions are based on average-case hardness⁹ [81]. Third, lattices can be used to implement FHE schemes, a pivotal technique for cryptosystems that was introduced in Section 1.2.3.

Public-key encryption lattice-based schemes

Although the goal of this thesis is to study cryptanalysis of lattice-based cryptosystems, it is still relevant to briefly mention some lattice-based cryptosystems currently in use, the advances in this area, and the current performance of these systems. For a comprehensive survey in this topic, the reader is referred to the surveys [97, 81], which this section is based on. Among the most important lattice-based cryptosystems are the NTRU [50], GGH [41], Ajtai-Dwork [4] and LWE [105] (extended later on to Ring-LWE [69]) cryptosystems.

NTRU. Proposed by Hoffstein, Pipher and Silverman in 1998¹⁰, NTRU (also known as NTRUEncrypt) is one of the most important lattice-based cryptosystems [50]. There is relatively little theoretical understanding of the NTRU cryptosystem and its associated average-case computational problems [97], although a variant of the NTRU cryptosystem has been proven secure [122] and NTRU has, in general, withstood significant cryptanalytic efforts [97]. In practice, the NTRU cryptosystem is efficient, exhibiting compact keys [97].

GGH. GGH, proposed by Goldreich, Goldwasser, and Halevi in 1997, is essentially a lattice analogue of the McEliece cryptosystem, which is based on the hardness of decoding linear codes over finite fields [41]. The cryptosystem does not have worst-case security guarantees, and it has been cryptanalyzed for practical parameter sizes, but not broken asymptotically, in 1999 [88]. The cryptosystem did also lay the foundation for other cryptographic constructions, most notable trapdoors, which admit security proofs under worst-case hardness assumptions (cf. Sections 5.4 and 5.5 of [97]).

Ajtai-Dwork. The Ajtai-Dwork lattice-based cryptosystem builds up on the groundbreaking result of Ajtai, who showed in 1996 the first worst-case to average-case reductions for lattice problems [4, 2]. Although the system represented a breakthrough in the field of lattice-based cryptosystems, given that it was the first scheme with a security proof under a worst-case complexity assumption, some downsides on the practical side are commonly noted, as the public keys are of size $\mathcal{O}(n^4)$, and its secret keys and ciphertexts are of size $\mathcal{O}(n^2)$, with similarly quadratic encryption and decryption timings. In 2004, Regev proposed several improvements to the Ajtai-Dwork cryptosystem [104]. Although greatly simplifying the implementation and analysis of the cryptosystem, the size of the public keys, secret keys and ciphertexts remained the same.

LWE-based. Regev proposed the first Learning With Errors (LWE)-based public-key encryption scheme, i.e. a cryptosystem whose security is based on the LWE problem, with a certain error [105]. Public keys are $\mathcal{O}(n^2)$ bits long, secret keys and ciphertexts are $\mathcal{O}(n)$ bits long and each ciphertext encrypts a single bit. In this context, n is the dimension of the underlying LWE problem. Later on, in 2008, Gentry,

⁹“For instance, breaking a cryptosystem based on factoring might imply the ability to factor some numbers chosen according to a certain distribution, but not the ability to factor all numbers” [81].

¹⁰Although this work has been published in 1998, it has indeed been completed in 1996.

Peikert, and Vaikuntanathan proposed a LWE-based public-key encryption cryptosystem, which can be viewed as a dual of Regev’s cryptosystem [97]. In 2010, Lyubashevsky, Peikert, and Regev proposed Ring-LWE, the ring-based analogue of learning with errors [69]. It was also shown that Ring-LWE is at least as hard as the NTRU learning problem, for appropriate parameters. Other public-key cryptosystems based on the LWE followed (cf. Sections 5.2.3 and 5.2.4 of [97]).

Performance considerations. Some work has been done to assess the performance of these cryptosystems, both in the context of the post-quantum realm and against classical (current) schemes. Regarding the comparison between quantum-immune and classical (current) schemes, Challa et al., have shown, in 2007, that a NTRU implementation is more efficient than a RSA implementation (for several numbers of bits between 128 and 10K), thereby suggesting that NTRU is sufficiently efficient to admit immediate use [19]. In 2009, Wu et al. showed, via a mobile java emulator, that NTRU is 200x more efficient than RSA (NTRU-251 vs RSA-1024) on mobile phones, and it works well on limited computing capability environments [131]. Another study in this direction has shown that NTRU compares very well against RSA (NTRU-256 vs RSA-1024 and RSA-2048) and ECC (NTRU-256 vs ECC NIST-224), when implemented on GPUs [48].

The inner workings of these cryptosystems are well covered and explained in the “Post quantum cryptography” survey, from 2009 [81]. Note also that the first open question identified in this survey (Section 8, “*Cryptanalysis*”), stating that “*more work is still needed to increase our confidence and understanding, and in order to support widespread use of lattice-based cryptography*“, resonates well with the goal of this dissertation, which is to increase the understanding of the possible computational performance of lattice-based cryptanalysis on current HPC systems.

1.3.2 Lattice-based cryptanalysis

Section 1.2.2 provided an overview of modern cryptanalysis, which primarily revolves around problems such as the discrete logarithm and factoring large integers. Lattice-based cryptosystems also base their security on hard mathematical problems. One of these hard problems is to find the shortest vectors in a given lattice, which is commonly referred to as the Shortest Vector Problem (SVP). This can be seen as an analogue of the discrete logarithm and integer factoring problems in modern cryptanalysis. Other problems that underpin the security of lattice-based cryptosystems include the Closest Vector Problem (CVP) and the Learning With Errors (LWE) problem.

Although only a very short¹¹ (and not necessarily the shortest) vector is needed to break a cryptosystem, the SVP is usually the central problem in this context, as algorithms that find very short vectors in a lattice usually use SVP algorithms in their workflow. The SVP consists in finding the non-zero vector \mathbf{v} of a given lattice \mathcal{L} , whose Euclidean norm $\|\mathbf{v}\|$ is the smallest among the norms of all non-zero vectors in the lattice \mathcal{L} and is denoted by $\lambda_1(\mathcal{L})$. It is well known that the SVP is NP-hard under random reductions, so no polynomial time exact algorithms are expected to be found. From here forward, we refer to an algorithm that solves this problem as an *SVP-solver*.

A crucial aspect of SVP-solvers (and therefore lattice-based cryptanalysis) is that they are faster on reduced lattice bases. Lattice basis reduction is the process of transforming a given lattice basis \mathbf{B} into another lattice basis \mathbf{B}' , whose vectors are shorter and more orthogonal than those of \mathbf{B} and where \mathbf{B} and

¹¹To attack LWE-based schemes, one has to solve the LWE problem, which in turn can be solved by solving a relaxed version of the SVP, denoted by α -SVP, as the e.g. “distinguishing attack” does e.g. cf. [6, 66].

\mathbf{B}' generate the same lattice, i.e., $\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{B}')$. The most practical algorithms for lattice basis reduction are the Lenstra-Lenstra-Lovász (LLL) and the Block Korkine-Zolotarev (BKZ) algorithms (cf. [65, 114]).

The LLL algorithm was the first tractable algorithm to reduce bases. It lays the foundation for many algorithms for problems on lattices, including BKZ. LLL has applications in many fields in computer science, ranging from integer programming to cryptanalysis [93]. LLL is a fast, polynomial-time algorithm that offers a moderate guarantee on the quality of the output basis. BKZ is a generalization of LLL, and offers an adjustable trade-off between the time complexity and the output quality, through a block-size parameter β : the higher the block-size β , the longer the algorithm takes to terminate, but the probability of obtaining a basis with better quality is higher. BKZ uses an SVP-solver in dimension β . As SVP-solvers have exponential time complexity, they become the dominant kernel of the algorithm for high values of β , both in terms of time consumption and impact on the solution.

There are several different classes of SVP-solvers. Enumeration algorithms were the first and probably the most studied SVP-solvers to date (see [61, 43] for a comprehensive overview). Sieving algorithms attracted increasing attention from the community since 2010, and became competitive with enumeration algorithms recently. The work in this thesis has contributed to this, as it presents techniques to implement scalable sieving algorithms efficiently. Currently, random sampling algorithms are presumably the fastest SVP-solvers, as they rank first the online SVP-Challenge¹². Another approach that has been shown effective at solving the SVP in high dimensions is a combination of enumeration-based algorithms and efficient lattice reduction algorithms. There are other classes of algorithms, such as algorithms based on the Voronoi cell of a lattice [1, *Relevant vectors*, Section VI C]. Although this class of algorithms currently offers the best theoretical time-complexity bounds, its algorithms are intractable in practice. In Chapter 3, these classes of SVP-solvers are covered in detail and it is motivated why, in our studies, we concentrated on sieving algorithms, and, to a lesser extent, on lattice basis reduction and enumeration algorithms.

The knowledge of the security of lattice-based cryptography is, in contrast to that of modern cryptography, limited. Although the cryptography community has developed a considerably solid theoretical knowledge of these algorithms, their practicability is still not well understood. A good example of this is that some algorithms perform way worse in practice than what is expected in theory, and vice versa. However, this understanding is of prime importance, because the parameters of cryptosystems (e.g. key sizes) are chosen based on the tractability of the best attacks (such as SVP-solvers in the context of lattice-based schemes). As explained in Section 1.2.2, the choice of the parameters of a cryptosystem is an empirical process: the best attacks are implemented in practice and then a *safe gap* is left such that the parameters of the cryptosystem are unreachable by any attack. Therefore, if the tractability of the attacks is miscalculated, either overly strong or weak parameters are selected, thus rendering the cryptosystems impractical or insecure, respectively. This is the core reason why highly optimized, parallel solvers of the underlying lattice problems of lattice-based cryptosystems have to be developed and studied.

1.4 Contributions, structure and technical details of the thesis

The work in this thesis contributes to the understanding of the practicability of SVP-solvers and lattice basis reduction algorithms on modern computer architectures. As a result, lattice-based cryptanalysis is assessed mainly from a computational engineering, rather than a mathematical, standpoint. In particular,

¹²www.latticechallenge.org/svp-challenge/

the contributions of the thesis include, among others:

1. Methods to parallelize enumeration algorithms, instantiated over the SE++ and ENUM algorithms, in Chapter 4.
2. A cache efficient, vectorized LLL implementation, in Chapter 5.
3. A parallel implementation of BKZ, including extensive benchmarks, allowing to understand the scalability of the BKZ algorithm, in Chapter 5.
4. Various scalable parallelization schemes for sieving algorithms, in Chapter 6.
5. Optimization techniques for these algorithms, especially regarding memory access, in Chapter 6.

This dissertation is structured in seven chapters. Each of the chapters is centered on a specific topic and includes an abstract, describing the contributions of the chapter, in the beginning.

Chapter 2 provides the preliminaries, such as notation and description of problems, for the comprehension of the dissertation. In particular, it introduces the concept of lattice, its notation and connection to cryptography. Moreover, some notions on computer architecture and parallel computing, essential for the comprehension of the document, are presented.

Chapter 3 provides a comprehensive collection of the algorithms and implementations used in lattice-based cryptanalysis. The algorithms are briefly explained and categorized, and existent implementations are pointed out. It is also stated what algorithms and implementations are important for this thesis, and in which chapter they are covered on.

Chapter 4 walks through enumeration routines, a fundamental class of algorithms to solve the SVP and the CVP. Some enumeration-based algorithms are introduced and their parallelization is described. Two different parallelization methods, based on OpenMP and POSIX threads, are provided. Practical experiments are reported and analyzed.

Chapter 5 deals with the primary concept in lattice theory, lattice basis reduction. The chapter introduces the concept of lattice basis reduction, and LLL and BKZ, the two fundamental algorithms in this context, and some of their variants. Practical experiments of novel parallel, highly efficient implementations of these algorithms are presented, alongside a comprehensive set of benchmarks that allow one to understand the performance of these algorithms on parallel architectures.

Chapter 6 is the largest chapter of this thesis, and it comprises results with sieving algorithms. In particular, the chapter describes (i) heuristics to speed up sieving algorithms in practice, (ii) highly scalable mechanisms to parallelize sieving algorithms on shared-memory systems and (iii) memory issues of sieving algorithms, and mechanisms to address them.

Chapter 7 concludes this dissertation and provides future lines of work.

Preliminaries

Synopsis. This chapter aims at providing the reader with the necessary information to fully comprehend this dissertation. Section 2.1 introduces the concept of lattices, from a very practical point of view. Section 2.1.1 introduces the notation, definitions and some relevant lattice properties and operations used throughout this dissertation. Section 2.2 provides a brief overview of parallel computing, also showing the test platforms where the tests in this dissertation were performed.

"Life is the only art that we are required to practice without preparation, and without being allowed the preliminary trials, the failures and botches, that are essential for training." - Lewis Mumford, American historian and writer.

2.1 Lattices

It is commonly argued that the roots of lattices date back to the 18th century, when remarkable mathematicians such as Lagrange and Gauss used lattices in number theory. However, it was not until the early 1980s that the computational aspects of lattices were studied. This was driven by the use of lattices in cryptography, which although not obvious [81], Ajtai shown possible in a breakthrough work in [3]. His result has sparked a whole new area of research, which has evolved considerably in the last decades and continues to rapidly expand today. This chapter presents some notation and important definitions pertaining to lattices, and some of the most important lattice problems on lattices.

2.1.1 Notation and definitions

Let \mathbb{R}^n be a n -dimensional Euclidean vector space. Vectors and matrices are written in bold face (or italic if represented with Greek letters), vectors are written in lower-case, and matrices in upper-case (or lower-case if represented with Greek letters), as in vector \mathbf{v} and matrices \mathbf{M} and μ . Vectors (also referred to as lattice points or simply points) in \mathbb{R}^n represent $1 \times n$ matrices. The Euclidean norm (or length) of a given vector \mathbf{v} in \mathbb{R}^n , $\|\mathbf{v}\|$, is $\sqrt{\sum_{i=1}^n v_i^2}$, where v_i is the i^{th} coordinate of \mathbf{v} . The term *zero vector* is used for the vector whose norm is zero, i.e., the origin of the lattice. The dot product of two vectors \mathbf{v} and \mathbf{p} is denoted by $\langle \mathbf{v}, \mathbf{p} \rangle$. The angle between two vectors \mathbf{v} and \mathbf{p} is denoted by $\phi_{\mathbf{v}, \mathbf{p}}$ and it holds that $\phi_{\mathbf{v}, \mathbf{p}} \in [0, \pi)$. For $1 \leq i$, we denote by $\pi_i(x)$ the orthogonal projection of x onto the space spanned by a basis $\mathbf{b}_1, \dots, \mathbf{b}_{i-1}$.

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

$$\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{b}_1, \dots, \mathbf{b}_m) = \left\{ \mathbf{x} \in \mathbb{R}^n : \mathbf{x} = \sum_{i=1}^m x_i \mathbf{b}_i, \mathbf{x} \in \mathbb{Z}^m \right\}. \quad (2.1)$$

where m is the *rank* of the lattice. When $n = m$, the lattice is said to be of *full rank*. When n is at least 2, each lattice has infinitely many different bases. Graphically, a lattice can be described as the set of intersection points of an infinite, regular n -dimensional grid. Figure 2.1 shows a lattice, with both n and m equal to 2, and its basis $(\mathbf{b}_1, \mathbf{b}_2)$ in red. The vector \mathbf{b}_3 , in blue, is the result of the linear combination $1 \times \mathbf{b}_1 - 2 \times \mathbf{b}_2$, and it is represented to show how a lattice vector can be obtained at the cost of a linear combination of the basis vectors. Also note that $(\mathbf{b}_2, \mathbf{b}_3)$ also forms a basis of the same lattice, but with shorter and more orthogonal vectors.

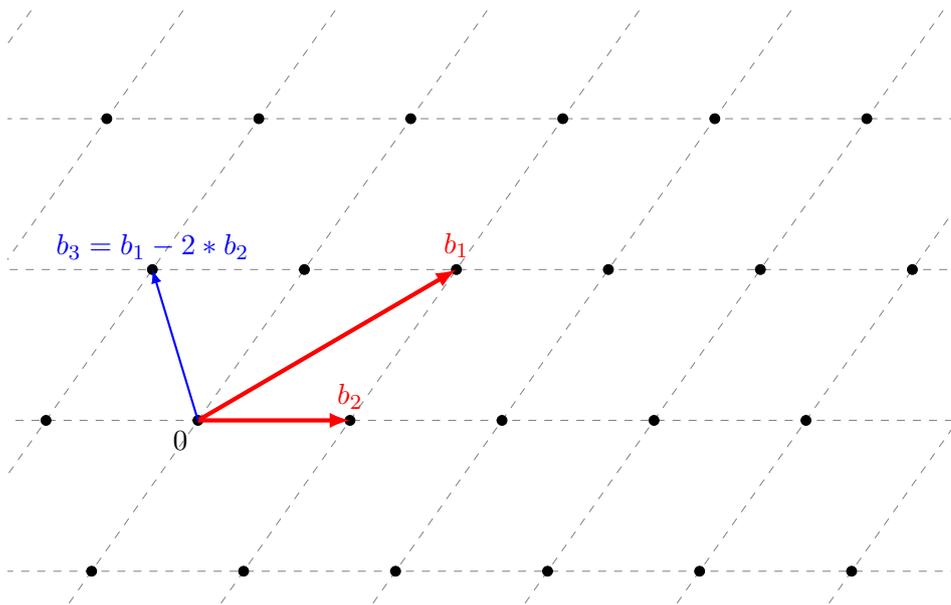


Figure 2.1: Example of a lattice in \mathbb{R}^2 and its basis $(\mathbf{b}_1, \mathbf{b}_2)$ in red.

For a comprehensive compilation about lattices and their properties, the reader is referred to [23, 80, 127].

Common concepts

We now present some core concepts to understand lattice basis reduction and the SVP, and some operations underpinning the workflow of algorithms that solve these problems.

Vector reduction. Rendering a given vector \mathbf{v} smaller using another vector \mathbf{p} is usually referred to as “reduce \mathbf{v} against \mathbf{p} ”. If $\|\mathbf{v} \pm \mathbf{p}\| \leq \|\mathbf{v}\|$, then the reduction is successful. Evaluating $\|\mathbf{v} \pm \mathbf{p}\| \leq \|\mathbf{v}\|$ implies that the reduction is actually made so it is possible to verify whether the vector became smaller. However, through the definition of the inner product $\|\mathbf{v} \pm \mathbf{p}\| = \langle \mathbf{v} \pm \mathbf{p}, \mathbf{v} \pm \mathbf{p} \rangle$, it is possible to conclude that if $2 \times |\langle \mathbf{v}, \mathbf{p} \rangle| > \|\mathbf{v}\|^2$, the reduction is successful [127]. In fact, the latter inequality, which involves primarily the computation of an inner product, is the fundamental operation of the “reduction” kernel in practice.

Orthogonality. Two vectors \mathbf{v} and \mathbf{p} are orthogonal when $\phi_{\mathbf{v},\mathbf{p}} = 90^\circ$, and $\langle \mathbf{v}, \mathbf{p} \rangle = 0$. The concept of orthogonality is important as there are many properties that are based on this concept. For example, as mentioned before, the goal of lattice basis reduction algorithms is to output a basis with as short and as orthogonal vectors as possible¹. The orthogonality of vectors is also important in the process of vector reduction.

Gram-Schmidt Orthogonalization. The Gram-Schmidt orthogonalization process turns a set of vectors into another set of vectors that are pairwise orthogonal and span the same space. Nevertheless, the final result is not necessarily a lattice basis, as this process may use non-integral coefficients. The Gram-Schmidt vectors are very important as they are required to calculate the orthogonality defect of a given basis. The orthogonality defect, in turn, indicates the proximity (in relative terms) between the basis vectors and the corresponding Gram-Schmidt vectors. An orthogonality defect of 1 means that the basis vectors are equal to the Gram-Schmidt vectors, and hence pairwise orthogonal.

Size-reduction. There are several different notions of a size-reduced lattice basis. Hermite was the first to provide a notion of size-reduction [49]. According to Hermite, a given lattice basis is size reduced if its Gram-Schmidt coefficients μ_{ij} satisfy $|\mu_{ij}| \geq \frac{1}{2}$. This notion is, however, relatively weak, as it is easy to achieve even for *not so reduced lattice bases*. A representative example are Goldstein-Mayer lattices, which are used in the SVP-Challenge², and throughout this thesis. Size-reducing Goldstein-Mayer lattice bases only guarantees that the Gram-Schmidt norms are the volume of the lattice for the first vector and 1 for the other vectors. Although a size-reduced basis has important properties for specific applications, they are not necessarily good bases in the context of the SVP and CVP. As it is shown in this thesis, the reduction of these lattice bases is crucial to accelerate SVP- and CVP-solvers. Moreover, some vectors in these lattice bases have coordinates with hundreds of digits, which requires a multiple precision module to handle them. This problem is addressed in Section 3.3.1.

Random and ideal lattices. Integer lattices are additive subgroups (meaning that the sum of any two elements, i.e. lattice vectors, is also an element of the group) of \mathbb{Z}^n . There are non-integer lattices as well, but in lattice-based cryptography, one works with integer lattices as solving integer lattice problems is generally as hard as solving non-integer lattices, but integer lattices are easier to handle computationally (e.g. no precision problems arise). There are specific types of lattices, with additional structure. One particularly relevant example are ideal lattices. They are important both in lattice-based cryptography, and while they have interesting properties for schemes (e.g. they have smaller key sizes), and, as a result, some algorithms can take advantage of their additional structure when solving certain problems, such as the SVP. This is addressed in Section 3.3.2, Chapter 3, although the contributions of this thesis do not pertain to ideal lattices.

Quality of lattice bases

This dissertation presents various experiments with SVP- and CVP-solvers, whose performance is affected by the quality of the underlying lattice bases. The quality of lattice bases is also a fundamental tool to assess the performance of lattice basis reduction algorithms. In view of that, it is important to define the quality of a given basis, and appropriate criteria to measure it. However, depending on the application domain, the definition of a good lattice basis may differ. In this thesis, lattice bases are assessed from

¹Note that unlike bases of vector spaces, lattices typically do not have orthogonal bases.

²www.latticechallenge.org/svp-challenge/

a cryptanalysis point of view. In particular, a *good basis* is one that renders SVP- and CVP-solvers faster, and contains the shortest possible vectors, among other factors. Even for this narrowed context, there is no *de facto* standard on how to measure the quality of a basis and measurement methodologies often differ from study to study [132, 21, 102, 126], although the Hermite factor/defect, the length defect and the orthogonality defect are among the most used ones (the three of which are small on good bases [20, 132, 21]). In this thesis, we use an extension of previous methodologies, which include the following criteria:

1. The sequence of Gram-Schmidt norms $\|\mathbf{b}_1^*\|, \dots, \|\mathbf{b}_n^*\|$, which decreases slowly for good bases [20].
2. The norm of the last Gram-Schmidt vector, which should be as large as possible.
3. The average of the norms of the basis vectors, which should be as small as possible (the product of the norms is encapsulated in the orthogonality defect).
4. Execution time of SVP- and CVP-solvers on the lattice.

To the best of our knowledge, there are no studies correlating the quality of the basis with the execution time of SVP- and CVP-solvers, other than [25], which, in collaboration with us, used the same criteria for assessing the quality of lattice basis. Section 5.2.2 expands on this topic, presenting empirical benchmarks measuring the quality of different bases, in the context of the SVP and CVP.

2.1.2 Lattice problems

This section introduces some of the most relevant problems on lattices, with applications on cryptography. In particular, it introduces the problem of reducing a given lattice basis, the Shortest Vector Problem (SVP) and the Closest Vector Problem (CVP). There are many other important problems, such as Learning with Errors (LWE) and the Shortest Integer Solution (SIS) and for a complete introduction of these problems, the reader is referred to the surveys [81, 103, 79].

Lattice basis reduction

As mentioned before, lattice basis reduction algorithms aim at improving the lattice basis, by rendering its vectors shorter and more orthogonal. It is informally said that lattice basis reduction is also the problem of searching for superior bases of a given lattice, given one that is not so good. This problem is very well studied, and there are several algorithms that are efficient in practice. In the context of this thesis, LLL and BKZ are the most important algorithms (see Section 3.2). The main reason why lattice basis reduction is so important in the context of lattice-based cryptanalysis is because the lattice basis influences the performance of algorithms for crucial problems such as the SVP and the CVP. Moreover, approximate versions of the SVP e.g. the α -SVP (which suffice to break some cryptosystems) can also be solved with lattice basis reduction algorithms.

The Shortest Vector Problem

The norm of a shortest vector³ of a lattice is denoted by $\lambda_1(\mathcal{L})$. The norm of the shortest vector in the lattice is also the minimal distance between any two vectors in the lattice. Formally, the SVP can

³Note that due to the natural symmetry in lattices, there is not only one shortest vector.

be defined as: given a basis \mathbf{B} of the lattice \mathcal{L} , find a non-zero vector $\mathbf{p} \in \mathcal{L}$ such that: $\|\mathbf{p}\| = \min \|\mathbf{v}\| : \mathbf{v} \in \mathcal{L}(\mathbf{B}), \|\mathbf{v}\| \neq 0$. This is typically called the exact version of the SVP problem, as there are many approximate versions of the problem. For a comprehensive review of the derivatives and approximate versions of the SVP, the reader is referred to [81]. Note, however, that the problem does not state anything about the basis, but as it will be shown later on, the type of basis that is used to solve the problem has a big impact on the practical performance of SVP-solvers. Emde Boas showed, in 1981, that the SVP with infinity norms is NP-hard [13]. In 1998, Ajtai showed that the SVP is NP-hard under randomized reductions for the Euclidean norms as well [3]. The Ajtai-Dwork cryptosystem bases its security on the γ -Unique SVP, a problem that derives from the SVP [4].

The Closest Vector Problem

Formally, the CVP can be defined as: given a basis \mathbf{B} of the lattice \mathcal{L} , and a target vector $\mathbf{v} \in \mathcal{L}$, find a vector \mathbf{p} that is closest to \mathbf{v} , i.e. such that: $\mathbf{p} = \min \|\mathbf{v}-\mathbf{w}\| : \mathbf{w} \in \mathcal{L}(\mathbf{B})$. Like the SVP, the CVP also has derivative problems and approximate versions. These problems are also overviewed in [79, 81], but are not further overviewed in this thesis as CVP represents a small part of the body of work conducted. Arora et al. have shown that the CVP is NP-hard to approximate within any constant factor [9]. Goldreich et al. have shown, through a reduction from the SVP to the CVP, that the hardness of the CVP implies the same hardness for SVP [42]. Cryptosystems whose security is based on the hardness of the CVP include, among others, the Goldreich-Goldwasser-Halevi (GGH) cryptosystem [41].

2.2 Parallel computing

Parallel computing is a broad area of computer science that has been regaining considerable traction over the past decade, due to the reappearance of (massively) parallel architectures. Up until the early 2000's, processor performance increased steadily due to higher clock-rates, made possible by the advances in the microprocessor industry, as exemplified by Moore's law. In the early 2000's, however, clock rate frequencies flat-lined, due to physical reasons such as heat dissipation, and the leading processor manufacturers started to offer multi-core processors, chips containing two or more (simpler) computing units.

There are many forms of parallelism in computers, ranging from instruction level parallelism to parallel computation of programs in clusters of thousands of multi-core computing nodes. The most common parallel programming paradigms are shared- and distributed-memory. In a shared-memory environment, there is a single, global memory address space, simultaneously visible to and accessible by many threads of computations. In distributed-memory environments, on the contrary, there are multiple memory spaces, each of which is visible to a single process. In the next subsections, we address the most relevant aspects of parallel programming both in shared- and distributed-memory models. Readers who want to read more about parallel computing are referred to additional literature e.g. [57, 96].

2.2.1 Shared-memory

In the shared-memory model, the memory address space is shared between multiple threads of computation. Threads communicate with one another by writing and reading data from the shared memory. As different threads access the same positions in memory, some kind of synchronization is required to guarantee that

no race conditions arise. Race conditions happen when two or more threads read (and at least one writes) the same memory location and the output is unpredictable. A trivial example is to have two different threads executing the operations $+ = 5$ and $\times = 2$, respectively, on a given variable $x = 10$. If the first thread is executed before the second thread, the final output will be $x = 30$, whereas if the second thread is executed first, the result will be $x = 25$.

There are several synchronization mechanisms between threads, on shared-memory, including mutexes (aka locks) and barriers. A mutex moderates the access to shared resources by allowing only one thread to access them at a given time. Mutex is the short form of “mutual exclusion”. Barriers are mechanisms where threads block at, until all the running threads reach it, which results in the release of all the threads attached to the barrier. Synchronization mechanisms can lead to a situation known as deadlock. Deadlocks occur when at least one pair of threads is mutually waiting for the other thread to release a given resource. Synchronization mechanisms must thus be used wisely, since deadlocks are easy to create but difficult to debug.

Parallelism can be attained at different, both higher and lower levels. An example of a higher level of parallelism is the execution of multiple processes on a message passing fashion, which we cover in subsection 2.2.2, where each process runs a set of threads that share memory. Instruction level parallelism or vectorization serve as excellent examples of parallelism achieved below the thread level.

Scheduling is a very relevant issue in shared-memory programming. OpenMP⁴ provides programmers with different schedulers, each of which appropriate for different work-distribution scenarios. For example, “static” is a scheduler that distributes the load evenly, before a loop execution. Dynamic and guided, on the other hand, distribute the load at many points in time, and thread assignment may also vary.

2.2.2 Distributed-memory

Programming models for distributed-memory are based on message passing. In this model, each process has access to a local memory and communicates with the remaining processes by explicitly sending and receiving messages. Messages can be sent synchronously or asynchronously. In synchronous messages, the sending and receiving processes block until the message is transmitted. In asynchronous messages, the process continues its work-flow. The programmer must make sure that data that is to be received is never accessed before it was indeed received, among other things.

A very important aspect of parallel programming models is that, although both shared- and distributed-memory can be used, in principle, regardless of the underlying architecture, it should be noted that a shared-memory model requires additional software (e.g. ScaleMP) to run on distributed-memory architectures, which is usually referred to as Distributed-Shared Memory (DSM), and performance penalties may incur. As a result, MPI⁵ is the *de facto* programming model when a given application is to run on a big distributed-memory computing environment, such as a big cluster. The choice of the programming model depends on the algorithm in question, and, in particular, its communication patterns, load profile and computational intensity i.e. the ratio of operations to data accesses.

⁴www.openmp.org/

⁵<http://www.mpi-forum.org/>

2.3 Test environment

Different machines were used throughout this thesis, as the complete set of experiments was performed over a span of three years. Table 2.1 shows the specification of each of these machines.

Codename	Peacock	Lichtenberg	Lara	Adriana
#Sockets	2	8	2	4
CPU manufacturer	Intel	Intel	Intel	Intel
Model number	E5-2670	E7-8837	E5-2698v3	X7550
Launch date	Q1'12	Q2'11	Q3'14	Q1'10
Micro-architecture	Sandy Bridge	Nehalem-C	Haswell	Nehalem
Frequency	2600 MHz	2667 MHz	2300 MHz	2000 MHz
Cores	8	8	16	8
SMT	Hyper-threading	Not available	Hyper-threading	Hyper-threading
L1 Cache	8 × 32 kB iC+dC	8 × 32 kB iC+dC	16 × 32 kB iC+dC	8 × 32 kB iC+dC
L2 Cache	8 × 256 kB	8 × 256 kB	16 × 256 kB	8 × 256 kB
L3 Cache	20 MB shared	24 MB shared	40 MB shared	18 MB shared
System memory	128 GBs	1 TB	756 GBs	128 GBs
Owner	ISC	HRZ	ISC	GPPD
Location	Germany	Germany	Germany	Brazil

Table 2.1: Specifications of the test platforms. SMT stands for Simultaneous Multi-Threading, iC/dC for instruction/data cache.

State of the Art of Lattice-based Cryptanalysis

Synopsis. Chapter 1 showed the role of cryptanalysis in cryptography in general, and lattice-based cryptanalysis in lattice-based cryptography in particular. This chapter provides a view of the current state of the art of lattice-based cryptanalysis, primarily from a practical standpoint. It presents the main problems in lattice-based cryptanalysis, and the algorithms that are used to solve these problems. Then, the main available implementations of these algorithms are identified and elementary benchmarks with the main libraries are conducted for the purpose of defining reference performance markers. The final part of the chapter relates the main contributions of this thesis to the state of lattice-based cryptanalysis.

"If it is not worse, it is likely better!" - Thijs Laarhoven, Dutch researcher. (This was Thijs reaction while discussing heuristics for HashSieve)

3.1 The landscape of lattice-based cryptanalysis

Lattice-based cryptanalysis started in the early eighties, with the appearance of LLL, which, among other applications, was used to break several cryptosystems [65, 64]. As Ajtai showed that certain lattice problems have interesting properties for cryptography and classical schemes are vulnerable against quantum attacks, the community intensified the work both on developing new schemes and assessing the hardness of the lattice problems underpinning such schemes. Among these problems, we can point out the SVP and the CVP, as well as their variants.

Although the cryptography community has developed a vast knowledge of the algorithms that solve these problems (especially the SVP) from a theoretical standpoint, the performance of these algorithms is yet not well understood in practice. For instance, the best algorithms in theory (i.e. those with lower asymptotic complexity), oftentimes perform worse than their counterparts in practice. While this gap in knowledge is extended to other application domains, it is especially problematic in cryptography, as the security parameters of cryptosystems are set based on how powerful attacks are in theory¹. In particular, both over and underestimates of the practical performance of attacks are problematic. Overestimates

¹This is also true for classical cryptosystems, such as RSA, as security parameters (e.g. the key length) are defined based on what one expects an attacker may do, with the best algorithm on a powerful computer architecture.

(i.e. estimating that the performance of attacks is higher than it actually is) compel cryptographers to use overly strong security parameters, which decreases the efficiency of the scheme and may ultimately render it impractical. On the other hand, underestimates also misguide cryptographers in selecting parameters, but in this case they may render the schemes insecure.

This is the core reason why highly optimized, scalable parallel algorithms for these problems have to be developed and their practical performance on high end modern computer architectures has to be understood. In other words, one has to know “how far can attacks really go” (for lattice-based cryptosystems, one can think of the highest possible lattice dimension on which one can solve the SVP). This is the only way to estimate the actual hardness of these problems and the actual performance of attacks. Such an assessment can only be rigorous when one has simultaneous knowledge of the algorithm and the underlying computer architecture, as some algorithms have specific properties that can be taken advantage of, on certain computer architectures. In fact, this led to the creation of the SVP-Challenge², which is a great tool to announce the latest results on the actual hardness of hard lattice problems. The challenge also shows the importance of small speedups (that are oftentimes reported in literature), for these algorithms; even a 10% speedup may even result in considerable speed gains, as these algorithms may run for months³.

The last years have witnessed many efforts towards this goal, and compiling and analysing those algorithms and implementation is part of the goal of this chapter. Non-scalable and, in particular, non-parallel algorithms are usually not considered in this process, as they cannot be deployed on high core count machines, and therefore have limited harm potential. Some of these implementations are used in the next chapters of this thesis, either for comparison against the newly developed implementations or as a building block. As said before, this thesis concentrates on three of those problems: lattice basis reduction, the SVP and the CVP.

3.2 Algorithms

3.2.1 Lattice basis reduction

Lattice basis reduction forms a cornerstone of lattice theory. Lattice basis reduction algorithms were used to break cryptosystems even before lattice-based cryptosystems existed e.g. [94], and can be used to attack lattice-based cryptosystems today. For example, the Ajtai-Dwork cryptosystem can be attacked with a combination of a lattice construction and lattice reduction algorithms [89]. There are two essential relations between the performance of lattice basis reduction algorithms and the vulnerability of lattice-based cryptosystems. First, lattice reduction may be sufficient to break a given system directly. Lattice reduction outputs a basis whose shortest vector is an approximation of the shortest vector of the lattice (and in some special cases the shortest vector itself); depending on the approximation needed to break the system, this may suffice. Second, lattice reduction is usually a building block of other, more complex attacks, which comprise lattice basis reduction algorithms and algorithms for other problems, e.g. the SVP. Therefore, studying lattice basis reduction algorithms and their practical performance is of major importance.

²www.latticechallenge.org/svp-challenge/

³<http://www.latticechallenge.org/svp-challenge/halloffame.php>

As mentioned before, LLL was the first algorithm for lattice basis reduction. The original algorithm was described with rational arithmetic. Given that it was too expensive, many LLL floating-point variants, i.e., versions that replace rational arithmetic with floating point arithmetic, were proposed. It should be noted that these versions may introduce errors, which are handled to guarantee completion or completion with high probability.

Today, there are a few variants of LLL with floating point arithmetic. In this thesis, we use Schnorr and Euchner’s LLL floating point variant, also referred to as LLL-SE [114], although Nguyen and Stehlé’s variant, referred to as L^2 , is also important [90]. Both variants are described in detail by their authors in a recent survey on LLL and its applications, published on the occasion of LLL’s 25th birthday (cf. [91, 113, 121]). Another LLL variant worth mentioning is LLL with deep insertions, also proposed by Schnorr and Euchner [114]. LLL with deep insertions, often referred to as DEEP, yields a basis of better quality. The difference to the original LLL is the swapping of vectors: instead of swapping the vectors \mathbf{b}_i and \mathbf{b}_k when the Lovász condition does not hold anymore, the \mathbf{b}_k is inserted between the first and the $(k - 1)$ th position (cf. the original paper for more details on DEEP [114]). The literature suggests that DEEP improves the quality of the basis substantially, while running reasonably well - although not as fast as LLL-SE - in practice cf. Section 4.4. in [127] and pages 20-21 in [114]. This dissertation does not show results specifically for DEEP, as the results we present are applicable to any variant of LLL.

The Block Korkine-Zolotarev (BKZ) algorithm is a generalization of LLL, which was proposed by Schnorr in 1987 [110], and it is currently the most practical lattice basis reduction algorithm. Here, Schnorr proposed a hierarchy of algorithms that generalize LLL and offer a trade-off between the running time and the quality of the yielded basis. The algorithm picks a window of at most β vectors at a time, and finds the shortest vector in the lattice that is the result of the projection of those vectors orthogonally to the span of the previous vectors in the basis (Section 5.3 in Chapter 5 explains the inner works of BKZ in full detail). Then, the shortest vector of the projected basis is inserted into the original basis and LLL is called over the entire basis, thus removing linear dependences that may arise after inserting a vector in the basis. The block of β vectors is then shifted one position to the right until all vectors are covered (the last blocks have less than β vectors). The algorithm stops when one complete round of sliding windows does not result in new vectors, as explained in full detail in Chapter 5.

Proposed by Chen and Nguyen, the BKZ 2.0 algorithm is an improved version of BKZ [21]. A major difference to BKZ is that BKZ 2.0 uses ENUM with the extreme pruning technique [38], which performs the lattice enumeration with a success probability p , for $\lfloor 1/p \rfloor$ different bases $G_1, \dots, G_{\lfloor 1/p \rfloor}$, obtained by randomizing the original basis B . We do not show results pertaining to BKZ 2.0 in this dissertation, primarily because we concentrate on the original BKZ (see Section 5.3) and a parallel BKZ 2.0 can be devised with the same mechanisms we present for BKZ. The complexity of the lattice basis reduction algorithms is summarized in Table 3.1.

3.2.2 The shortest vector problem

The Shortest Vector Problem (SVP) is one of the most studied problems in lattice-based cryptanalysis. In this context, the SVP crops up at two distinct points. First, very short vectors are needed to break some cryptosystems (i.e. one has to solve the α -SVP, an approximate version of the SVP, to break the system), and understanding the hardness of the SVP is paramount to understand the hardness of the α -SVP (also because there are no specific solvers for the α -SVP, instead one uses SVP-solvers and stop them at some

Algorithm	Time Complexity	Type
LLL [65]	$O(d^{4+\epsilon} n \log^{2+\epsilon} B)$	Exact
Schnorr-Euchner (LLL-SE) [114]	$O(d^3 n (d + \log B)^{1+\epsilon} \log B)$	Heuristic
Deep Insertions (DEEP) [114]	Unknown	Heuristic
Nguyễn and Stehlé (L^2) [90]	$O(d^{3+\epsilon} n (d + \log B) \log B)$	Heuristic
BKZ [110]	Unknown	Heuristic
BKZ 2.0 [21]	Unknown	Heuristic

Table 3.1: Lattice basis reduction algorithms (with a fast multiplication algorithm).

point). Second, BKZ, the most practical lattice basis reduction algorithm that can also be used to solve the α -SVP, uses SVP-solvers as part of its logic.

There are many algorithms to solve the SVP. The first and also the most studied class of SVP-solvers is the family of enumeration algorithms. Enumeration algorithms are among the most practical SVP-solvers on random lattices. Enumeration has been studied since the early eighties and several improvements have been proposed over the years. A remarkable example is extreme pruning [38], which reduces the amount of work in the algorithm substantially, at the same time the probability of success is reduced by a much smaller factor. In particular, a combination of enumeration-based algorithms with extreme pruning and efficient lattice reduction algorithms seems to be an efficient approach to solve the SVP in high dimensions⁴.

The progress in sieving algorithms took off only in 2008, when they were first-hand shown to be tractable for moderate dimensions [92], even though still uncompetitive with enumeration routines. In 2010, Micciancio et al. proposed ListSieve and GaussSieve, the first sieving heuristic that outperformed enumeration routines [83], although for a brief period of time. In the past few years, sieving algorithms have been attracting increasing attention, due to promising results simultaneously on the theoretical and practical sides e.g [76, 59, 73, 124, 10]. In particular, the current most practical sieving algorithms, HashSieve and LDSieve, are competitive against enumeration if enough resources are available [73, 70].

Sieving algorithms have specific properties that make them especially interesting. First, they are asymptotically better than enumeration ($2^{\mathcal{O}(n)}$ vs. $2^{\mathcal{O}(n \log n)}$), which means that they have to be faster than enumeration for a sufficiently large dimension n . Second, they can take advantage of specific lattices, such as ideal lattices, whereas enumeration algorithms cannot (see [108, Section 6.1]). Third, as sieving algorithms are iterative, reductions may be failed occasionally and convergence is still achieved. This last feature is especially interesting for parallel computing, as it happens often that the parallelization of vector reductions may lead to some missed reductions. Lastly, advances on sieving algorithms have been published during the last years and it is still argued that considerable room for improvement exists.

Although enumeration and sieving are the most studied families of SVP-solvers, other important algorithms exist. In 2002, Agrell et al. proposed an algorithm for the SVP, called *Relevant vectors*, which is based on the Voronoi cell of a lattice [1, *Relevant vectors*, Section VI C], similarly to the algorithms by Micciancio et al. [82]. Even though the algorithms based on the Voronoi cell of a lattice offer the best theoretical time-complexity bounds, they are intractable in practice [108, 128], [25, Section 4.1.4]. Another important class of SVP-solvers is that of Random Sampling (RS) algorithms. RS algorithms were inactive for some years, given that the most important contribution dates back to 2003, when Schnorr proposed a RS method called Random Sampling Reduction (RSR) [112], which was later on improved by

⁴www.latticechallenge.org/svp-challenge/

Buchmann et al. [18]. Recently, RS algorithms were revitalized, as Fukase et al. proposed a new very, efficient algorithm based on RS [36].

Algorithm	Family	Time Complexity	Space Complexity	Type	Year
Relevant Vectors [1]	Voronoi Cell	2^{2n}	2^n	CVP/SVP	2002
Micciancio et al. [82]	Voronoi Cell	2^{2n}	2^n	SVP	2010
AKS [5]	Sieving	$2^{3.40n+o(n)}$	$2^{1.99n+o(n)}$	SVP	2001
Nguyễn-Vidick [92]	Sieving	$2^{0.42n+o(n)}$	$2^{0.21n+o(n)}$	SVP	2008
ListSieve (LS) [83]	Sieving	$2^{3.20n+o(n)}$	$2^{1.33n+o(n)}$	SVP	2010
GaussSieve (GS) [83]	Sieving	Unknown, see text		SVP	2010
LS-birthday [101]	Sieving	$2^{2.47n+o(n)}$	$2^{1.24n+o(n)}$	SVP	2009
WLTB sieve [129]	Sieving	$2^{0.39n+o(n)}$	$2^{0.26n+o(n)}$	SVP	2011
Three-level sieve [133]	Sieving	$2^{0.38n+o(n)}$	$2^{0.28n+o(n)}$	SVP	2013
Overlattice sieve [11]	Sieving	Trade-off, see text		CVP/SVP	2014
HashSieve [59]	Sieving	Trade-off, see text		SVP	2015
LDSieve [10]	Sieving	Trade-off, see text		SVP	2016
Kannan [44] ⁵	Enumeration	$2^{O(n \log n)}$	Irrelevant	CVP/SVP	1983
ENUM [114]	Enumeration	$2^{O(n^2)}$	Irrelevant	SVP	1994
SE [1]	Enumeration	$2^{O(n^2)}$	Irrelevant	SVP/CVP	2002
Extreme Pruning [38]	Enumeration	Unknown	Irrelevant	SVP	2010
MW-Enum [84]	Enumeration	$2^{o(n \log n)}$	Irrelevant	SVP	2015
Fukase et al. [36]	RS	Unknown		SVP	2015

Table 3.2: Algorithms for the SVP and the CVP, and their time and space asymptotic complexities. n is the lattice dimension. ⁴Improved version of original algorithm.

Table 3.2 shows the most relevant algorithms for the SVP and CVP, as well as their time and space complexities. As one cannot bound the number of samples required by GaussSieve, its time complexity is not known. However, it was empirically determined that GaussSieve’s execution time grows according to $2^{0.468n+o(n)}$ in [83] and $2^{0.568n+o(n)}$ in [75]. GaussSieve’s space complexity seems to grow according to $2^{0.18n}$, and is *safely* bounded by $2^{0.21n}$, as it can be reasonably conjectured that this also bounds the kissing constant [83]. The Overlattice sieve algorithm’s time complexity may vary between $2^{0.3774n}$ and $2^{0.415n}$, while its space complexity varies between $2^{0.2075n}$ and $2^{0.2925n}$, depending on the parameters α and β (space complexity increases when time complexity decreases and vice versa) [11]. HashSieve’s complexities also depend on parameters that can be traded-off. In particular, HashSieve attains both a time and space complexity of $2^{0.3366n+o(n)}$ (space complexity comes down to $2^{0.2075n+o(n)}$ when applying HashSieve’s core to Nguyễn-Vidick’s sieve), when optimizing these parameters for time. LDSieve’s complexities also depend on α and β , which can be parametrized. With $\alpha = \beta = \frac{1}{2}$, i.e. the best parameters for time optimization, LDSieve’s time complexity becomes $2^{0.292n+o(n)}$, which is currently the best known complexity for solving the SVP heuristically [10]. Space complexity matches this time complexity when $\alpha = (\frac{1}{4}, \frac{1}{2})$ and $\beta = \frac{1}{2}$.

3.3 Implementations

There is a considerable number of implementations of most of the algorithms described in Section 3.2. This section enumerates the most important of these implementations, some of which are used throughout

the thesis for comparison purposes (for both correctness and performance). The implementations are divided between lattice basis reduction algorithms and CVP/SVP-solvers.

3.3.1 Lattice basis reduction algorithms

There are a few libraries implementing these algorithms and their variants, some of which are described in the following.

NTL. The Number Theory Library (NTL) is a widely used library written by Victor Shoup, which implements a large set of algorithms for number theory⁶. Some lattice algorithms, such as LLL and BKZ, are included. Enumeration algorithms are not part of the API, as they are only implemented inside the BKZ function. NTL is mainly used as a baseline implementation for the implementations of LLL and BKZ proposed in this thesis. The implementations that the library provides are not parallel.

fpLLL. Having had contributions by many different authors, the fpLLL library implements a floating point version of LLL, BKZ and an enumeration routine for the SVP. The library does not provide a parallel implementation of LLL, but it has been gaining considerable reputation among the community for its efficiency. The last version of fpLLL is available from a GitHub repository⁷.

pLLL. The pLLL Lattice Reduction Library was developed by the Applied Algebra group at the University of Zurich⁸ and launched in July 2014. The pLLL library is possibly the most complete among lattice libraries, and certainly the most complete regarding lattice basis reduction algorithms. It provides both multiple variants of LLL and BKZ, including different auxiliary methods such as Gram-Schmidt orthogonalization. The library is thread-safe and includes a shared-memory parallel implementation of enumeration.

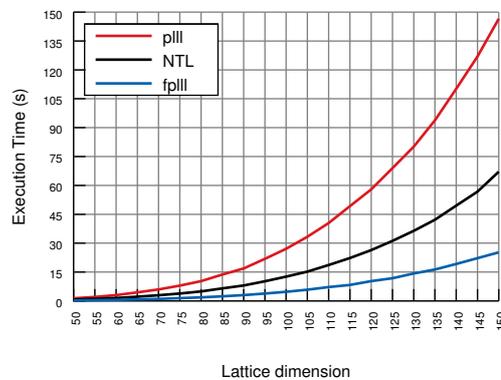


Figure 3.1: Execution time of the LLL implementations in pLLL, NTL and fpLLL, for lattices between dimension 50 and 150, for one thread, on Lara.

There are other libraries or packages that implement lattice basis reduction algorithms. For instance, there is a Haskell package that implements a basic LLL function⁹. However, the three libraries mentioned above are the main references for lattice reduction algorithms, and, to the best of our knowledge, the most efficient ones. The performance of these libraries is primarily assessed in this chapter. Figure 3.1 shows the performance of pLLL, NTL and fpLLL for LLL, on Lara (cf. Section 2.3). The codes were compiled with GNU g++ 4.8.4. Figures 3.2 and 3.3 show the performance of pLLL, NTL and fpLLL for BKZ with $\beta = 20$ and

⁶<http://www.shoup.net/ntl/>

⁷<https://github.com/dstehle/fplll>

⁸<https://felix.fontein.de/plll/>

⁹<https://hackage.haskell.org/package/Lattices>

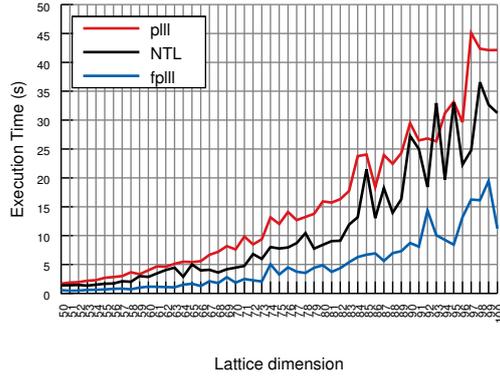


Figure 3.2: Execution time of BKZ implementations for plll, NTL and fplll, with $\beta = 20$, for lattices between dimension 50 and 100, on Lara.

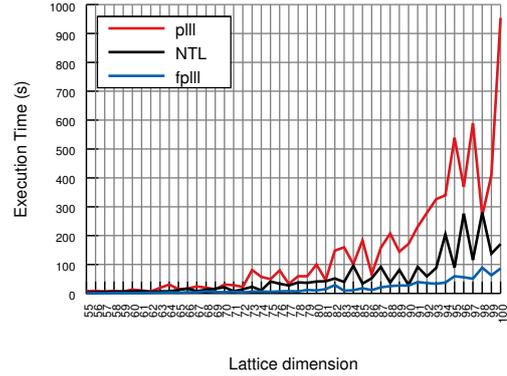


Figure 3.3: Execution time of BKZ implementations for plll, NTL and fplll, with $\beta = 25$, for lattices between dimension 50 and 100, on Lara.

$\beta = 25$, for one thread, respectively. fplll is by far the fastest library, whereas NTL is generally faster than plll. It is important to note that the outputs of the libraries are different (plll and NTL yield the same basis for LLL, in general, whereas the three libraries yield all different bases for BKZ), as they have different implementations. We carried out a comparison of the quality of the yielded bases in Section 5.2.2.

Another aspect worth mentioning is *multiple precision*. Some lattices, such as those in the SVP-challenge, involve vectors with very large coordinates, which may have hundreds of digits and may not be represented in native data types such as integers or even longs. 64 bit words are not enough to store these numbers and so multiple precision software layers are necessary. Multiple precision libraries implement formats to represent values with very large numbers of digits, by breaking them down into smaller chunks, of up to 64 bit words. At the same time, the algorithms that perform the basic operations such as sums or multiplications, account for these formats. GMP¹⁰ and MPFR¹¹ are among the most popular multiple precision libraries. NTL implements a custom multiple precision layer and offers compatibility to GMP. fpLLL and plll work both with GMP and MPFR.

3.3.2 CVP- and SVP-solvers

Table 3.3 compiles most of the available implementations of CVP- and SVP-solvers. The table includes ENUM, which is the *de facto* enumeration algorithm used in practice. Kannan’s algorithm is not included as it is only important from a theoretical standpoint (ENUM is considerably more practical). LatEnum is a library associated with fpLLL, which includes parallel SVP and CVP solvers¹². It was written by Xavier Pujol, one of the developers of fpLLL. LatEnum was also the only library to include parallel versions of lattice algorithms, implemented with MPI, until plll was launched (which includes a parallel implementation for shared-memory).

It is infeasible to conduct a comprehensive, fair comparison between all the SVP-solvers mentioned in Table 3.3, as there are many variables influencing the comparison, ultimately rendering it unfair. For instance, some of these algorithms require some parameters whose optimality is very difficult (if possible at all) to determine upfront. Also, sieving algorithms may require impractical amounts of memory to

¹⁰<https://gmplib.org/>

¹¹www.mpfr.org/

¹²<http://xpujol.net/>

Algorithm	Sequential	Shared-memory	Distributed-memory	GPUs
ENUM	(LatEnum,✓),([29],✓)		(LatEnum,✓)	N/A
ENUM-EP		N/A	([58],✓)	
ENUM-SE++		([26],✓)	N/A	
MW-Enum	([84],✓)		N/A	
RS-based		([36],✗)	N/A	([109],✓)
Voronoi Cell	(pIII,✓)	N/A	N/A	N/A
ListSieve	(pIII,✓),([75],✓)	([75],✓)	N/A	N/A
LS-Birthday	(pIII,✓)	([75],✓)	N/A	N/A
GaussSieve	(pIII,✓),([83] ¹³ ,✓)	([86],✓),([76],✓)	([15],✓)	N/A
Overlattice		([11],✗)	N/A	N/A
HashSieve		([73],✓)	N/A	N/A
LDSieve		([74],✓)	N/A	N/A

Table 3.3: Sequential and parallel implementations of different SVP-solvers, in the form of (implementation, code availability). N/A stands for not available. ¹³This led to the creation of the *gsieve* library: <https://cseweb.ucsd.edu/pvoulgar/impl.html>

operate, depending on the lattice dimension. There are, however, some take-home lessons that have been reported consistently in the literature. First, Voronoi Cell algorithms are still impractical, despite their good theoretical time complexities e.g. [108, 128], [25, Section 4.1.4]. Second, sieving algorithms have better complexity than enumeration algorithms, but until this thesis work started, they were not as practical and, especially, scalable. This is covered in detail in Chapter 6. Third, enumeration with (extreme) pruning is one of the most efficient algorithms to solve the SVP in practice [60], but recent sieving algorithms seem to be competitive [59, 10, 73, 74]. Throughout this thesis, we conduct one-to-one comparisons, which help to define the practicability of this set of algorithms.

3.4 Contributions of this thesis to the state of the art

The previous sections of this chapter showed the landscape of lattice based-cryptanalysis. This is a very active field. For example, in 2014, new sieving algorithms were introduced, including the Overlattice sieve, HashSieve and LDSieve. While this thesis can naturally not cover the full spectrum of algorithms that were proposed, sufficiently many were implemented to demonstrate the relevance of the techniques developed here, and some were even implemented as they were published, including HashSieve and LDSieve. This thesis contributes primarily to the understanding of the practicability and scalability of lattice algorithms for lattice basis reduction and, to a larger extent, the SVP. This is especially relevant because many results obtained in practice are not congruent with those expected from theory. Additionally, understanding the performance of these algorithms in practice is paramount as it guides parameter selection for schemes.

In Chapter 4, we show that parallel enumeration efficiency depends heavily upon the right load balancing of the enumeration tree, which can be achieved with the right use of load balancing (cf. also [26]). We propose two different scalable implementations of ENUM, both implemented with OpenMP and an ad hoc demand-driven mechanism, implemented with POSIX threads. Both versions scale linearly in most cases, and super-linearly in specific instances. The implementation based on the demand-driven mechanism is the fastest published parallel full enumeration SVP-solver to date, as it surpasses the state of the art implementation by Dagdelen et al. [29].

Chapter 5 centers on lattice basis reduction algorithms. First, we show that LLL lends itself to vectorization, if the right data structures are used (cf. also [71]). We propose a vectorized version of LLL which surpasses NTL, attaining relevant gains for high dimensional lattices. Then, we analyze the parallelization of BKZ, using the parallel Enumeration implementations presented in Chapter 4 as building blocks. We show that for high block sizes, good scalability is achieved.

Chapter 6 represents the bulk of the contributions of this dissertation. The first result of the chapter is the parallelization of the ListSieve algorithm (cf. also [75]), answering positively the question posed in [86], of whether a parallel ListSieve could surpass GaussSieve (as in 2011 the parallel version of GaussSieve did not scale well). Next, we show that it is possible to parallelize GaussSieve, attaining almost linear scalability, by slightly relaxing the properties of the algorithm and allowing missed reductions (cf. also [76]). From this point on, sieving algorithms became regarded as scalable algorithms, which increased the efforts towards refining them. We also present the idea that some reductions in sieving algorithms (and in particular in GaussSieve) can be avoided by taking into account the coordinates of the vectors (cf. also [35]). This is in fact the fundamental idea behind HashSieve, a very efficient sieving algorithm proposed in 2015 [59]. In this chapter, we also propose parallel versions of HashSieve (cf. also [73]), using probable lock-free data structures, and LDSieve (cf. also [74]). These implementations motivated further investigation around the memory usage of sieving algorithms, and how it can be enhanced, which we present at the end of the chapter (cf. also [70]). The parallel HashSieve implementation in [70] currently holds the record on the highest dimension broken by sieving algorithms on random lattices, as we solve the SVP on a random lattice in dimension 107, for published algorithms.

Parallel Vector Enumeration

Synopsis. This chapter introduces SE++ and ENUM, enumeration-based CVP- and SVP-solvers, and efficient parallel implementations of these algorithms on multi-core CPUs. In particular, we present two different models to parallelize the algorithms, based on work-sharing and demand-driven mechanisms, which we implemented with OpenMP (applied to SE++ and ENUM) and POSIX threads (applied to ENUM), respectively. We show that, although both scale well, the demand-driven implementation attains better results, especially for higher thread counts, due to factors such as efficient task generation, task handling, and memory usage. For instance, the demand-driven implementation creates tasks only when the previous tasks are executed, thus enabling each task to start at the right location in the tree (as opposed to the OpenMP-based implementation, where tasks start at the root, thus repeating computation). The demand-driven implementation presented in this chapter is the fastest known parallel enumeration-based SVP-solver to date, attaining super-linear speedups in some instances.

"Life is really simple, but men insist on making it complicated.", Confucius.

4.1 Enumeration algorithms

Many of the ideas presented in this Section have been published in [26].

Breakthrough papers on the SVP and the CVP date back to 1981, when Pohst presented an approach that examines lattice vectors inside a hypersphere [99], and to 1983, when Kannan showed a different approach using a parallelepiped [52]. These two different types of enumeration-based solvers lay the foundation for virtually every enumeration-based method that followed. For instance, many extensions were proposed in the following years, including one by Fincke and Pohst, in 1985 [34], and by Kannan (following Helfrich's work [46]), in 1987 [52]. In 1994, Schnorr and Euchner proposed a significant improvement of Pohst's method [114], that was later found to be substantially faster than Pohst's and Kannan's approaches [1].

Currently, the most practical full enumeration algorithms are ENUM [114] and SE++ [40], which differ mainly in the pre-processing. Some sort of pruning (either linear [111] or extreme [38]) can be applied to both algorithms. Roughly speaking, pruning is a technique that discards computation by pruning

the enumeration tree, lowering the success probability of the algorithm, but the amount of computation is typically decreased by a much larger factor than the success probability.

SE++ is a CVP-solver, which can be modified to solve the SVP, proposed by Ghasemmehdi and Agrell in 2011 [40]. The algorithm is an improved version of the SE algorithm described by Agrell et al. in [1], which is in turn based on ENUM, proposed by Schnorr and Euchner [114]. The SE++ algorithm consists of 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 \mathbf{B} , is reduced (e.g., with either the BKZ or LLL algorithms). The resulting matrix \mathbf{D} is transformed into a lower-triangular matrix, which is usually referred to as \mathbf{G} . This process can either be accomplished with a QR decomposition or the Cholesky decomposition. This transformation can be seen as a change of the coordinate system. The decomposition of \mathbf{D} also generates an orthonormal matrix \mathbf{Q} . The target vector \mathbf{r} , i.e., the reference vector for the closest vector (if the SVP is being solved, \mathbf{r} is the origin of the lattice), is also transformed into the coordinate system of \mathbf{G} , i.e., $\mathbf{r}' = \mathbf{r}\mathbf{Q}^T$. Finally, the sphere decoding is initialized with the dimension of the lattice n , the transformed target vector \mathbf{r}' and the inverse of \mathbf{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. On one hand, it is the process of enumerating lattice points inside a hypersphere (cf. [40] for a detailed mathematical description). On the other hand, it can be described as a tree traversal, which also makes it easier to explain our parallelization approaches. This traversal is a depth-first traversal on a weighted tree, which is formed by all vectors of projections of \mathcal{L} orthogonal to the basis vectors. One refers to the process of visiting a child node (decrementing i , where i denotes the depth of the node that is being analyzed at any given moment) as *moving down* and the process of visiting a parent node (incrementing i) as *moving up*. In the following, we describe how the tree traversal is performed and how it is impacted by the variables used throughout.

The algorithm starts at the root of the tree and stops when it reaches the root again. Each node at depth $(i - 1)$ that is being visited is determined by \mathbf{u}_i , which is an array that contains the position of each node in the tree, having the hypersphere in dimension $(i - 1)$ as the reference. \mathbf{u}_i is updated whenever a new tree node is visited, and depends on Δ_i , which is itself updated dynamically. Δ_i contains the step that has to be taken, when at a given node, to visit its next sibling. Note that Δ_i only contains one value at each given instant, although there is a Δ_i for each depth level. Based on the Schnorr-Euchner refinement [114], the siblings of each node are visited in a zigzag pattern, and so Δ_i contains a sequence of different and symmetric values, e.g. 0, 1, -1, 2, -2, etc. In Figure 4.1, which we comment later on, for simplicity, the zigzag pattern was ignored and Δ_i was given the values 0, 1, 2, etc. The squared distance from the target vector \mathbf{r} (note that if the SVP is being solved, then \mathbf{r} is the zero-vector) to the node that is being analyzed is denoted by λ_i (and so it changes depending on the current node), while C is the squared distance of \mathbf{r} to the closest vector to \mathbf{r} found so far. C is initialized to infinity. If $\lambda_i < C$, the algorithm will *move down*, otherwise it will *move up* again. Whenever a leaf is reached, the values of the vector \mathbf{u} are saved in $\hat{\mathbf{u}}$, which represents the closest vector to \mathbf{r} found so far, and C is updated, which reduces the number of nodes that still have to be visited. Although the algorithm behaves as a tree traversal, there is no physical tree (i.e. a data structure) implemented.

Although in this dissertation we only work with SE++, we note again that this is an improved version of another algorithm, called SE [1]. There is one fundamental difference between these algorithms;

as proposed by Ghasemmehdi and Agrell [40], a vector \mathbf{d} is used to store the starting points of the computation of the projections. The value $\mathbf{d}_i = k$ determines that, in order to compute $\mathbf{E}_{i,i}$ (see [40] for further details about matrix \mathbf{E} , which contains the projections of the orthogonally displaced target vector \mathbf{r} to the basis vectors), we should start the projection from the k th layer, where $k > i$, and only calculate the values of $\mathbf{E}_{j,i}$ for $j = k - 1, k - 2, \dots, i$, thereby avoiding redundant calculations.

The ENUM algorithm lays the foundation for SE++, and therefore the algorithms share key points. For instance, both algorithms enumerate the lattice points inside a hypersphere, in order to find the shortest vectors. The hypersphere is also examined, layer by layer, in a zigzag pattern, which also maps well on a tree traversal. However, there are three key differences between SE++ and ENUM. First, ENUM resorts to the Gram-Schmidt orthogonalization in the pre-processing phase (as opposed to a QR or Cholesky decomposition in SE++). Second, ENUM starts at a leaf (instead of the root of the tree, as in SE++) of the tree; this is because the first leaf to be found is in fact the first vector of the basis, thus saving the computation incurred from the root to the first leaf. The third and last difference between ENUM and SE++ is that ENUM discards symmetric branches of the tree, which SE++ does not. However, this can only be done for the SVP (as we showed in [26] and we show in the next section), and not for the CVP.

In this chapter, we propose parallel implementations of SE++ and ENUM, for shared-memory CPUs, based on two different strategies. The first strategy, presented in Section 4.2.1, is based on work-sharing, and it is implemented with OpenMP tasks. This model splits the tree among different OpenMP tasks, in such a way the workload of the different threads is as balanced as possible. This strategy was applied both to SE++ and ENUM. The second strategy is based on demand-driven workload distribution and is presented in Section 4.2.2. We applied this strategy to ENUM (and compared it to ENUM with the OpenMP-based strategy). We implemented it with POSIX threads, and it replicates the OpenMP tasking mechanism with low-level control, which enables one to generate tasks more efficiently, avoid duplicated computation, and improve memory usage. As a result, the demand-driven implementation attains better scalability than the OpenMP-based implementation, especially for high thread counts. The demand-driven implementation presented in this chapter is the fastest known parallel enumeration-based SVP-solver, attaining super-linear speedups on some instances.

4.2 Parallel ENUM and SE++ on shared-memory systems

As mentioned before, from a computational standpoint, enumeration algorithms consist in a depth-first tree traversal. Thus, the parallelization of SE/SE++ and ENUM can be trivially accomplished if the different branches of the enumeration tree are computed in parallel (synchronization is only required to update the best vector at each instant). The biggest challenge with such a model is to balance the computation that pertains to different branches of the tree, as the tree is highly imbalanced. Although the tree is imbalanced, there is a pattern concerning the computational load: the branches become computationally lighter from the left to the right. Taking advantage of this pattern, we devised different parallel implementations of the SE++ and the ENUM algorithms, based on a work-sharing mechanism, implemented with the OpenMP tasking system and published in [26], and one unpublished ad hoc demand-driven mechanism, implemented with POSIX threads.

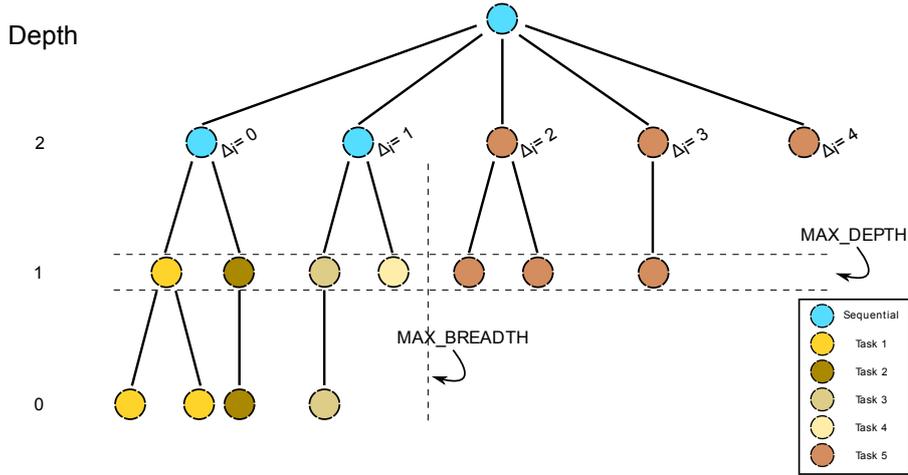


Figure 4.1: Map of the algorithm workflow on a tree, partitioned into tasks, according to the parameters $\text{MAX_BREADTH} = 2$ and $\text{MAX_DEPTH} = 1$, $n = 3$ and running with 4 threads. For simplicity, the values of Δ_i are merely representative, as e.g. they ignore the zigzag pattern.

4.2.1 Tasking mechanism on SE++

As previously mentioned, the workflow of the algorithm can be naturally mapped onto a tree traversal, where different branches can be computed in parallel. Figure 4.1 shows a partition of these branches into several tasks that can be computed in parallel, by different threads. In [26], we proposed a model to parallelize SE++, based on two parameters MAX_BREADTH and MAX_DEPTH , which define the number and size of tasks so that the computation is balanced among threads and high scalability is achieved. The implementation associated to the model was written in C, and creates tasks with OpenMP. The implementation can both solve the CVP and the SVP (a few lines of code are added for the latter). Once tasks are created, they are automatically added to a queue of tasks, and scheduled by the OpenMP run-time system among the running threads. This system also defines the order of execution of the created tasks, in run-time. (Very fine-grained) synchronization is only used to update the best vector found at any given instant (the closest to the target vector, at a given moment), as explained below.

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 (and their descendants) in that level is created, as also shown in Figure 4.1. However, when creating a task for a given node whose $|\Delta_i| = \text{MAX_BREADTH}$, it entails not only that node but also all its siblings (unless they already belong to other tasks) and their descendants, as exemplified by Task 5, in Figure 4.1. Note that, although in Figure 4.1 there is only one task verifying this, this can happen for any level of the tree. Once tasks are created, they are (either promptly or after some time) assigned to one of the threads within the team, by the OpenMP run-time system. There is an implicit barrier at the end of the single region, which means that all the created tasks will be, at that point, already processed.

The `MAX_BREADTH` and `MAX_DEPTH` parameters were created so that all threads execute a similar amount of work, given that the tree is considerably imbalanced. We created the `MAX_BREADTH` parameter based on the fact that the rightmost subtrees in the tree contain fewer descendants and are, therefore, lighter. We identify the workload associated to each subtree with the value of $|\Delta_i|$, as it always holds that a bigger $|\Delta_i|$ translates into a lighter subtree (due to the zigzag pattern, Δ_i can be negative, and thus the use of its absolute value). Note, however, that the value of $|\Delta_i|$ can only be used to compare the weight of different nodes of the same subtree and at the same depth. Therefore, we can assume that, after a given $|\Delta_i|$ (which we compare against `MAX_BREADTH`), all the subtrees should be grouped together. If `MAX_BREADTH` is high, then more (and finer-grain) tasks are created. The optimal values of both parameters have to be chosen empirically (later we show experiments with different values for `MAX_BREADTH`). The maximum depth is chosen based on the number of threads in the system, so that the enumeration tree is more split, i.e. the number of tasks is larger, for higher thread counts. Thus, we define $\text{MAX_DEPTH} = n - \log_2(\#Threads)$, which determines the lowest depth that is reached to split subtrees. Similarly to `MAX_BREADTH`, the value for this parameter was also chosen based on empirical tests.

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. This is because in order to compute a given node (or sub-tree), one must know information from the parent nodes in the tree, such as the previous positions in \mathbf{E} and Δ . Storing all this information for all tasks multiplies memory usage and becomes infeasible. The level of the subtree that was assigned and the nodes that have to be recomputed, given by the vector $\mathbf{u_Aux}$, are passed as arguments to each task. Additionally, the value of $|\Delta_i|$ is also passed to the task, in order to differentiate subtrees that were grouped together from single subtrees. In particular, $|\Delta_i|$ is compared to `MAX_BREADTH`. If $|\Delta_i|$ is smaller than `MAX_BREADTH`, then only the root node of the task and its descendants are to be computed; if not, then both the root, its siblings (whose $|\Delta_i| \geq \text{MAX_BREADTH}$) and their descendants are computed.

As said before, in this version we recompute the parent nodes in the tree, otherwise memory usage would rise to impractical levels; to avoid this, one would have to store the data for all the tasks upfront. Therefore, instead of allocating each vector and matrix for each task, it is only necessary to allocate a much smaller vector $\mathbf{u_Aux}$, per task, which contains the coefficients of the nodes that have to be recomputed. To do so, each thread concurrently allocates its own (private) block of memory (a struct) for matrix \mathbf{E} and vectors \mathbf{u} , \mathbf{y} , λ , Δ and \mathbf{d} (for details about these structures see [40]) and re-uses the same memory for the execution of all the tasks that are assigned to it. Empirical tests showed that, by allocating these data structures per thread (instead of per task), performance can be improved by a factor of as much as 20%.

The value of C is stored in a global variable, accessible by every thread. Threads check the value of C , which dictates the rest of the nodes that are visited by each thread. C is initialized with $1/\mathbf{H}_{1,1}$, instead of infinity, to prevent the creation of unnecessary tasks. For the same reason, $\hat{\mathbf{u}}$ is initialized with $\hat{\mathbf{u}} = (1, 0, \dots, 0)$. Although these variables are shared among all the threads, only one thread updates them at a time. An OpenMP critical section is used to manage this synchronization. Every time a thread executes the critical section, it checks $\lambda_0 < C$ again, since other threads might update those values in the meantime.

Improved SE++

The SE++ algorithm computes the whole enumeration tree, thereby computing several vectors that are symmetric of one another. When solving the SVP, the purpose of the algorithm is to find the shortest vector \mathbf{v} of norm $\lambda_1(\mathcal{L})$, and so it is not relevant whether \mathbf{v} or $-\mathbf{v}$ is found, since they have the exact same norm. Therefore, the computation of one of these two vectors can be avoided, thus reducing the number of vectors that are ultimately computed. In fact, given that ENUM only solves the SVP, it already discards symmetric branches in the tree. In [26], we showed how to incorporate this optimization in SE++, which yielded an implementation we refer to as *Improved SE++*, from here forward.

The idea is, similarly to ENUM, to use a variable called *last_nonzero*, which stores the largest depth i of the vector \mathbf{u} for which $u_i \neq 0$. Using \mathbf{u}_i , one can determine which subtree to visit next. To avoid symmetric branches, \mathbf{u}_i is updated differently, depending on whether the nodes contains symmetric subtrees; on trees that contain symmetric subtrees, the value of \mathbf{u}_i is incremented, searching only in one direction. It should be noted that there are only subtrees whose computation can be avoided on the leftmost nodes of each level. Each time the algorithm moves up on the tree and $i \geq \text{last_nonzero}$, *last_nonzero* 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. For more details on this optimization, we refer the reader to [26]. In the next subsection, we show that this improved version of SE++ is about 50% faster than SE++, and faster than the parallel ENUM implementation proposed by Dagdelen et al. in [29], for 1-32 threads.

Results

The codes were written in C and compiled with the GNU g++ 4.6.1 compiler, with the -O2 optimization flag (-O3 was slightly slower than -O2). We used Peacock for this set of benchmarks (cf. Table 2.1). Additionally, we used NTL¹ for LLL and BKZ basis reduction, and Eigen² for the QR decomposition, inverse and transpose matrix computations. Although the code was written in C, we used g++ to compile it, as these libraries are written in C++. We used Goldstein-Mayer bases for random lattices, available from the SVP-Challenge³, all of which were generated with seed 0. Although the execution times of the programs were fairly stable, each program was executed three times and the best sample was selected. The basis pre-processing was not included in the time measurements. We tested our implementation to solve the CVP and SVP (SE++ can be slightly modified to solve the SVP [26]).

As mentioned before, `MAX_BREADTH` and `MAX_DEPTH` must be defined empirically, and so we performed some tests in order to find the optimal value of these parameters. In particular, for `MAX_BREADTH`, we tested different lattice dimensions and thread counts. For simplicity, Figures 4.2 and 4.3 show only partial results. The results were very similar when executing the CVP and the SVP, even though the figures pertain only to the CVP runs. Figure 4.2 shows the execution time for different values of `MAX_BREADTH` for BKZ-reduced lattices (with block-size 20) when running with 16 threads (for other thread counts the results were very similar), when solving the CVP. Figure 4.3 shows the number of tasks that are created in our parallel implementation, for 4, 8 and 16 threads, when solving the CVP.

¹<http://www.shoup.net/ntl/>

²<http://eigen.tuxfamily.org/>

³<http://www.latticechallenge.org/svp-challenge/>

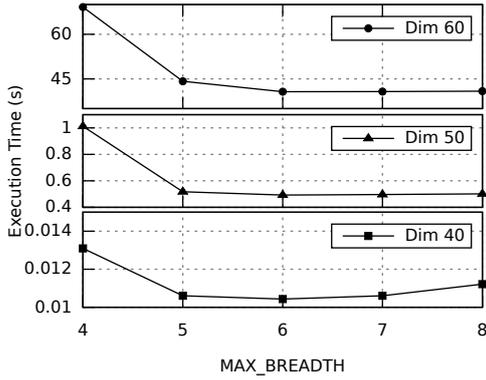


Figure 4.2: Execution time of our SE++ implementation with 16 threads solving the CVP on random lattices in dimensions 40, 50 and 60. BKZ-reduced bases with block-size 20.

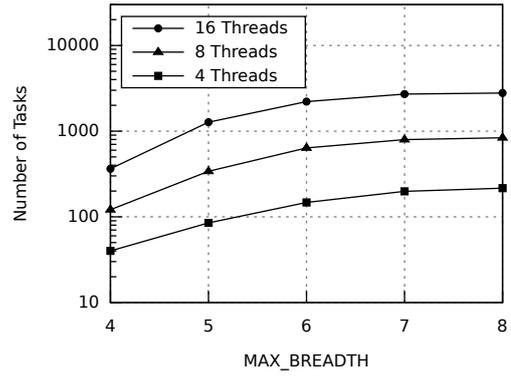


Figure 4.3: Number of tasks created by our SE++ implementation, with 16 threads, for the CVP on a random lattice in dimension 50. BKZ-reduced basis with block-size 20.

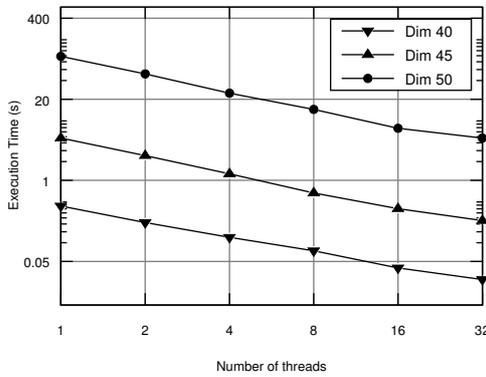


Figure 4.4: Execution time of SE++ solving the CVP on random lattices in dimensions 40, 45 and 50 for LLL-reduced bases, for 1-32 threads.

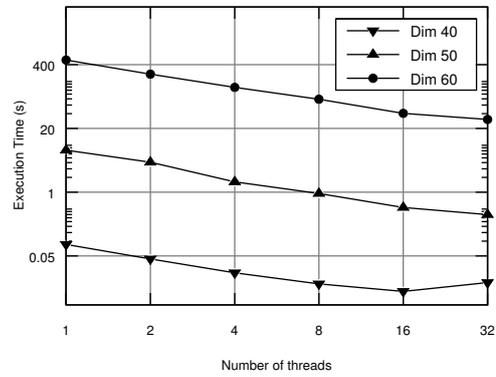


Figure 4.5: Execution time of SE++ solving the CVP on random lattices in dimensions 40, 50 and 60, for BKZ-reduced bases, for 1-32 threads.

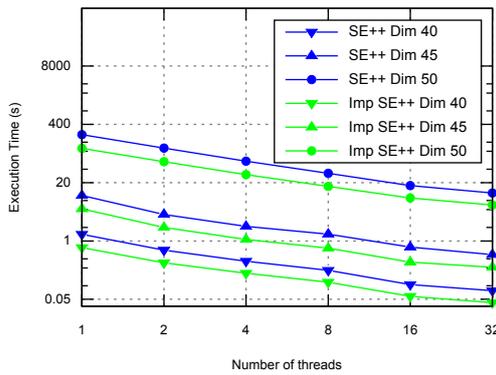


Figure 4.6: Execution time of our implementations solving the SVP on random lattices in dimensions 40, 45 and 50 for LLL-reduced bases, for 1-32 threads.

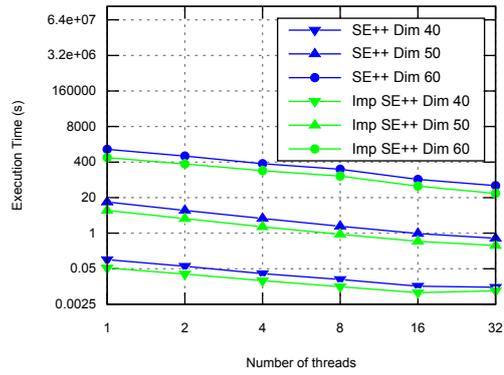


Figure 4.7: Execution time of our implementations solving the SVP on random lattices in dimensions 40, 50 and 60, for BKZ-reduced bases, for 1-32 threads.

For the SE++ (solving the SVP) and the *Improved SE++*, the number of tasks as a function of the MAX_BREADTH is similar. The higher the value of MAX_BREADTH, the higher the number of tasks that are created. We set MAX_BREADTH to 6, since performance was slightly better than for other values, despite of creating many more tasks than MAX_BREADTH = 5. Since the difference between the number of created tasks is much higher than the difference between the execution time, it is possible to conclude that the task queue of the OpenMP runtime system is very efficient. To choose the best values for MAX_DEPTH, the level at which tasks are created was set manually. For each level, we measured the execution time of all tasks and compared it to the total execution time of the algorithm.

To ensure linear and super-linear speedups, the percentual execution time of the heaviest task has to be lower than $\frac{1}{\#Threads}$. Many values of MAX_DEPTH as a function of the thread count were tested, but we observed that $MAX_DEPTH = n - \log_2(\#Threads)$ offered the best load balancing. In particular, we observed that the sweet spot of load balancing and granularity of the tasks is achieved for this combination of parameter values.

We tested the SE++ and the *Improved SE++* with LLL- and BKZ-reduced bases (BKZ ran with block-size 20). For LLL-reduced bases, our implementations were tested with lattices in dimensions 40, 45 and 50. For BKZ-reduced bases, they were tested with lattices in dimensions 40, 50 and 60, since they run much faster on BKZ-reduced bases (typically, the stronger the reduction of the lattice basis, the faster the SVP run afterwards is). Figure 4.4 shows the execution time of SE++, for the CVP, running with 1-32 threads⁴, with LLL-reduced bases, and Figure 4.5 shows the same tests for BKZ-reduced bases. Figures 4.6 and 4.7 show the execution time, under the same conditions, for the SVP, of SE++ and the *Improved SE++* (which includes the optimization that avoids symmetric branches), on LLL- and BKZ-reduced bases, respectively.

There is a number of conclusions to be drawn. First, SE++ scales linearly for up to 8 threads and almost linearly for 16 threads, for both the CVP and the SVP. The implementation can also benefit from Simultaneous Multi-threading (SMT). Second, BKZ-reduced bases are much faster to compute, both for the CVP and SVP, than LLL-reduced bases. Last, but not least, our implementation solves the CVP much faster than the SVP, but the results depend on the target vector and on the tested lattice. In our experiments, we used a target vector $\mathbf{t} = \mathbf{sB}$, where $\mathbf{s}_i = 0$ for $i = 1, \dots, n/2$, and $\mathbf{s}_i = 0.75$, for $i = n/2 + 1, \dots, n$, and \mathbf{B} is the basis of the lattice. This vector is neither too close nor too far away from the basis vectors.

A few points need to be addressed regarding the scalability of our implementations. In the first place, and as in [29], the implementations might possibly execute a smaller workload than the sequential executions would. This may occur because some threads may find a vector that is strictly closer to \mathbf{r} at an earlier point in time than the sequential execution would. This justifies the super-linear speedups that are achieved for some cases, such as for SE++ solving the CVP, with a BKZ-reduced 50-dimensional lattice, using four threads.

For the remaining cases, efficiency levels of $>90\%$ are attained for the majority of the instances of up to 8 threads, except for lattices in dimension 40, where the workload is too small to outweigh the creation and management of more than 4 threads. With 32 threads, the scalability is, in most cases (e.g. CVP on lattice 40, in Figure 4.5), lower than for up to 16 threads, presumably because of the use of two CPU sockets, which is naturally slower than the use of a single socket, due to the Non-Uniform Memory Access (NUMA) organization of the RAM (note that for 16 threads it is already lower, and the maximum

⁴We used the parallel version running with a single thread as a single-core baseline, which is 5% slower than the pure-sequential version.

scalability is achieved for 4 and 8 threads). In addition, Figures 4.6 and 4.7 show that the *Improved SE++* outperforms SE++ for the SVP by a factor of $\approx 50\%$, at the same time similar scalability is achieved.

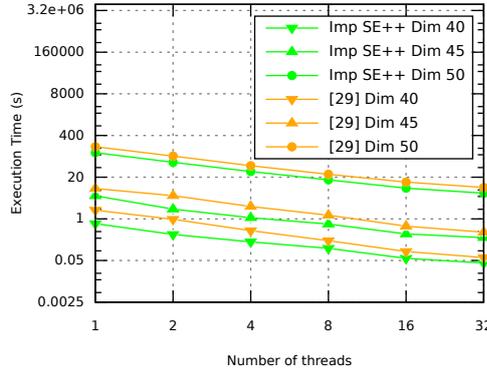


Figure 4.8: Execution time of our implementations and the implementation in [29], for the SVP on LLL-reduced lattices.

We compared our implementation against the state of the art parallel implementation of an enumeration algorithm for the SVP. This implementation, which we call *Dagdelen2010*, was proposed by Dagdelen et al., in [29], and was claimed to achieve linear and super-linear speedups, depending on the tested instance. Figure 4.8 shows a comparison between the *Improved SE++* and the *Dagdelen2010* implementation, for the SVP on three random lattices. In general, our implementation scales better. For the lattice in dimension 45, our implementation executes less workload than the *Dagdelen2010* implementation. In general, with one thread, SE++ is slower than *Dagdelen2010*, by a factor of 10% to 25%, even though it was 25% faster for the lattice in dimension 40 (these results are not in Figure 4.8, though). However, the *Improved SE++* outperforms [29] by a factor of 35% to 60%, as also shown by Figure 4.8, thus being considered the fastest deterministic enumeration-based solver when published.

4.2.2 An ad hoc demand-driven mechanism (on ENUM)

Although the approach described in Section 4.2.1 has shown satisfactory results, it does not scale well for low lattice dimensions (e.g. dimension 30). However, in lattice basis reduction algorithms, such as BKZ, enumeration is often executed on low dimensional lattices (e.g. between 20 and 35). Given that, in general, enumeration accounts for the biggest chunk of execution time in BKZ and lends itself very well to parallelism, enumeration methods that scale well even on low dimensions are essential building blocks for parallel implementations of BKZ. We believe that the reason why the scalability of the OpenMP-based implementation presented in Section 4.2.1 is reduced for low dimensions is because the computation from the root to the point where tasks actually start is repeated, for all threads, given that to compute the actual task, information from the parents is required. Note that the `MAX_DEPTH` parameter increases with the number of threads. For low dimensions, this may mean that a large percentage of the entire enumeration tree is executed by multiple tasks/threads, as there may be little computation from the `MAX_DEPTH` level till the leaves. This can be understood more clearly by comparing two trees, of a low and a high dimension, say 30 and 80, where tasks start at the same `MAX_DEPTH` level; while in dimension 30, a high `MAX_DEPTH` may mean very small tasks, the same `MAX_DEPTH` may mean very big tasks for dimension 80.

This could be mitigated or even fixed if each task would encapsulate information regarding its starting point in the tree and auxiliary information needed to start at that particular point. With the work-sharing mechanism, implemented with OpenMP tasks, this would be impractical to accomplish, given that so many tasks are created that memory usage would be prohibitive. An alternative would be to create new tasks only when the previous tasks are finished, but this is not possible to accomplish with OpenMP, at least in an efficient manner. Moreover, with OpenMP, there is no practical way to recycle memory at the task level.

To solve this two-fold problem, we propose an ad hoc demand-driven mechanism, implemented with POSIX threads (pThreads). The benefit of this model, in comparison to the OpenMP-based implementation, is that tasks do not duplicate computation, as they start execution at the right position in the tree (this information is encapsulated in each task). This is only possible with this model because new computation (the analogue of a task in the previous model) is only issued when the previously assigned computation is done, as the master thread can be held still until this is verified. With OpenMP, on the contrary, we have to issue all the tasks at once which would either eat up memory (if we encapsulated the information so each task starts at the right position in the tree) or duplicate computation (as each task starts at the root of the tree and computes all the nodes in between). This leads on to a demand-driven model, i.e. worker threads execute tasks and notify the master thread for additional tasks when the previous ones are completed.

As the model operates at the thread level, memory is easy to recycle too, as each thread receives the point in the tree where it is supposed to start at (along with the data necessary to do so), whenever new computation is issued. The model works with a master thread, which issues and assigns computation to other (worker) threads, which in turn are responsible for asking the master thread for additional work, using POSIX signals. Computation is encapsulated in the very same form as in the OpenMP-based implementation, i.e. the `MAX_BREADTH` and `MAX_DEPTH` parameters are used in the very same way, as they are effective in balancing the computation.

The following code illustrates the implementation of the model.

```

LOOP:
if (SPAWNED_THREADS < N_THREADS) {
    //copy data structures from the master thread to
    //the worker thread's private data structures
    pthread_create(&threads[SPAWNED_THREADS], ...);
    SPAWNED_THREADS++;
}
else{
    free_workers = 0;
    for(int i=0;i<N_THREADS;i++){
        if(ready[i] == 1)
            free_workers = 1;
    }
}
if(free_workers == 0){
    pthread_cond_wait(&(master_cond), &(shared_lock));
}

```

```

for(int i=0;i<N_THREADS;i++){
    if(ready[i] == 1){
        //copy data structures from the master thread
        //to the worker thread with id = i
        ready[i] = 0;
        pthread_mutex_lock(&(lock[i]));
        pthread_cond_signal(&(cond[i]));
        pthread_mutex_unlock(&(lock[i]));
    }
}
if(no_more_tasks){
    goto TERMINATE;
}
goto LOOP;

```

First, the master thread spawns `N_THREADS`, assigning them different parts of the tree. Once `N_THREADS` are running and no worker thread is free (which the master thread knows using the “ready” array), the master thread waits on a wait condition (aka `pthread_cond_wait`). The worker threads signal the master thread when they finish the computation they were assigned, and block on private wait conditions, as shown below. The master thread iterates once more on the tree and prepares to assign work. This is done by checking which threads are ready (conversely, which threads had signalled it). Note that the master thread checks all worker threads, because it may happen that more threads signal the master thread for more work before the master thread starts to assign work. The master thread then assigns tasks (i.e. copies the data structures of the *new tasks* to the worker thread’s private data structures) to the worker threads that had signalled it, signals those very same threads and blocks on the condition once again (note the `LOOP` label at the end). This is repeated until no tasks are available, when the code jumps to the `TERMINATE` label. Here, a global variable “terminate” is set to 1 (if the variable is set to 1, the worker threads will die) and the master thread waits for the worker threads with a join command (this code is not shown above). The worker threads always check this global variable, dying when the variable is 1 (and signalling the master thread for more work otherwise), as shown down below. Each worker thread runs the following pseudo-code:

```

NEW_TASK:

//compute task (code is omitted on purpose)

if(terminate == 0){
    pthread_mutex_lock(&(shared_lock));
    ready[id] = 1;
    pthread_cond_signal(&(shared_lock));
    pthread_mutex_unlock(&(shared_lock));
    pthread_cond_wait(&(cond[id]), &(lock[id]));
    goto NEW_TASK;
}

```

```
}
```

```
pthread_exit(NULL);
```

Essentially, when worker threads are spawned, they execute one task (in the pseudo-code above, this is omitted on purpose). Then, they check whether the “terminate” flag is set to 0. After executing the task, if the “terminate” flag is set to 0, the worker threads block until the master thread signals them, which means that new work has been assigned. Assigning work means to copy the data structures and information pertaining to the new task. The master thread will only signal a worker thread that has requested work; this is controlled with the “ready” array, which worker threads write on. Figure 4.9 shows the scheme in practice, for 2 threads and 2 work requests.

Optimizations

Initially, our implementation had only one task ready to assign to worker threads. In this model, it can happen that work assignment is serialized. For instance, if two threads requested work in a short time frame, the master thread could only serve the first thread, and the second thread would have to wait for the master thread to prepare another task. We improved this with a task list, i.e. the master thread prepares many tasks at once, and stores them in a list so that different threads can be served at the same time, if need be. This lowers the latency of work assignment. The task list has only as many tasks as running threads, so memory usage is not exaggerated. We implemented the task list with a circular array, to make sure that task preparation and task assignment follow the same order.

Another improvement that made sense during the refinement of our implementation was to replace a global lock that the master thread has during its execution, until it stops at the wait primitive. This is not necessarily inefficient for the model with one task (i.e. without a task list), because even with multiple work requests, the master thread could only serve one thread at a time. However, as we implemented a task list, it matters whether more than one thread can signal the master thread for more work. Therefore, we implemented a binary semaphore so that multiple worker threads can signal the master thread for more work. Put simply, the master thread sets the semaphore value to 0 and waits until it becomes 1 (which happens when some thread requests more work). The worker threads execute the signal operation, i.e. they set the value of the semaphore to 1 and signal the master thread, which awakes (in case it is sleeping) and assigns work. This process goes on till all tasks are executed.

Results

We applied this model to ENUM, as the ultimate goal is to integrate this implementation in BKZ. Not only is ENUM easier to integrate in BKZ than SE++, as it is also more efficient, as the Gram-Schmidt orthogonalization is also computed in LLL, and thus recycled for the ENUM call within BKZ. Table 4.1 shows a comparison between both implementations, where the OpenMP-based scheme is also applied to ENUM, i.e. SE++ was not used in these tests. In general, the ad hoc demand-driven mechanism scales better than the OpenMP-based implementation, and attains super-linear speedups in some instances (e.g. we obtained super-linear speedups for dimension 50, for 2-16 threads). Figure 4.11 shows the scalability of both implementations running with 64 threads, on Lara (cf. Table 2.1). The demand-driven mechanism scales better until dimension 50, which is essentially justified by (i) tasks starting at the right place in the

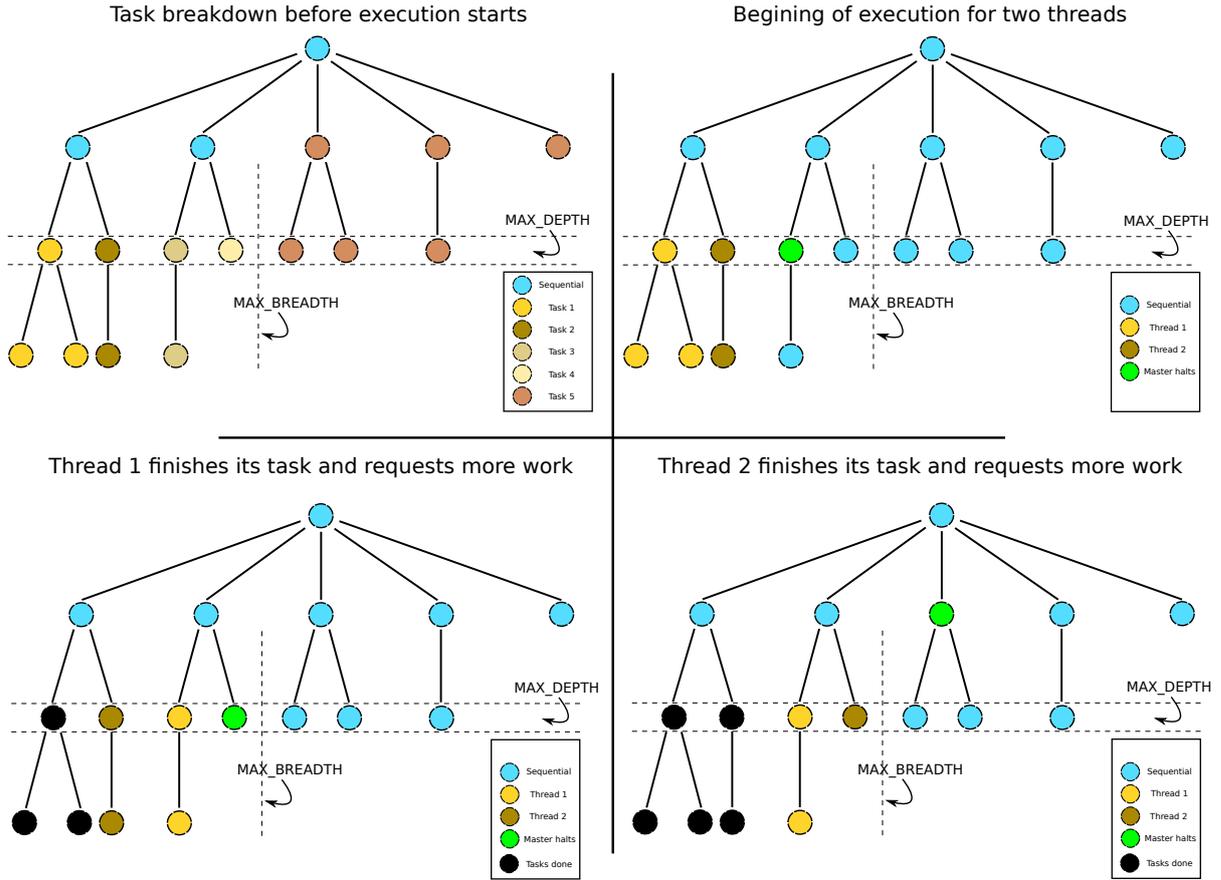


Figure 4.9: Example of the ad hoc demand-driven mechanism for two threads. Upper-left figure: original task breakdown. Upper-right figure: beginning of execution for two threads (each executes the computation that corresponds to a task). Bottom-left figure: thread 1 finishes its task and asks the master thread for additional work. Bottom-right figure: thread 2 finishes its task and asks the master thread for additional work.

	Demand-driven ENUM				OpenMP-based ENUM			
Threads	30	40	50	60	30	40	50	60
2	1.65	2.00	2.01	1.97	1.20	1.91	2.01	1.92
4	2.34	3.92	4.05	3.94	1.72	3.93	3.86	3.83
8	2.27	6.41	8.14	7.74	1.84	6.82	7.43	7.82
16	1.90	9.65	16.01	15.06	1.10	11.56	15.42	15.19
32	1.84	10.19	28.79	25.81	0.51	16.59	29.20	28.88
64	1.28	10.24	37.93	39.00	0.06	1.06	38.63	38.54

Table 4.1: Scalability of the proposed demand-driven and OpenMP-based ENUM implementations, for lattices in dimensions 30, 40, 50 and 60. Grayed-out cells represent better scalability. Tests on Lara.

tree, as opposed to starting at the root, in the work-sharing version and (ii) in contrast to the work-sharing version, where threads are created and discarded later on, if previous tasks have found better vectors, in the demand-driven version such tasks are not even created.

It should be noted that the optimal values for the `MAX_BREADTH` and `MAX_DEPTH` change with this implementation. We determined that 10 is the best `MAX_BREADTH` in the experiments with this implementation, for which we ran auxiliary experiments. The best `MAX_DEPTH` values

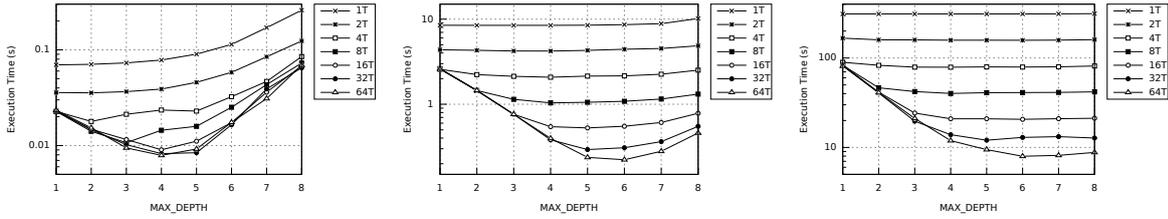


Figure 4.10: Various MAX_DEPTH values for 1 to 64 threads, for demand-driven parallel ENUM respectively for dimensions 40, 50 and 60.

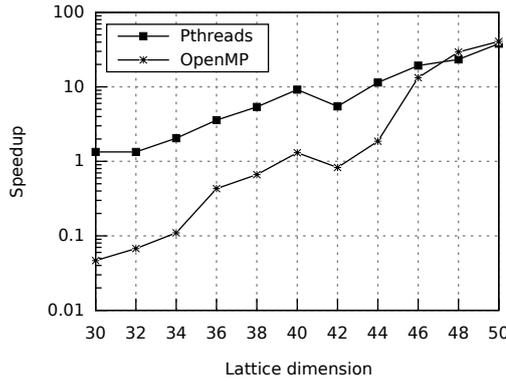


Figure 4.11: Execution time of both parallel enumeration implementations running with 64 threads, for lattice dimensions 30-50, on Lara.

vary upon the number of threads and dimension, as shown in Figure 4.10. This is explained by the overhead difference between the implementations, and the order in which tasks are assigned to threads, which ultimately dictates the optimality of the parameters. The difference of overhead between both implementations happens because (1) the implementations incur different overhead in assigning work (thread signalling vs OpenMP run-time task system), (2) in the OpenMP implementation, considerable computation, depending on the lattice dimension, is duplicated, as every task starts at the root of the tree and (3) in the demand-driven implementation, more memory is copied, as each work assignment requires memory copies.

We do not include results for ENUM with extreme pruning (as used in BKZ 2.0), given that, to the best of our knowledge, there is no publicly available code that includes optimal bounding functions to extreme pruned enumeration. However, in theory, our model should also perform well with pruned enumeration, even if the enumeration tree is more imbalanced (due to pruned branches of the tree), if proper values for MAX_BREADTH and MAX_DEPTH are selected. We advise that the best values for these parameters must be determined empirically, as the extent of pruning depends greatly on the used bounding function.

4.2.3 Summary

In this chapter, we showed that enumeration algorithms can scale well on shared-memory systems. We presented two different strategies to parallelize ENUM and SE++, based on OpenMP tasks and a low-level, demand-driven mechanism implemented with POSIX threads. While the OpenMP-based mechanism delivers satisfactory results, the demand-driven mechanism delivers even better results, especially for high thread counts. This is due to two main reasons. First, tasks are only issued when previous tasks have been

done, which enables the creation of tasks that start at the right place in the tree. In OpenMP, on the other hand, all tasks have to be issued in the beginning, and it is impractical to record all the positions in the tree where tasks should start at, as this also requires big data structures, so they start at the root, thus incurring duplicated computation. With this model, with POSIX threads, memory is easy to recycle at the thread level, and no additional memory is required as the master thread always stalls at the next position in the tree to compute. Secondly, some tasks are discarded immediately by the master thread, if previous tasks have found better vectors. In the work-sharing version, implemented with OpenMP, all tasks are created, and tasks can only be discarded after their execution started.

Despite the good results of both versions (and the demand-driven mechanism in particular) for high lattice dimensions, neither version delivers satisfactory results for low lattice dimensions (e.g. 30), especially when the core count is high. This is relevant, as in lattice basis reduction algorithms, such as BKZ, enumeration is often executed on low dimensional lattices (e.g. between 20 and 35). Given that the parallelization of BKZ can be achieved at the cost of a parallel enumeration, as we show in Chapter 5, enumeration methods that scale well even on low dimensions are essential building blocks for parallel implementations of BKZ. In Chapter 5, we address this issue.

Parallel Efficient Lattice Basis Reduction

Synopsis. This chapter describes parallel variants of LLL and BKZ, the most relevant lattice basis reduction algorithms to date. We give, for the first time, a description of a vectorized LLL implementation, which is based on careful data re-arrangement that increases cache locality and enables the vectorization of two key kernels. For low dimensions (up to 100), our implementation is faster than NTL, a reference implementation for LLL, when vectorized. However, the highest performance gains are achieved with high dimensional lattices, which have enough data elements so that vectorization attains its fullest potential. A new parallel BKZ implementation is also presented. Its scalability grows with the block-size, scaling reasonably for high window sizes, delivering only very modest speedups for small window sizes, which we analyze and comment on.

"Nature may reach the same result in many ways.", Nikola Tesla, Serbian-American inventor.

5.1 Lattice basis reduction

As mentioned before, lattice basis reduction is the process of transforming a given lattice basis \mathbf{B} into another lattice basis \mathbf{B}' , whose vectors are shorter and more orthogonal than those of \mathbf{B} , and where \mathbf{B} and \mathbf{B}' generate the same lattice, i.e., $\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{B}')$. There is no single definition of what a reduced basis is. Oftentimes, it is said that the goal of lattice reduction algorithms is to yield a *nearly orthogonal* (as bases cannot always be orthogonalized) basis. All bases have a given orthogonality defect δ , which compares the product of the lengths of the vectors in the basis with the volume of the parallelepiped they define. If these quantities are equal, then $\delta = 1$ and the basis is said to be orthogonal. The orthogonality defect δ of a basis with rank m , which generates lattice \mathcal{L} , is thus given by $\delta = \prod_{i=1}^m \|\mathbf{b}_i\| / \text{vol}(\mathcal{L})$ and $\delta \geq 1$.

LLL-reduced and BKZ-reduced bases, reduced respectively with LLL and BKZ, seem to be *de facto* terms when referring to reduced bases. This is because each lattice basis reduction algorithm transforms the basis under a specific notion of reduction, and using the name of the algorithm to refer to a reduced lattice makes it clear what reduction notion was used for the transformation. Although there are more notions of reduction (and algorithms employing them), LLL and BKZ are the core references in this

area, due to their practicability and trade-off in terms of execution time and quality of the output. In the following, we describe two variants of both algorithms and their implementations.

5.2 The LLL algorithm

While it is possible to achieve the optimal notion of reduction in reasonable time for two or three dimensions, other dimensions become intractable if a strong notion of reduction is applied [22]. The LLL algorithm, named after its authors Lenstra, Lenstra and Lovász, is a breakthrough in this area, as it achieves a new notion of reduction algorithm with linear time complexity.

In this section, we analyze LLL purely from a computational standpoint, refraining from delving into the mathematics. For a complete description and analysis of the algorithm, the reader is referred to literature such as [22, Section 2.6], [125, Chapter 17] and the LLL original paper [65]. LLL lays the foundation for many other algorithms and has applications in many fields in computer science, ranging from testing conjectures to solving quadratic equations and of course solving lattice problems [118]. In the context of lattice-based cryptanalysis, LLL is relevant because (i) it can reduce a given basis in polynomial time, (ii) the first vector of the reduced basis is very short and (iii) it serves as an essential building block and inspiration to other very important algorithms, such as BKZ, which we present in Section 5.3.

In the following, we present some concepts of LLL, taken from [114]. A given lattice basis $\mathbf{b}_1, \dots, \mathbf{b}_m \in \mathbb{Z}^n$ has an associated Gram-Schmidt orthogonalization $\hat{\mathbf{b}}_1, \dots, \hat{\mathbf{b}}_m \in \mathbb{Z}^n$, which is computed from the basis $\mathbf{b}_1, \dots, \mathbf{b}_m \in \mathbb{Z}^n$ and the so-called Gram-Schmidt coefficients $\mu_{i,j} = \langle \mathbf{b}_i, \hat{\mathbf{b}}_j \rangle / \langle \hat{\mathbf{b}}_j, \hat{\mathbf{b}}_j \rangle$, with the following recursive formula:

$$\hat{\mathbf{b}}_1 = \mathbf{b}_1, \hat{\mathbf{b}}_i = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \hat{\mathbf{b}}_j, \text{ for } i = 2, \dots, m. \quad (5.1)$$

and $\mu_{i,i} = 1$ and $\mu_{i,j} = 0$ for $i < j$. The vectors $\hat{\mathbf{b}}_1, \dots, \hat{\mathbf{b}}_m \in \mathbb{Z}^n$ are linearly independent, but they are not necessarily lattice points. Let δ be a constant such that $\frac{1}{4} \leq \delta \leq 1$. A basis is LLL-reduced with δ if it is size-reduced and if

$$\delta \|\hat{\mathbf{b}}_{k-1}\|^2 \leq \|\hat{\mathbf{b}}_k + \mu_{k,k-1} \hat{\mathbf{b}}_{k-1}\|^2 \text{ for } k = 2, \dots, m. \quad (5.2)$$

For practical purposes, we are interested in a δ close to 1 (usually it is set to 0.99). In the next subsection, we present an LLL floating point version, introduced by Schnorr and Euchner in [114], and some modifications we have made to implement it.

5.2.1 Floating-point LLL

As mentioned before, the original LLL algorithm was described with rational arithmetic, which is overly expensive. Lagarias and Odlyzko were pioneers on the implementation of LLL with rational arithmetic, but the set of experiments was too small due to the cost of rational arithmetic and limited computing power available at that time [63]. Over the years, considerable efforts were devoted to develop floating-point variants of LLL, which are considerably faster than rational variants, but also introduce errors, which must be corrected. As most of the first proposals were not provable, many researchers sprang to a

search for provable floating-point variants of LLL. The most outstanding contributions in this matter are due to Schnorr, in 1988 [115], and, more recently, Nguyễn et al. [93]. Currently, the most practical implementations of LLL are heuristics with high probability of success, such as the seminal heuristic due to Schnorr and Euchner, proposed in [114], which we revisit in this thesis. The implementation uses the exact representation of the basis vectors $\mathbf{b}_1, \dots, \mathbf{b}_m \in \mathbb{Z}^n$ and a floating point representation of the Gram-Schmidt coefficients $\mu_{i,j}$ and the square norms of the Gram-Schmidt vectors $\|\hat{\mathbf{b}}_i\|^2$. \mathbf{v}' represents the floating point value of a given \mathbf{v} exact value. The integer τ represents the number of bits used in the implementation.

The basis has to be represented exactly because errors in the basis change the lattice, and thus cannot be reverted. To represent the basis exactly, it is necessary to have a multiple precision mechanism to represent such numbers and perform arithmetic operations on them¹. All the remaining errors can be corrected using the (exact) basis. The LLL version proposed by Schnorr and Euchner includes some mechanisms that mitigate the floating point errors (cf. [114, Section 3]). In particular:

- 1) Whenever the algorithm enters stage k , it computes $\mu_{k,j}$ for $j = 1, \dots, k-1$ and $c_k = \|\hat{\mathbf{b}}_k\|^2$, using the basis vectors $\mathbf{b}_1, \dots, \mathbf{b}_k$. This corrects possible errors in $\mu_{k,j}$ and c_k , since the basis vectors are exact.
- 2) If a large reduction coefficient $|\lceil \mu_{k,j} \rceil| > 2^{\tau/2}$ occurs when reducing \mathbf{b}_k , we decrease the stage k to $k-1$. This corrects the coefficients $\mu_{k-1,j}$ and $\mu_{k,j}$ for $j = 1, \dots, k-1$ as well as c_{k-1} , c_k , \mathbf{b}'_{k-1} , \mathbf{b}'_k .
- 3) If $|\langle \mathbf{b}'_k, \mathbf{b}'_j \rangle| < 2^{-\tau/2} \|\mathbf{b}'_k\| \|\mathbf{b}'_j\|$, then the algorithm computes $\langle \mathbf{b}_k, \mathbf{b}_j \rangle'$ instead of $\langle \mathbf{b}'_k, \mathbf{b}'_j \rangle$. Since the leading bits in the computation of $\langle \mathbf{b}'_k, \mathbf{b}'_j \rangle$ cancel out, the value $\langle \mathbf{b}'_k, \mathbf{b}'_j \rangle$ is too inexact.

We implemented this floating-point LLL variant, with a few changes. This implementation is also the basis of our work in [71]. Algorithm 1 shows the pseudo-code of our implementation. The input of the algorithm is the lattice basis and a reduction parameter δ , which defines the extent of the reduction.

The algorithm starts at stage $k = 2$, by computing the Gram-Schmidt orthogonalization (lines 10-19 in Algorithm 1), which starts with the computation of the inner product between two vectors. If the precision loss is too high, the exact dot product has to be computed, for which we use the exact version of the basis. The Gram-Schmidt orthogonalization outputs the approximate values of one row of the coefficient vectors of the orthogonal basis, μ_k , and the square norm of the corresponding orthogonal vector.

The next step is a size-reduction procedure of the vector \mathbf{b}_k with all vectors \mathbf{b}_j , for $j = k-1, \dots, 1$ (lines 21-34 in Algorithm 1), if the size-reduction is possible. This procedure consists in subtracting the coordinates of one vector by another, whose coordinates are multiplied by a constant i.e. $(\mathbf{b}_k = \mathbf{b}_k - \lceil \mu_{k,j} \rceil \times \mathbf{b}_j)$. If $|\mu_{k,j}| > 1/2$ holds true, it is possible to perform a size-reduction. If the reduction takes place, we approximate the k -th row of the basis.

Finally, the reduced vector will be swapped with its predecessors unless the Lovász condition holds (lines 36-43 in Algorithm 1). This condition ensures that successive vectors are at least δ times bigger than their respective predecessor. The described process is repeated for each vector in the basis, until all vectors are LLL-reduced. Once this condition is verified, an LLL-reduced basis with δ is returned. To improve the numerical stability and the performance of the variant, we modified it as follows:

¹In this thesis, we use GMP, a library for arbitrary precision arithmetic, available from <https://gmplib.org/>

Algorithm 1: The heuristic LLL algorithm with floating point arithmetic, proposed by Schnorr and Euchner [114]. The lines in blue differ from the original algorithm.

Input: A basis $(\mathbf{b}_1, \dots, \mathbf{b}_n)$, $\delta \in (1/2, 1)$ and all $\mu_{i,j}$ and $c_i = \|\hat{b}_i\|^2$, computed by the Gram Schmidt orthogonalization.

Output: An LLL-reduced basis with δ .

```

1   $k = 2$ ;
2  //number of precision bits in double precision
3   $\tau = 53$ ;
4   $Fc = false$ ;
5   $last\_k = 0$ ;
6  for  $i = 1, \dots, m$  do
7  |  $\mathbf{b}'_i = (\mathbf{b}_i)'$ ;
8  while  $k \leq m$  do
9  | //Computation of  $\mu_{k,1}, \dots, \mu_{k,k-1}$  and  $c_k$ 
10 |  $c_k = \|\mathbf{b}'_k\|^2$ ;
11 | if  $k = 2$  then
12 | |  $c_1 = \|\mathbf{b}'_1\|^2$ ;
13 | for  $j = 1, \dots, k-1$  do
14 | | if  $\langle \mathbf{b}'_k, \mathbf{b}'_j \rangle^2 < 2^{-2 \times \tau \times 0.15} \|\mathbf{b}'_k\| \|\mathbf{b}'_j\|$  then
15 | | |  $s = \langle \mathbf{b}_k, \mathbf{b}_j \rangle'$ ;
16 | | | else
17 | | | |  $s = \langle \mathbf{b}'_k, \mathbf{b}'_j \rangle$ ;
18 | | |  $\mu_{k,j} = (s - \sum_{i=1}^{j-1} \mu_{j,i} \mu_{k,i} c_i) / c_j$ ;
19 | | |  $c_k = c_k - \mu_{k,j}^2 c_j$ ;
20 | do
21 | |  $Fc = false$ ;
22 | | for  $j=k-1, \dots, 1$  do
23 | | | if  $|\mu_{k,j}| > 1/2$  then
24 | | | |  $Fc = true$ ;
25 | | | | for  $i = 1, \dots, j-1$  do
26 | | | | |  $\mu_{k,i} = \mu_{k,i} - \lceil \mu_{k,j} \rceil \times \mu_{j,i}$ ;
27 | | | | |  $\mu_{k,j} = \mu_{k,j} - \lceil \mu_{k,j} \rceil$ ;
28 | | | | |  $\mathbf{b}_k = \mathbf{b}_k - \lceil \mu_{k,j} \rceil \times \mathbf{b}_j$ ;
29 | | | | | if  $Fc$  then
30 | | | | | |  $\mathbf{b}'_k = (\mathbf{b}_k)'$ ;
31 | | | | | | if  $k > last\_k$  then
32 | | | | | | |  $Recompute\_GS()$ ;
33 | while  $Fc$ ;
34 | //Swap  $b_{k-1}, b_k$  or increment k
35 | if  $\delta c_{k-1} > c_k + \mu_{k,k-1}^2 c_{k-1}$  then
36 | | swap( $\mathbf{b}_k, \mathbf{b}_{k-1}$ );
37 | | swap( $\mathbf{b}'_k, \mathbf{b}'_{k-1}$ );
38 | | if  $k > last\_k$  then
39 | | |  $last\_k = k$ ;
40 | |  $k = \max(k - 1, 2)$ ;
41 | else
42 | |  $k = k + 1$ ;
43 return  $(\mathbf{b}_1, \dots, \mathbf{b}_n)$  ;

```

1. As in the NTL implementation, we replaced the 50% precision loss test of [114] by another, which tolerates a loss of up to 15% in the computation of the inner products.
2. Unlike L^3FP in [114], we check whether the values fit into a `double` data type (to compute the dot product in line 14 of Algorithm 1 with `doubles`), as we use `xdoubles` to store approximate values. If they do, we use `doubles` to compute the dot product as operations become more efficient than on `xdoubles`.
3. If a given basis vector \mathbf{b}_k can be reduced, Schnorr et al. test whether the precision loss is too high. If so, the algorithm tries to reduce \mathbf{b}_k again. However, in our implementation, \mathbf{b}_k is always reduced again, even when the precision loss is low (lines 21-34 in Algorithm 1). This is also how the algorithm is implemented in NTL. In addition, we also recompute the Gram-Schmidt orthogonalization the first time \mathbf{b}_k is reduced, since errors may occur that are hard to correct at a later stage (lines 32-33 in Algorithm 1; note that `Recompute_GS` executes the same code as lines 10-19).

LLL requires multiple precision capability to handle large coordinates, which are common in most lattices available from the SVP-Challenge². One option to implement multiple precision is the GNU Multiple Precision Arithmetic Library (GMP) library. In our implementation, we used GMP to store exact values.

The extended exponent double precision data type (`xdouble`) allows to represent floating-point numbers with the same precision as a `double`, but with a much larger exponent. It is implemented as a class, where two instance variables are used, a `double` x and a `long` e , to store the mantissa and the exponent, respectively. For any given number in the form $x \times b^e$, x denotes the mantissa, b the base and e the exponent.

The data structures of our implementation consist of 2-dimensional arrays, of either `xdoubles` for floating-point arithmetic (Gram-Schmidt coefficients μ and the approximate basis \mathbf{B}'), or the GMP `mpz_t` data type for exact arithmetic (exact basis \mathbf{B}), for matrices. For vectors, we used 1-dimensional arrays containing `xdoubles` (square norms of the Gram-Schmidt vectors - no vectors with exact precision are needed). In addition, two `xdouble` arrays are used to store the square norms of the approximated basis vectors (used in line 14 of Algorithm 1) and the result of $\mu_{k,i}c_i$ (computed in line 18 and needed again in line 19 of Algorithm 1).

5.2.2 Performance

Figure 5.1 shows the performance of `pLLL`, NTL, `fpLLL` and our implementation, on Lara (cf. Table 2.1). The execution time alone does not provide enough data to conclude which implementation is the best, as the final basis is different (all implementations use different variants of LLL). Therefore, Figure 5.1 should be read together with Figures 5.2-5.7, which show the quality of the different implementations, for the criteria we enumerated in Section 2.1.1. As shown in Figure 5.1, `pLLL` is much slower than NTL, `fpLLL` and our implementation, and `fpLLL` is significantly faster than both our implementation and NTL. There is a small difference between our implementation and NTL, although NTL is faster for the majority of the cases and our implementation does not terminate from dimension 194 onwards. This is because our

²www.latticechallenge.org/svp-challenge/

error correction mechanisms are not as complete as NTL's or fpLLL's, and we plan to revisit this topic in the future. However, it should be noted that our implementation serves as an excellent tool to carry out studies about the quality of the output bases and the room for performance optimization in LLL. Due to being considerably slower than the other libraries, pLLL is not be used for further tests throughout this dissertation.

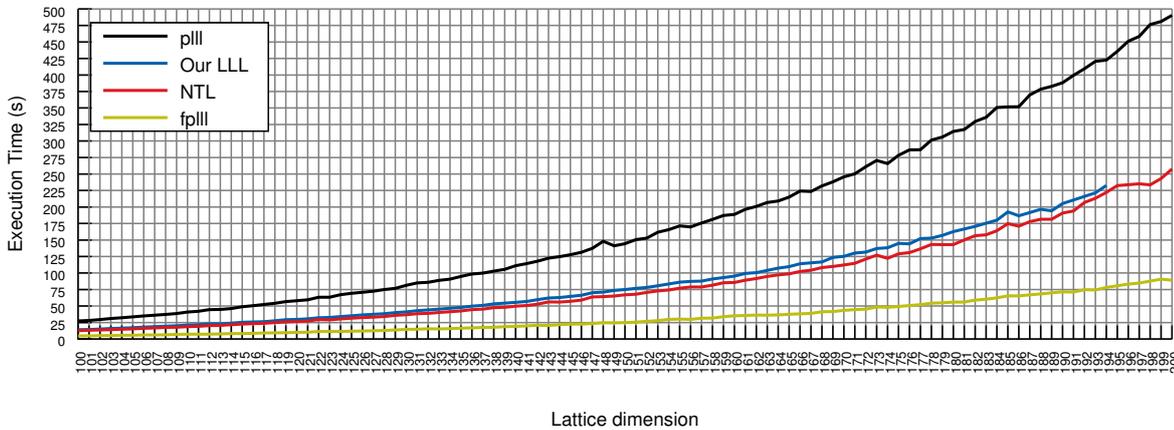


Figure 5.1: Execution time of LLL implementations in pLLL, NTL, fpLLL, and our LLL implementation, for lattices between dimension 100 and 200, on Lara.

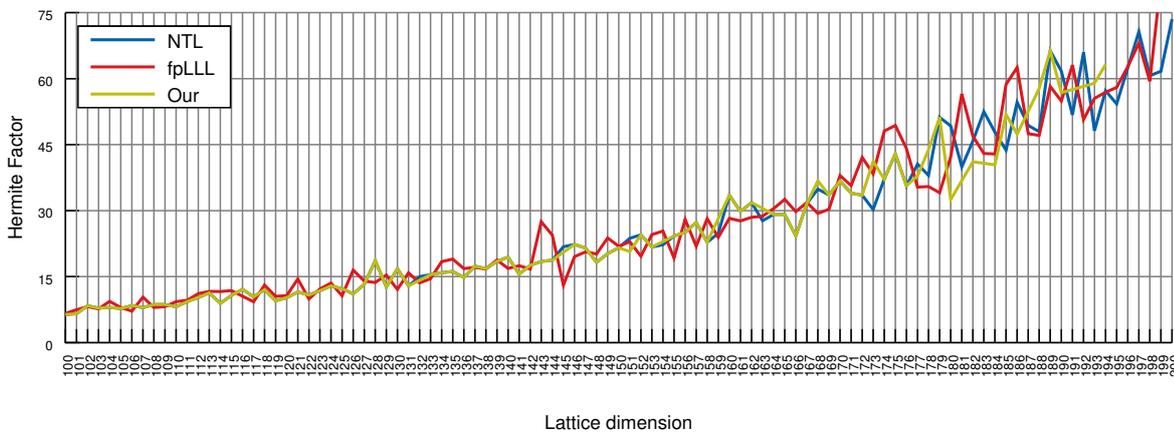


Figure 5.2: Hermite factor of LLL implementations in NTL, fpLLL, and our LLL implementation, for lattices between dimension 100 and 200, on Lara.

In general, both our implementation and NTL's tend to deliver better bases than fpLLL, although fpLLL output bases are better in some cases. The difference of quality between our implementation and NTL's is small. In fact, on a vast majority of lattices, the Hermite factor, the length defect, the orthogonality defect, the slope of the Gram-Schmidt curve, the norm of the last Gram-Schmidt vector and the average norm of the output vectors are identical (cf. Figure 5.2-Figure 5.7). These parameters start to diverge mostly from dimension 170 onwards, as the differences between the implementations become more significant then. We believe that due to the similarities of both implementations, the analysis and the conclusions we draw about our implementation are, to a large extent, relevant to NTL as well. Figure 5.8 shows the execution time, in seconds, of the parallel Enumeration SVP-solver presented in Section 4.2.2, running with 64

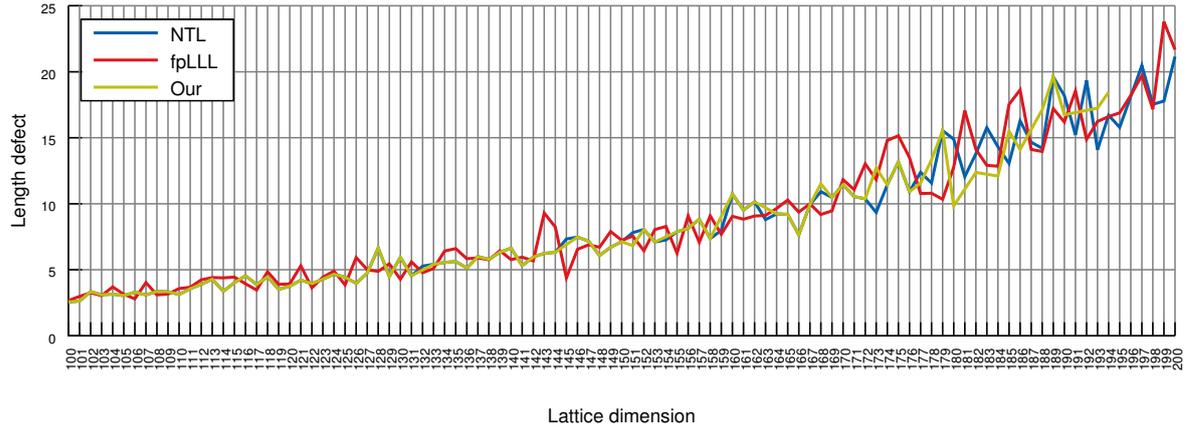


Figure 5.3: Length defect of the LLL implementations in NTL, fpLLL, and our LLL implementation, for lattices between dimension 100 and 200, on Lara.

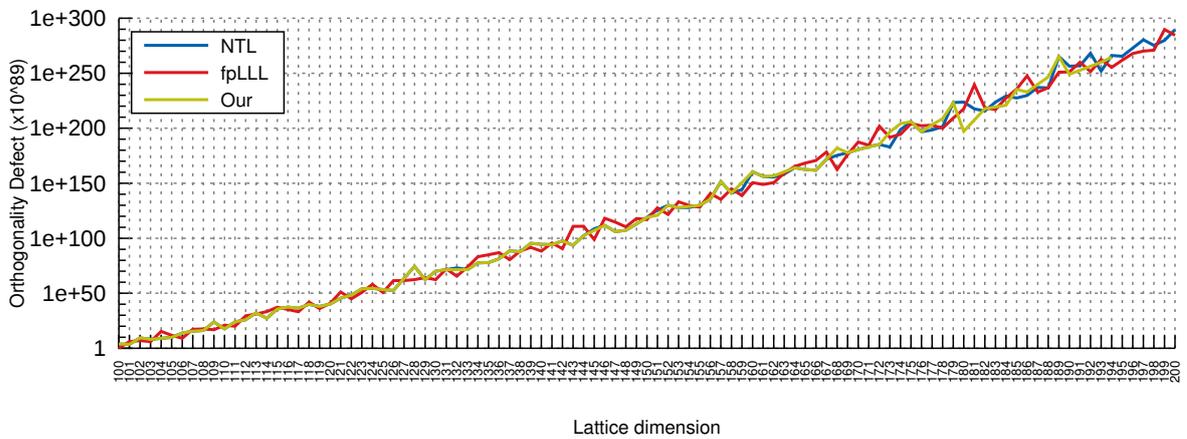


Figure 5.4: Orthogonality defect δ of the LLL implementations in NTL, fpLLL, and our LLL implementation, for lattices between dimension 100 and 200, on Lara.

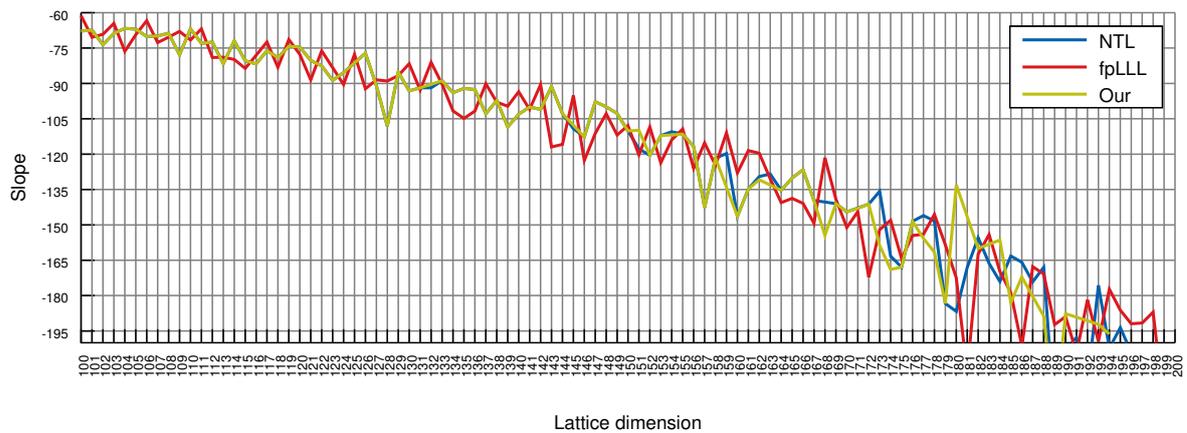


Figure 5.5: Slope of the Gram-Schmidt curve associated with the output basis of the LLL implementations in NTL, fpLLL, and our LLL implementation, for lattices between dimension 100 and 200, on Lara.

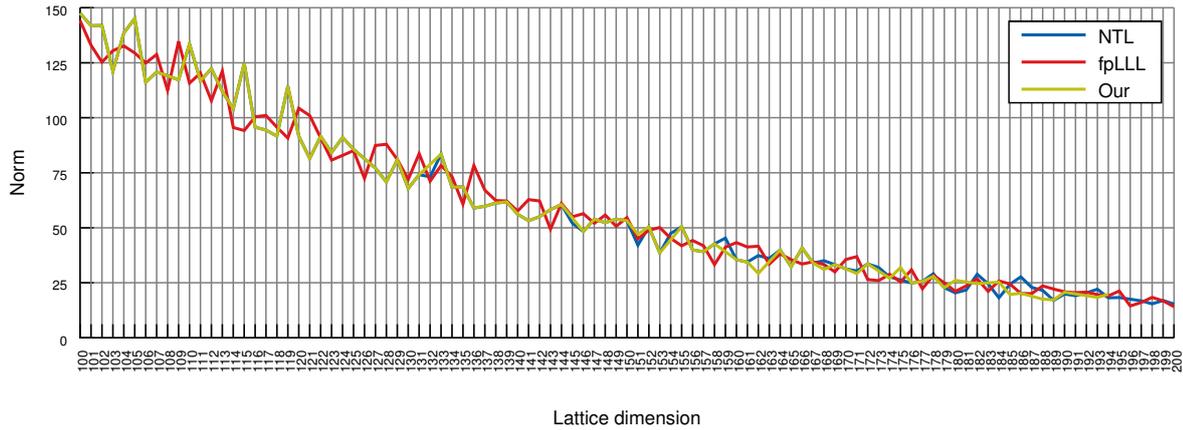


Figure 5.6: Norm of the last Gram-Schmidt vector in the output basis of the LLL implementations in NTL, fpLLL, and our LLL implementation, for lattices between dimension 100 and 200, on Lara.

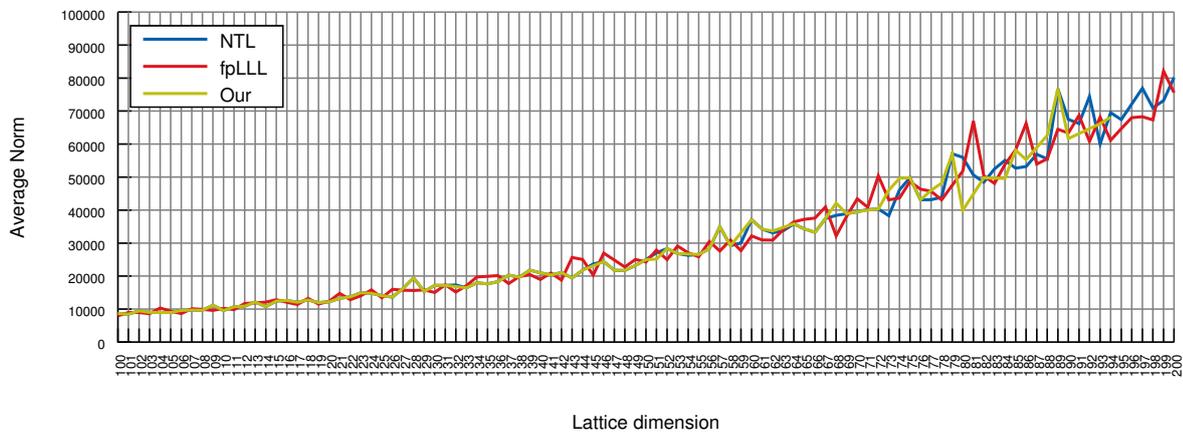


Figure 5.7: Average norm of the vectors in the output basis of the LLL implementations in NTL, fpLLL, and our LLL implementation, for lattices between dimension 100 and 200, on Lara.

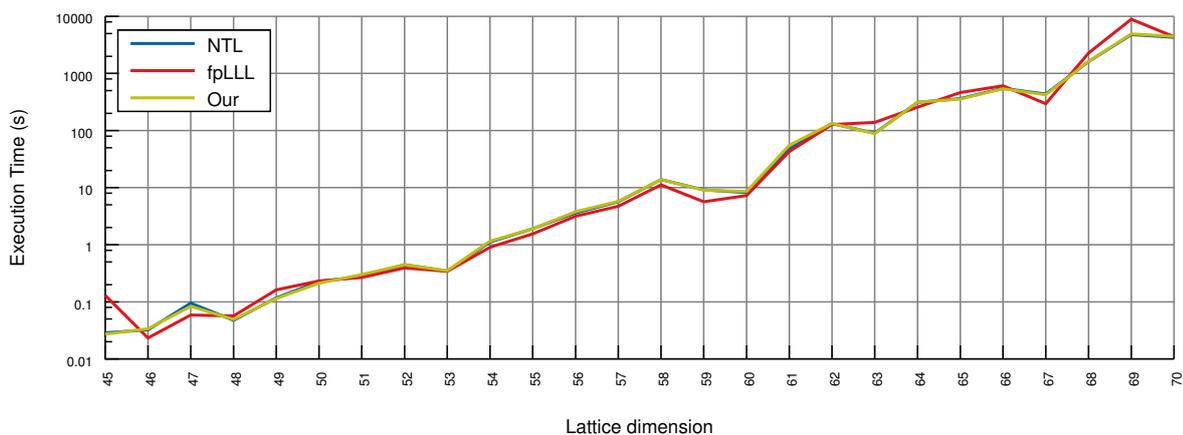


Figure 5.8: Execution time (in seconds) of the parallel ENUM, running with 64 threads, on lattices LLL-reduced by NTL, fpLLL, and our LLL implementation, for lattices between dimension 45 and 70, on Lara.

threads on Lara, on the lattices yield by NTL, fplll and our implementation. Although the SVP-solver is faster on some lattices output by fplll, it is generally faster on lattices yielded by our implementation.

5.2.3 Data structures re-organization and SIMD vectorization

Many of the ideas presented in this Section have been published in [71].

In the following, we present a re-arrangement of the data structures in our LLL implementation, so that both cache locality is leveraged and SIMD vectorization is enabled. Figure 5.9 shows the re-arrangement of the data structure to store the approximate version of the lattice basis. On the left side, we store an array of n pointers to other arrays, each of which has n elements (NTL stores data identically). Each element is stored as a `xdouble` object, which is a struct of two elements (a `double` and a `long`). On the right side, we show the data structure re-arranged, which corresponds to a transformation from an array of structs (AoS) to a struct of arrays (SoA).

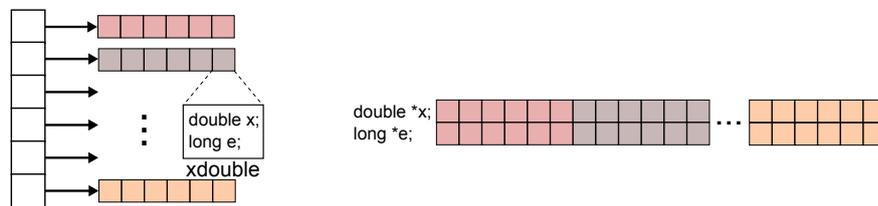


Figure 5.9: Original (left side) and re-arranged (right side) data structures.

As the original data structure is an array of structs, cache locality is low. With the re-arrangement, multiple vectors are brought to cache with two accesses (arrays `*x` and `*e`). A vector in dimension n has n coordinates of 16 bytes each (8 bytes for the `long` and 8 bytes for the `double`). Therefore, accessing array `*x` brings 8 elements to each L1 cache line, assuming a 64 bytes L1 cache line size. More elements are also brought to L2 cache, thereby reducing memory access latency in comparison to the original implementation.

We store the Gram-Schmidt coefficients μ in an identical data structure, although in the form of a lower triangular matrix. The re-arrangement is done in a similar way, as shown in Figure 5.10. The major difference is index calculation. In the new format, $\mu_{i,j}$ is accessed at the index $(i \times (i - 1) / 2) + j$, thereby incurring a slight additional overhead to access elements.

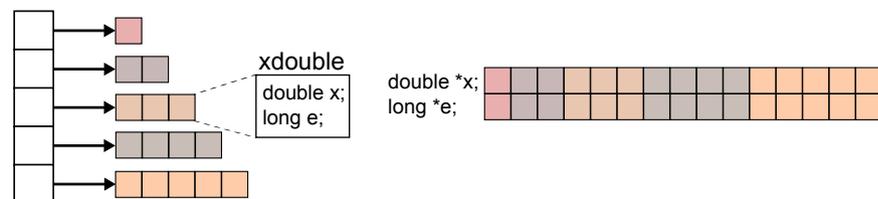


Figure 5.10: Original (left side) and re-arranged (right side) data structures.

These re-arrangements also allow one to vectorize (i) the *dot product* between two vectors when they fit in `doubles` (cf. line 5, step 2 of L^3FP in [114]) and (ii) the add and multiply (*AddMul* $a = a + b \times c$) (cf. line 8 of the same source). Note that when vectors do not fit into `doubles`, no vectorization is used

in (i), as this kernel represents a tiny percentage of the overall execution time. For (ii), we were able to partially vectorize the operation, as it is performed exclusively with `xoubles`. We split the kernel in two steps: the multiplication and the addition. First, we multiply the elements (`xoubles`) of one array by the corresponding elements of a second array, which has no dependencies and can be vectorized. In particular, the mantissas are multiplied by one another and the exponents are summed up (both operations are vectorized). Then, we sum up the partial multiplications without vectorization.

Experiments

We used NTL's implementation of LLL as a reference implementation. We note that the implementation in NTL is faster than our base implementation due to being considerably more efficient than ours in terms of Gram-Schmidt computations. We have not devoted much time to optimize this process since, although a bit slower than NTL, our implementation serves as an appropriate tool to conduct experiments on enhancing LLL through vectorization and data-structure re-arrangements. However, our main goal is to propose optimizations that can be applied to any LLL implementation (including NTL's).

We refer to our implementation as either (i) *base* implementation, for the non-optimized, implementation, (ii) *optimized/OPT*, for the version with the data structures re-arranged or (iii) *vectorized/VEC* for the version with re-arranged data structures and vectorization enabled. For 256-bit SIMD vectorization we used AVX2, while for 128-bit SIMD vectorization SSE 4.2 was used.

We used random Goldstein-Mayer lattice bases, available on the SVP-Challenge website. For tests with these lattices, we ran 50 seeds for each dimension. For tests with lattices from the Lattice-Challenge³, we run a single seed for each dimension, as no lattice generator is available. We conducted these experiments on Lara (see Table 2.1). Each core has 32 KB of L1 instruction and data cache (a cache line has 64 bytes). L2 caches have 256 KB and are not shared. The code was compiled with GNU g++ 4.8.4. We compiled the code with the `-O2` optimization flag, since it was slightly better than `-O3`.

Goldstein-Mayer lattice bases (low dimensions): We used the lattice generator to generate 50 lattices with seeds 1-50 per dimension, and thus have a statistically significant result. We ran our LLL implementation and NTL's implementation for lattices in dimensions 80-100. The performance of our *base* implementation is comparable to NTL's (it is at most 3% slower), as shown in Figure 5.11 (note the zoom-in section where the performance difference is accentuated). As shown in Figure 5.1, NTL is faster than our LLL implementation after dimension 150 (these results pertain to seed 0 only, though). Either way, the final set of experiments serves well when studying the effect of vectorization in LLL.

Two results in Figure 5.11 deserve particular attention. First, the *optimized* (OPT) version does not perform considerably better than the *base* version. We believe that the lattice dimensions we tested are too low so that cache locality gains can outweigh the overhead incurred in this version (recall that this version introduces overhead for accessing memory, as it is more complex to calculate indices). To conclude this, we:

1. measured the cache misses of our implementation at the first level of cache, as shown in Figure 5.12 (for the results with the other cache levels, the reader is referred to [71]). As the figure shows, our OPT version incurs much fewer cache misses than the base version, and in particular, the difference increases with higher lattice dimensions. This indicates that at some point (i.e. for a sufficiently large lattice dimension), the OPT version should be faster than the base version.

³<http://www.latticechallenge.org/>

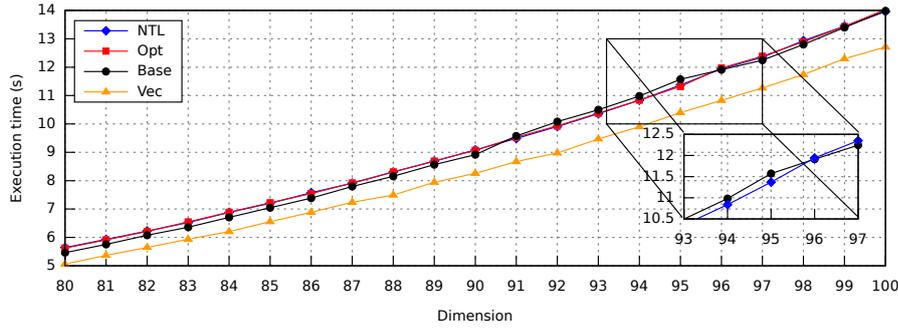


Figure 5.11: Execution time of our LLL implementation and NTL's, for lattices from the SVP-Challenge. Note the zoom-in section for Base and NTL, between dimensions 93-97.

2. tested the implementations on higher lattice dimensions, which is shown afterwards with Ajtai lattices, a different kind of lattices. Ideally, we would test all LLL implementations on Goldstein-Mayer lattices in much higher dimensions, but those would be impractical to solve; Ajtai lattices allow for such tests as they require far less computation in LLL.

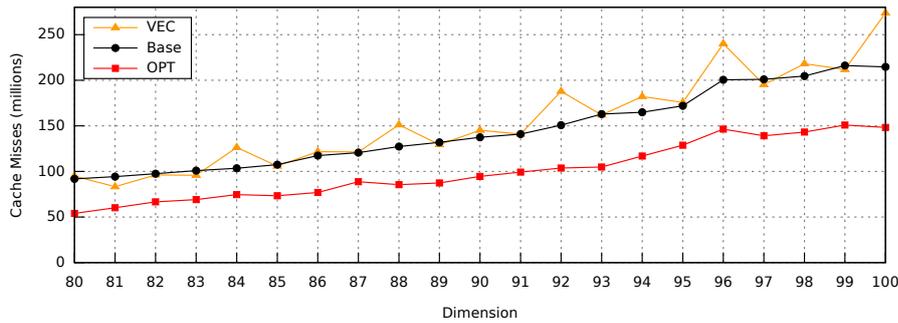


Figure 5.12: L1 cache misses (in millions) of our LLL implementation.

Second, the *vectorized* (VEC) version obtains speedups of 9-11% over the base version and NTL. This version incurs, somewhat surprisingly, more cache misses than the other versions, and even dimensions incur additional cache misses (the misses in L2 and L3 cache levels have no such pattern cf. [71]). We believe that this happens because, as performance increases, more memory accesses are performed within the same timespan, thus shortening the window of opportunity for efficient prefetching. As far as the improvement is concerned, we could obtain a theoretical maximum speedup of 4x (as we vectorize 8-byte doubles) with 256-bit SIMD vectorization, and 2x with the 128-bit SIMD vectorization, for the same reason (but for 8-byte longs). Thus, in theory, we could achieve an overall speedup of 19.5%, as the *dot product* loop takes approximately 16% of the execution time of the base version (for a lattice in dimension 100), for which we used 256-bit SIMD registers, while the *AddMul* loop takes approximately 31%, for which we used 128-bit SIMD registers⁴. A 11% speedup is in our view a good result, as the maximum number of vectorized elements is n (in this case 100, at most), which is not sufficient to achieve the full potential of vectorization. Note that it is not possible to vectorize all elements in the *AddMul* kernel given that the number of computed elements per iteration increase with the iteration.

⁴As the number of elements that are vectorized in the loop decreases, there may not be 4 elements, which are necessary to use 256-bit SIMD.

Ajtai lattices (high dimensions): For Goldstein-Mayer lattices, it is only practical to run LLL within limited dimensions, as we mentioned before. As the Lattice-Challenge is based on Ajtai lattices, it allows one to conduct benchmarks with larger lattice dimensions, given that Ajtai lattices contain far smaller numbers and LLL reduces them much faster. Note that Goldstein-Mayer lattices have numbers with over 300 digits, while Ajtai lattices have numbers with no more than 3 or 4 digits, and so computation becomes significantly cheaper.

Figure 5.13 compares our implementation against NTL’s, for lattices between dimension 200 and 800. NTL is approximately 2x faster than our base implementation, primarily because it is considerably more efficient at computing the Gram-Schmidt kernel in higher lattice dimensions. However, the key point in this subsection is not to show how our implementation compares to NTL, but what performance gain can be attained when optimizing it, or in other words, the gain obtained with vectorization. As the figure shows, we obtain a 6% speedup by simply switching to the OPT (aka with re-organized data structures) version. This backs up our claim that re-organizing the data structures delivers higher gains for higher lattice dimensions, as the previous experiments were only done for lattices up to dimension 100.

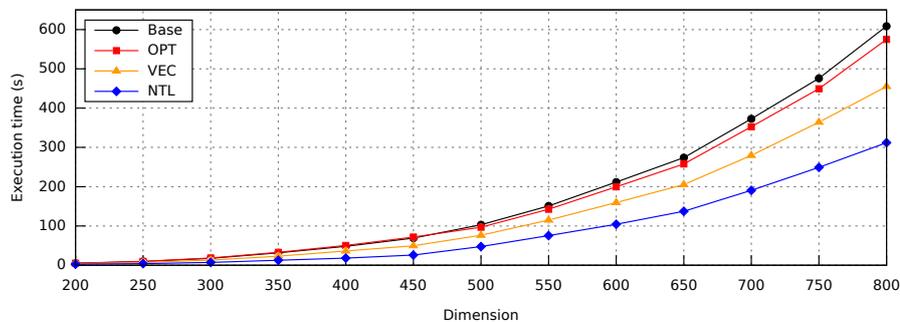


Figure 5.13: Execution time of our LLL implementation and NTL’s LLL implementation, for lattices from the Lattice-Challenge.

In addition, we obtain a speedup of as much as 35% (from which 6% is obtained from the data structures re-arrangement). For the vectorization, we could achieve a theoretical speedup of 36.9%, as the *dot product* loop takes approximately 26.4% of the execution time of the base version (for a lattice in dimension 100), for which we used 256-bit SIMD registers, while the *AddMul* loop takes approximately 60.6%, for which we used 128-bit SIMD registers. The overall speedup of 35% (29%, if the speedup from the re-arrangement is deducted) is thus closer to the maximum possible speedup of 36.9%, which backs up our claim that the vectorization benefit increases with the lattice dimension.

Summary

Although a comprehensive body of work pertaining to LLL has been published in the last decades, there were no studies regarding the vectorization of LLL and the impact of its data structures and kernels on cache locality. With this section, we fill this gap in knowledge. We proposed a re-organization of the data structures in the algorithm, which automatically renders the algorithm more cache friendly and enables the vectorization of two computationally expensive kernels.

We used NTL’s LLL implementation as the reference implementation, for two main reasons. One, the quality of NTL’s implementation is actually identical to that of our implementation, as we showed in the

previous section. Two, the data structures used in NTL are also similar to those of our base version, and so the techniques we propose can also be applied to NTL.

We show that (i) our data structure re-arrangement increases performance with the lattice dimension (ii) vectorizing the dot product and AddMul kernels can achieve as much as 35% speedup on larger lattices and (iii) our implementation is $\approx 10\%$ more efficient than NTL's on lower dimensional lattices (up to dimension 100).

Our data structure re-arrangement benefits cache locality, as elements are stored consecutively in memory, at the cost of additional overhead due to complex index calculation. Benchmarks on low dimensional Goldstein-Mayer lattices showed that performance is improved by the re-arrangement, as the introduced overhead is not outweighed by the gains in cache locality. However, on higher lattice dimensions (for which we used Ajtai lattices), performance is better than without re-arrangement and increases with the dimension. In addition, vectorization gains are also increased, as there are enough operands to amortize vectorization overhead.

5.3 The Block Korkine-Zolotarev (BKZ) algorithm

In 1987, Schnorr presented a new hierarchy of reduction algorithms that generalize LLL, offering a parametrizable trade-off between running time and the quality of the solution [110]. The most outstanding element in this hierarchy is what Schnorr called *Block Korkine-Zolotarev* reduction, today known as BKZ. The algorithm uses a parameter β to define the size of a block, which defines the space over the original basis (or projected lattice) used to run the SVP on. The execution time of the algorithm increases exponentially with β , i.e. the dimension of the lattice on which the SVP is solved, but the quality of the solution also increases.

Starting from a LLL-reduced basis $\mathbf{B} = (\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_n)$, the BKZ algorithm performs iterative *rounds*, i.e. $n - 1$ consecutive sliding SVP calls, in blocks of dimension $\leq \beta$ over the basis, from j to $k = \min(j + \beta - 1, n)$. The solutions of the SVP-solver, are inserted in the basis, in the beginning of each block (or projected lattice L). Thus, at the end of the algorithm, the vector in the beginning of each block $B_{[j,k]}$ is also the shortest vector of the projected lattice $L_{[j,k]}$. Although in theory any SVP-solver can be used in BKZ, enumeration is usually the algorithm used in the literature for this purpose. The SVP-solver finds the solution of the projected lattice $L_{[j,k]}$, yielding a solution \mathbf{s} such that $\|\pi_j(\mathbf{s})\| = \lambda_1(L_{[j,k]})$.

Inserting one of these solutions \mathbf{s} in the basis, as in $\mathbf{B} = (\mathbf{b}_1, \dots, \mathbf{b}_{j-1}, \mathbf{s}, \mathbf{b}_j, \dots, \mathbf{b}_n)$, renders \mathbf{B} invalid. Therefore, at the end of each round, $\mathbf{B} = (\mathbf{b}_1, \dots, \mathbf{b}_n)$ is LLL-reduced and vectors resulting in linear dependences are eliminated. Whenever at least one SVP call in a given round yields a new solution, a new round is executed. The algorithm is described in Algorithm 2.

Despite its importance, no clear-cut upper bound for BKZ's asymptotic complexity has ever been proposed. In practice, BKZ is only practical for certain values of β , say up to 30.

5.3.1 State of the art of BKZ

As mentioned before, the original BKZ algorithm was proposed by Schnorr, in 1987 [110]. A series of publications, building on the original algorithm, followed in the three next decades. The most outstanding contribution is due to Chen et al., who presented BKZ 2.0 in 2011 [20, 21]. BKZ 2.0 is an enhanced version of BKZ, which includes a few improvements, the biggest being the usage of sound (also called

Algorithm 2: BKZ algorithm by Schnorr [110]

1 **Input:** A LLL-reduced basis $\mathbf{B} = (\mathbf{b}_1, \dots, \mathbf{b}_m)$, a blocksize $\beta \in \{2, \dots, n\}$, and the Gram-Schmidt coefficients;

2 $z \leftarrow 0; j \leftarrow 0; \text{LLL}(b_1, \dots, b_m, \mu);$

3 **while** $z < n - 1$ **do**

4 $j \leftarrow (j \bmod (n - 1)) + 1;$

$k \leftarrow \min(j + \beta - 1, n);$

$h \leftarrow \min(k + 1, n);$

5 $\mathbf{v} \leftarrow \text{Enum}(\mu_{[j,k]}, \|\mathbf{b}_j^*\|^2, \dots, \|\mathbf{b}_k^*\|^2);$

6 **if** $\mathbf{v} \in B$ **then**

7 $z \leftarrow z + 1; \text{LLL}(\mathbf{b}_1, \dots, \mathbf{b}_h, \mu);$

8 **else**

9 $z \leftarrow 0; \text{LLL}(\mathbf{b}_1, \dots, \mathbf{v}, \mathbf{b}_j, \dots, \mathbf{b}_h, \mu);$

extreme) pruning (cf. Chapter 4). The main idea is that as pruned enumeration is considerably faster than the default enumeration, BKZ can be run with much higher block-sizes. Bounding functions play a core role in BKZ 2.0, as they define the extent of pruning in the enumeration calls. However, the best bounding functions for BKZ 2.0 remain unclear, as there is no disclosed, publicly available implementation of BKZ 2.0, to our knowledge. Very recently, Aono et al. proposed a variant of BKZ, called progressive BKZ, which starts reducing the basis with a small block-size increasing it gradually over the lifetime of the reduction [7]. According to the authors, the algorithm is approximately 50 times faster than BKZ 2.0 (cf. [7, Figure 13]).

Parallel implementations. The investigation of parallel methods for BKZ has, somewhat surprisingly, been limited to a handful of publications. In 2014, Liu et al. proposed a parallel variant of BKZ, with higher complexity, claiming that the original algorithm is not suitable for parallel computing (cf. Section 5 [67]), a claim that we dispute in this thesis. The only metric Liu et al. used to judge the quality of the output bases is the length of the shortest vector in the basis, which is often smaller than that of the original algorithm. In 2014, Arnreich and Correia proposed AC_BKZ, a parallel variant of BKZ, which runs multiple ENUM calls on different blocks in parallel (instead of a single ENUM call), within a single round [25, 8]. The vectors found by the ENUM calls are inserted in the basis, which is LLL-reduced before the following round. The process is repeated till no ENUM call outputs new vectors. Analyzed with a comprehensive set of criteria, similar to that used in Section 5.2.2, Correia concluded that the output bases are of inferior quality, when compared to the original BKZ algorithm. The implementation scales well only until 4 threads, and stalls for higher thread counts (cf. Table 4.9 in [25]).

5.3.2 Parallel BKZ on shared-memory machines

"For the execution of the voyage to the Indies, I did not make use of intelligence, mathematics or maps."
Christopher Columbus, Portuguese explorer.

It is well known that enumeration is the dominant kernel of BKZ, representing virtually all the

computation of the algorithm with high block-sizes [25, 21]. In most applications of BKZ, including those in cryptanalysis, the higher the block-size the better the quality of the output basis. This is because higher block sizes in BKZ deliver better quality bases, in general. It is known that after a certain block-size, the quality of the bases does not increase any more (or at least significantly), but for moderate dimensions, the lowest block-size to deliver a very good basis is already impractical, as the complexity of BKZ grows exponentially with the block-size. This motivates the need to study parallel BKZ implementations.

The parallelization of BKZ with multiple ENUM calls in parallel is not efficient, as speedups are modest and the quality of the output basis tends to be worse than that of the original algorithm. To devise a parallel implementation of BKZ, we integrated our parallel enumeration kernels presented in Sections 4.2.1 and 4.2.2. First, we integrated the OpenMP-based enumeration that we proposed in Section 4.2.1. The results are shown in Tables 5.1-5.3. Not surprisingly, for most cases, the scalability results of BKZ are similar to the scalability figures of the ENUM implementation itself (although lower, given that enumeration does not represent the entire computation in BKZ), as enumeration already represents a big percentage of the computation in BKZ. We show results for block-sizes 20, 30 and 40. Until a block-size of 30, the scalability of enumeration is reduced. For a block-size of 40, the scalability of enumeration is considerably better, and so is the scalability of BKZ itself. Increasing the block-size β , where scalability should be better, renders BKZ impractical, and so we were not able to test that. Nevertheless, we believe that the scalability of BKZ would be similar to the scalability of ENUM for higher block-sizes.

The speedup results for our parallel BKZ implementation with the demand-driven parallel implementation of ENUM, which was presented in Section 4.2.1, are shown in Tables 5.4-5.6, for block-sizes 20, 30 and 40. Again, the results are similar to the scalability results of the OpenMP-based ENUM implementation itself, in general. For high thread counts e.g. 64 threads, BKZ is consistently better with the demand-driven implementation than with the OpenMP-based one (cf. results for 64 threads, with block-sizes 30 and 40). If BKZ were practical with higher block-sizes, we would expect the scalability of BKZ to match that of the enumeration implementations, therefore attaining linear scalability at some point.

5.3.3 Summary

Enumeration dominates the execution time of BKZ, by far, for high block-sizes e.g. [67, 20, 25]. In fact, for block-sizes higher than 40, it is responsible for virtually all the computation in the algorithm. Attempts to parallelize BKZ running multiple enumeration calls in parallel, thus exploiting parallelism at a coarse-grain level, have shown to deliver (in some cases significantly) lower quality bases and slowed convergence rates [67, 25, 8]. Therefore, the intuitive solution is to integrate a parallel enumeration kernel in BKZ. In this scheme, the scalability of BKZ is closely related to the scalability of the implementation used to enumerate vectors. However, enumeration only scales well for high block-sizes (cf. Section 4.2.1), which render BKZ impractical. Up until block-size 30, BKZ is practical, but, as we have shown in this section, scalability is reduced (as the workload in enumeration is too small so that high scalability of ENUM is attained). For block-size 40, scalability increases considerably. We conjecture that BKZ would attain linear scalability if higher block-sizes were possible, as enumeration would represent virtually all the computation of the algorithm.

N	2T	4T	8T	16T	32T	64T
60	1.00	1.00	0.98	0.87	0.65	0.52
61	1.00	0.99	0.97	0.86	0.62	0.43
62	0.98	1.01	0.99	0.86	0.58	0.43
63	1.00	1.00	0.97	0.84	0.58	0.36
64	1.00	0.99	0.98	0.93	0.78	0.66
65	1.02	1.01	1.00	0.87	0.65	0.42
66	1.00	0.98	0.98	0.90	0.73	0.57
67	1.00	0.99	0.98	0.92	0.76	0.62
68	1.00	0.98	1.00	0.95	0.85	0.71
69	1.00	0.97	1.00	0.92	0.83	0.69
70	0.99	0.98	0.99	0.93	0.81	0.64
71	1.00	1.00	0.99	0.94	0.83	0.68
72	1.00	0.97	0.97	0.90	0.69	0.48
73	1.02	1.02	1.01	0.94	0.82	0.67
74	0.99	0.99	0.99	0.90	0.69	0.50
75	1.00	1.00	0.99	0.92	0.73	0.56
76	1.00	0.99	1.00	0.90	0.77	0.57
77	1.01	1.00	1.00	0.91	0.72	0.56
78	1.00	0.94	0.98	0.88	0.68	0.49
79	1.00	0.99	0.99	0.94	0.81	0.65
80	0.99	0.99	1.00	0.94	0.79	0.63

Table 5.1: Parallel BKZ ($\beta=20$) with task-based parallel enumeration, for 2-64 threads, on Lara.

N	2T	4T	8T	16T	32T	64T
60	1.23	1.40	1.49	1.30	0.46	0.22
61	1.25	1.42	1.49	1.31	0.55	0.25
62	1.31	1.56	1.64	1.48	0.43	0.24
63	1.36	1.59	1.74	1.30	0.43	0.19
64	1.25	1.40	1.49	1.26	0.51	0.25
65	1.30	1.54	1.67	1.31	0.45	0.23
66	1.18	1.31	1.37	1.26	0.59	0.28
67	1.25	1.39	1.48	1.35	0.55	0.24
68	1.34	1.59	1.70	1.38	0.40	0.17
69	1.23	1.38	1.45	1.32	0.51	0.27
70	1.23	1.40	1.47	1.31	0.50	0.23
71	1.25	1.41	1.49	1.33	0.48	0.26
72	1.22	1.37	1.44	1.29	0.49	0.25
73	1.26	1.44	1.52	1.38	0.49	0.26
74	1.24	1.40	1.49	1.36	0.53	0.21
75	1.33	1.56	1.71	1.44	0.42	0.17
76	1.35	1.61	1.73	1.48	0.42	0.17
77	1.33	1.59	1.69	1.48	0.43	0.15
78	1.35	1.61	1.72	1.47	0.43	0.15
79	1.29	1.49	1.60	1.41	0.43	0.25
80	1.32	1.54	1.65	1.43	0.40	0.18

Table 5.2: Parallel BKZ ($\beta=30$) with task-based parallel enumeration, for 2-64 threads, on Lara.

N	2T	4T	8T	16T	32T	64T
60	1.94	3.59	5.98	9.08	6.45	1.46
61	1.84	3.35	5.54	8.40	7.70	1.86
62	1.86	3.56	5.90	9.06	8.17	1.72
63	1.81	3.60	6.11	9.43	8.17	1.81
64	1.87	3.57	6.08	9.51	7.73	1.86
65	1.90	3.62	6.23	9.90	8.19	1.67
66	1.90	3.58	6.04	9.29	8.41	1.99
67	1.89	3.61	6.13	9.69	6.58	1.58
68	1.89	3.52	6.03	9.33	8.54	1.88
69	1.89	3.58	6.06	9.46	9.73	2.66
70	1.89	3.57	6.11	9.43	9.16	1.69
71	1.89	3.63	6.23	9.84	7.18	1.56
72	1.87	3.60	6.12	9.44	7.72	1.48
73	1.81	3.56	6.01	9.37	9.35	1.95
74	1.89	3.51	6.19	9.65	8.14	1.87
75	1.89	3.59	6.16	9.58	8.10	1.86
76	1.88	3.62	6.13	9.64	7.81	1.83
77	1.87	3.52	5.76	8.82	8.94	2.31
78	1.90	3.63	6.18	9.52	8.17	1.87
79	1.88	3.54	6.13	9.70	7.28	1.81
80	1.90	3.65	6.21	9.85	8.15	1.95

Table 5.3: Parallel BKZ ($\beta=40$) with task-based parallel enumeration, for 2-64 threads, on Lara.

N	2T	4T	8T	16T	32T	64T
60	1.03	1.01	0.95	0.86	0.73	0.60
61	1.00	0.99	0.91	0.80	0.64	0.46
62	1.02	0.98	0.90	0.79	0.64	0.47
63	1.02	0.99	0.90	0.79	0.64	0.46
64	1.01	0.99	0.96	0.91	0.83	0.69
65	1.01	0.98	0.90	0.78	0.63	0.45
66	1.01	0.99	0.96	0.85	0.73	0.57
67	1.01	0.99	0.94	0.86	0.76	0.60
68	1.00	1.00	0.97	0.93	0.86	0.76
69	0.99	0.98	0.94	0.89	0.79	0.66
70	1.01	0.98	0.95	0.89	0.79	0.65
71	1.00	0.99	0.95	0.90	0.81	0.67
72	1.00	0.99	0.94	0.82	0.67	0.51
73	1.00	1.00	0.95	0.87	0.77	0.61
74	1.01	0.98	0.91	0.79	0.64	0.47
75	1.02	0.98	0.92	0.83	0.69	0.52
76	1.02	0.99	0.93	0.86	0.72	0.55
77	1.04	1.00	0.93	0.84	0.68	0.51
78	1.02	0.99	0.91	0.80	0.64	0.46
79	1.01	1.00	0.95	0.89	0.79	0.64
80	1.01	1.00	0.94	0.88	0.77	0.61

Table 5.4: Parallel BKZ ($\beta=20$) with demand-driven parallel enumeration, for 2-64 threads, on Lara.

N	2T	4T	8T	16T	32T	64T
60	1.29	1.36	1.30	1.15	0.96	0.74
61	1.30	1.36	1.31	1.17	0.99	0.77
62	1.38	1.46	1.35	1.19	0.97	0.71
63	1.40	1.48	1.36	1.18	0.93	0.68
64	1.31	1.35	1.28	1.14	0.96	0.73
65	1.36	1.45	1.33	1.14	0.89	0.62
66	1.25	1.31	1.27	1.15	0.98	0.77
67	1.30	1.38	1.31	1.15	0.95	0.72
68	1.41	1.50	1.37	1.13	0.87	0.60
69	1.30	1.35	1.29	1.14	0.95	0.72
70	1.30	1.36	1.29	1.14	0.94	0.69
71	1.32	1.38	1.30	1.13	0.91	0.67
72	1.30	1.36	1.27	1.11	0.91	0.65
73	1.34	1.40	1.30	1.12	0.89	0.63
74	1.32	1.39	1.29	1.13	0.91	0.66
75	1.41	1.48	1.32	1.08	0.81	0.54
76	1.23	1.49	1.33	1.08	0.81	0.53
77	1.40	1.48	1.32	1.08	0.81	0.54
78	1.41	1.51	1.32	1.07	0.79	0.54
79	1.37	1.44	1.30	1.09	0.84	0.58
80	1.38	1.40	1.30	0.98	0.80	0.54

Table 5.5: Parallel BKZ ($\beta=30$) with demand-driven parallel enumeration, for 2-64 threads, on Lara.

N	2T	4T	8T	16T	32T	64T
60	1.89	3.45	5.54	7.47	7.55	6.68
61	1.91	3.45	5.58	7.56	7.46	6.71
62	1.92	3.57	5.85	8.01	8.05	7.03
63	1.94	3.56	5.91	7.76	7.83	6.79
64	1.93	3.57	5.88	8.08	8.02	6.46
65	1.93	3.58	6.00	8.22	8.15	6.37
66	1.95	3.59	5.98	8.34	8.62	7.61
67	1.94	3.57	5.91	7.83	7.65	6.44
68	1.93	3.56	5.89	8.13	8.27	7.17
69	1.93	3.57	5.94	8.34	8.57	7.55
70	1.94	3.58	5.94	7.80	7.98	6.65
71	1.95	3.60	5.98	7.60	7.65	5.97
72	1.93	3.56	5.90	7.19	7.42	6.43
73	1.94	3.57	5.90	8.00	7.90	6.96
74	1.94	3.59	5.94	7.56	6.58	6.30
75	1.95	3.59	5.95	7.64	7.40	6.40
76	1.94	3.59	5.93	7.59	7.20	6.01
77	1.93	3.51	5.71	7.35	7.16	6.28
78	1.94	3.60	5.94	7.45	7.18	6.21
79	1.94	3.59	5.89	7.25	6.96	6.07
80	1.94	3.60	5.87	7.46	7.04	6.27

Table 5.6: Parallel BKZ ($\beta=40$) with demand-driven parallel enumeration, for 2-64 threads, on Lara.

Scalable and Efficient Sieving Implementations

Synopsis. This chapter introduces sieving-based SVP-solvers and techniques to efficiently implement and parallelize these algorithms on multi-core CPUs, such that high performance and scalability are attained. In particular, various memory-related enhancements are proposed. The chapter centers on ListSieve, GaussSieve, HashSieve and LDSieve, the core sieving algorithms to date. The techniques proposed in this chapter improve the current existent parallel schemes of GaussSieve, and represent the first parallelization of HashSieve and LDSieve, the most practical among all sieving algorithms. To this day, the HashSieve implementation presented in this chapter holds the record of the highest random lattice dimensions where the SVP was solved, for published algorithms.

"People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.", Donald Knuth.

6.1 Sieving algorithms

History and timeline. Sieving algorithms are randomized, probabilistic algorithms to solve the SVP. The first sieving algorithm for the SVP, named after its authors Ajtai, Kumar and Sivakumar (AKS), dates back to 2001 [5]. The algorithm has $2^{O(n)}$ complexity (both time and space complexity) and creates a big list of vectors, which grows exponentially with the lattice dimension, to which a sieving procedure is applied, in order to find two vectors whose difference is the shortest non-zero vector of the lattice. This is actually the underlying idea of all sieving algorithms that followed. A certain number of random lattice vectors, i.e. vectors that are generated randomly but do belong to the input lattice, are iteratively created and subsequently reduced against one another. Reducing a vector \mathbf{v} against another vector \mathbf{k} means to make \mathbf{v} smaller by subtracting (possibly a multiple of) \mathbf{k} to it. Different sieving algorithms employ different rationales on how (e.g. which order) they perform the reduction of the vectors.

Some iterations in sieving algorithms might result in collisions, i.e. a given vector \mathbf{v} is reduced to the zero-vector (the origin of the lattice). This is a big impediment to prove the completion and complexity bounds of sieving algorithms [83]. As a result, it is common to make theoretical heuristic assumptions in order to arrive at collision bounds, thereby proving correctness [15]. After AKS and Nguyen-Vidick's

Algorithm	Provable	Time Complexity	Space Complexity
AKS [5]	✓	$2^{3.40n+o(n)}$	$2^{1.99n+o(n)}$
Nguyễn-Vidick [92]	✗	$2^{0.415n+o(n)}$	$2^{0.208n+o(n)}$
ListSieve (LS) [83]	✓	$2^{3.20n+o(n)}$	$2^{1.33n+o(n)}$
GaussSieve (GS) [83]	✗	Unknown, see Section 3.3.2	
LS-birthday [101]	✓	$2^{2.47n+o(n)}$	$2^{1.24n+o(n)}$
Two-level sieve [129]	✓	$2^{0.39n+o(n)}$	$2^{0.26n+o(n)}$
Three-level sieve [133]	✓	$2^{0.38n+o(n)}$	$2^{0.28n+o(n)}$
Overlattice sieve [11]	✗	Trade-off, see Section 3.3.2	
HashSieve [59]	✗	Trade-off, see Section 3.3.2	
LDSieve [10]	✗	Trade-off, see Section 3.3.2	

Table 6.1: Sieving algorithms and their time and space asymptotic complexities. n is the lattice dimension.

sieve, many sieving algorithms were proposed, but only some, including ListSieve (cf. Table 6.1), were proved to solve the SVP, as opposed to the others, which are heuristics [83]. GaussSieve, for instance, is a heuristic version of ListSieve [83]. Proposed in 2010, along with ListSieve, GaussSieve achieved higher performance compared to other sieving algorithms and became the first sieving algorithm to surpass enumeration algorithms (although for a brief period of time, as later in the same year extreme pruning was proposed, thereby making enumeration the best SVP-solver).

ListSieve and GaussSieve grow the list progressively, as opposed to starting off with a big list of vectors, as in AKS. ListSieve samples new vectors and reduces them against the vectors in the list, adding then the samples to the list, as shown in Algorithm 3. GaussSieve samples new vectors and not only reduces them against the vectors in the list, it also reduces the vectors in the list against the sample vectors, before adding these to the list, as shown in Algorithm 4. This is the condition of a reduced basis achieved by the Gauss/Lagrange lattice basis reduction algorithm for two dimensional lattices, hence the name of the algorithm [83]. ListSieve’s maximum list size can be bounded in theory (cf. Kabatiansky et al. [37]), which also bounds the running time of the algorithm. GaussSieve’s running time, on the other hand, cannot be bounded in theory.

Random vectors, commonly referred to as *samples*, are typically generated with Klein’s algorithm [54]. Both ListSieve and GaussSieve execute until a certain stopping criterion, $K \geq c$, where K is the

Algorithm 3: ListSieve

1 **Input:** Basis \mathbf{B} , stopping criterion c ;

Init.: $L \leftarrow \{\}$

while $K < c$ **do**

$\mathbf{p} \leftarrow \text{SampleKlein}(\mathbf{B})$;

$\mathbf{v} \leftarrow \text{ListReduce}(\mathbf{p}, L)$;

if $\|\mathbf{v}\|=0$ **then**

$K \leftarrow K + 1$;

else

$L \leftarrow L \cup \{\mathbf{v}\}$;

return $\text{BestVector}(L)$;

function ListReduce(\mathbf{p}, L)

while $\exists \mathbf{v}_i \in L : \|\mathbf{p} - \mathbf{v}_i\| \leq \|\mathbf{p}\|$

$\wedge \|\mathbf{p}\| \geq \|\mathbf{v}_i\|$ **do**

$\mathbf{p} \leftarrow \mathbf{p} - \mathbf{v}_i$;

return \mathbf{p} ;

end function

function BestVector(L)

return $\mathbf{p} : \forall \mathbf{v} \in L, \|\mathbf{p}\| < \|\mathbf{v}\|$;

end function

Algorithm 4: GaussSieve

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

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

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

number of collisions, is met. c is usually set in the form $c = \alpha \times mls + \beta$, where mls is the maximum size of the list L up to that point. When the sieving process finishes, the shortest vector of the lattice is expected to be in L , with a certain, yet high probability. Originally in ListSieve, samples are generated with perturbations, a technique useful to infer the asymptotic complexity of the algorithm (see [83]). The pseudo-codes presented in Algorithm 3 and Algorithm 4, on the other hand, are practical implementations of ListSieve and GaussSieve, where randomly generated vectors are not perturbed. ListSieve samples vectors and reduces them as much as possible against the vectors stored in L , where the freshly reduced vector is inserted once the reduction process is finished. Once in L , vectors are never removed or modified. According to the original description of the algorithms, the reduction process can pick the vectors in L in any order. However, it is known that keeping L sorted by increasing norm is more efficient in practice, since the process can be aborted when a vector bigger than the sample is found e.g. [75].

In GaussSieve, sample vectors are not only reduced against the vectors in the list L as the list vectors are also reduced against the sample vectors. As a result, the elements in L will be pairwise reduced, which means that the inequality $\min(\|\mathbf{p} \pm \mathbf{v}\|) \geq \max(\|\mathbf{p}\|, \|\mathbf{v}\|)$ holds for all $\mathbf{v}, \mathbf{p} \in L$. Another difference between the algorithms is that in GaussSieve the list L can both grow and shrink, as opposed to ListSieve, where the list can only grow. In addition, GaussSieve uses a stack S to store vectors that are removed from L , which happens when the vectors in L are changed, as a consequence of a reduction against a sample vector. The use of the stack eases the handling of the vectors that no longer can be guaranteed to satisfy the aforementioned inequality. This happens because when a vector \mathbf{v} is generated and reduced against an element in L , there might be elements in L that are no longer pairwise reduced with \mathbf{v} . Reverting this is not as simple as reducing such vectors by \mathbf{v} , because it might happen that they become no longer Gauss-reduced with other elements in L . These elements are therefore brought to stack S (and reduced against \mathbf{v}) and picked in the subsequent iteration, as if they were freshly generated samples, thus becoming pairwise reduced with the whole list L .

2010-2015. Between 2010 and 2015, other sieving algorithms were proposed, but mainly with theoretical implications. In 2011, Wang et al. proposed a two level sieving approach [129], which improves on the basic idea of Nguyễn-Vidick's sieve; while the latter uses balls centered in vectors and

makes reductions between the vectors that “fall” within the same balls [92], Wang et al. propose to use dual level balls, or balls with smaller balls inside them, and only try to make reductions between vectors that fall within the same big and small ball at the same time. In 2013, Zhang improved on the result of Wang et al., presenting a three level sieving [133], which essentially extends the two-level sieving by one layer. The natural question would be whether it would make sense to increase the number of layers, but Zhang et al. argued that this would considerably increase the complexity of the run-time analysis of the algorithm, while providing little benefit. Despite the attractive asymptotic complexity of these dual- and three-level sieving algorithms, no implementations are known as these techniques allegedly introduce too much overhead. In 2014, Becker et al. proposed a sieving algorithm based on over lattices, called Overlattice sieve [11], improving upon the run-time complexity of the previous algorithms at the cost of (even more) memory. In practice, this algorithm delivers reasonable performance and lends itself to parallelism [11], but far superior algorithms were published right after.

Advanced (post 2015) sieving algorithms. Last year, Laarhoven proposed HashSieve, an algorithm that improved significantly on the complexity of sieving algorithms [59]. Laarhoven showed that a well-known method from the field of nearest neighbour search, called locality-sensitive hashing, can be used to significantly speed up the search step in sieving. The crucial difference between HashSieve and previous sieving algorithms, such as GaussSieve, is essentially how the selection of vectors for the reduction process is carried out. Instead of going through all the vectors in the system in linear time, as in GaussSieve, HashSieve uses \mathbb{T} independent hash tables $H_1, \dots, H_{\mathbb{T}}$ to look up nearby vectors, which substantially reduces the search space of vectors (Section 6.3.4 shows the practical consequences of this). Given a target vector \mathbf{v} , the algorithm computes the hash value $h_i(\mathbf{v})$ and looks up hash table H_i in the bucket labelled $h_i(\mathbf{v})$. With these locality-sensitive hash functions h_i , vectors mapped to the same bucket have higher probability of being nearby than vectors mapped to different buckets. For more information on the family of hash functions used in HashSieve, the reader is referred to [59]. For each of the \mathbb{T} hash tables, there is a hash function h_i , leading to \mathbb{T} hash tables $H_1, \dots, H_{\mathbb{T}}$ with different hash functions $h_1, \dots, h_{\mathbb{T}}$. As the range of each of these combined hash functions is $\{0, 1\}^{\mathbb{K}}$, the number of buckets in each table is $2^{\mathbb{K}}$, where \mathbb{K} is the number of hash functions from the original family of hash functions, that are combined into each combined hash function h_i [59].

The pseudo-code of the HashSieve algorithm is given in Algorithm 5. The algorithm repeats the following procedure: (i) sample a random lattice vector \mathbf{v} (or get one from the stack S); (ii) find nearby candidate vectors \mathbf{w} to reduce \mathbf{v} with, in the hash tables, by determining a set of candidates \mathcal{C} , which is the set of vectors where every vector \mathbf{w} is such that $h_i(\mathbf{w}) = h_i(\mathbf{v})$; (iii) use the reduced vector \mathbf{v} to reduce other vectors \mathbf{w} in the hash tables (and if such a vector \mathbf{w} is reduced, move it onto the stack); and (iv) add \mathbf{v} to the stack or hash tables. The algorithm aims at building a large set of short, pairwise reduced vectors until two are $\lambda_1(\mathcal{L})$ apart from each other. After that many collisions are generated, and the algorithm terminates when a number c of collisions is reached.

Increasing \mathbb{K} and \mathbb{T} renders the hash functions more and more selective, and will cause them to only map vectors to the same bucket if they are *really close* to one another in space. However, increasing \mathbb{K} and \mathbb{T} comes at the cost of increasing space complexity; to actually be able to find list vectors in this hash table, we need to store all list vectors in each of the hash tables as well. Also, to find candidate reducing vectors in these hash tables, we need to compute the \mathbb{T} hash values of the target vector \mathbf{v} and perform \mathbb{T} hash table look-ups. Thus, at some point, making the number of hash tables \mathbb{T} bigger will not improve the

Algorithm 5: The HashSieve algorithm

```
1 Input: (Reduced) basis  $\mathbf{B}$ , collision threshold  $c$ ;  
2 Initialize stack  $S \leftarrow \{\}$ , collisions  $cl \leftarrow 0$   
3 Initialize  $T$  empty hash tables  $H_1, \dots, H_T$   
  
4 while  $cl < c$  do  
5   Pop vector  $\mathbf{v}$  from  $S$  or sample it if  $|S| = 0$   
6   while  $\exists \mathbf{w} \in H_1[h_1(v)], \dots, H_T[h_T(v)] : \|\mathbf{v} \pm \mathbf{w}\| < \|\mathbf{v}\|$  do  
7     for each Hash table  $H_i, \dots, H_T$  do  
8       Obtain the set of candidates  $C = H_i[h_i(\mathbf{v})]$   
9       for each  $\mathbf{w} \in C$  do  
10        Reduce  $\mathbf{v}$  against  $\mathbf{w}$  and Reduce  $\mathbf{w}$  against  $\mathbf{v}$   
11        if  $\mathbf{w}$  has changed then  
12          Remove  $\mathbf{w}$  from all  $T$  hash tables  $H_i$   
13          if  $\mathbf{w} == 0$  then  $cl++$   
14          else Add  $\mathbf{w}$  to the stack  $S$   
  
15   if  $\mathbf{v} == 0$  then  $cl++$   
16   else Add  $\mathbf{v}$  to all  $T$  hash tables  $H_i$ 
```

time complexity any more. For the hash family considered in [59], it was shown that $\mathcal{K} = 0.2209n + o(n)$ and $\mathcal{T} = 2^{0.1290n+o(n)}$ are asymptotically optimal. Thus, for high dimensions, the choices $\mathcal{K} = \lfloor 0.2209n \rfloor$ and $\mathcal{T} = \lfloor 2^{0.1290n} \rfloor$ (i.e., the leading terms rounded to the nearest integer) seem reasonable. From now on, we will refer to these as *optimal parameters*.

Very recently, Becker et al. proposed another algorithm that was eventually dubbed LDSieve [10]. LDSieve differs from HashSieve in the way the buckets are designed (i.e. the property defining when a vector is to be added to a bucket), and in the method for finding the right buckets. The design of the buckets is quite straightforward: choose a random direction in space (by e.g. sampling a vector \mathbf{c} at random from the unit sphere), and a vector is now added to this bucket if its normalized inner product is larger than some constant α : $\frac{\langle \mathbf{v}, \mathbf{c} \rangle}{\|\mathbf{v}\| \cdot \|\mathbf{c}\|} \geq \alpha$. Intuitively, if both $\frac{\langle \mathbf{v}, \mathbf{c} \rangle}{\|\mathbf{v}\| \cdot \|\mathbf{c}\|}$ and $\frac{\langle \mathbf{w}, \mathbf{c} \rangle}{\|\mathbf{w}\| \cdot \|\mathbf{c}\|}$ are large, then we also expect $\frac{\langle \mathbf{v}, \mathbf{w} \rangle}{\|\mathbf{v}\| \cdot \|\mathbf{w}\|}$ to be large, and so $\mathbf{v} - \mathbf{w}$ is likely to be shorter than \mathbf{v} or \mathbf{w} .

As using completely random, independent buckets (defined by the vectors \mathbf{c}) would lead to a large overhead of finding the buckets that a vector is in (as many inner products have to be performed), the decoding procedure and the way the random vectors \mathbf{c} are chosen is somewhat complicated. Instead of using T random vectors $\mathbf{c}_1, \dots, \mathbf{c}_T$, the authors add some structure to these vectors by choosing a random subcode $S \subset \mathbb{R}^{n/m}$ of size $T^{1/m}$, and defining the code words $C = \{\mathbf{c}_1, \dots, \mathbf{c}_T\}$ as the product code $C = S \times S \times \dots \times S = S^m$. In other words, a code word \mathbf{c}_i is formed by concatenating m code words $\mathbf{c}_{i_1}, \dots, \mathbf{c}_{i_m} \in S$. By choosing m small (i.e. *almost-constant*; say $m = O(\log n)$), it is guaranteed that the product code S^m is almost random as well [10, Theorem 5.1], while taking m super-constant allows us to find the right buckets corresponding to a vector with almost no overhead: if there are t buckets in total and $t_0 \ll t$ buckets contain \mathbf{v} , then the time for finding these buckets is proportional to t_0 rather than to t . This *efficient decodability* is crucial for obtaining an improved performance [10].

Analysis of room for improvement. Up until 2015, it was consistently argued that sieving algorithms were way less efficient than enumeration with (extreme) pruning on random lattices e.g. [43, 62, 109, 11]. In addition, enumeration with pruning has been shown to scale well [29, 48, 58], while only moderate scalability was achieved for GaussSieve [86]. It was however shown that sieving could leverage the

structure of ideal lattices [108, Section 6.1], thus increasing the interest around sieving, which were always algorithms of interest as they have better complexity than enumeration ($2^{\mathcal{O}(n)}$ vs. $2^{\mathcal{O}(n \log n)}$), and so they are expected to be better for a sufficiently large lattice dimension n . The fact that sieving algorithms can exploit the additional structure of ideal lattices ignited additional interest because of their use in actual cryptosystems, due to reasons such as reduced key size [98, 68, 123]. Therefore, some relevant questions can be posed:

- Can sieving algorithms be improved?
- Can sieving algorithms be implemented in a way that high scalability is achieved?
- Why are sieving algorithms so much less practical than enumeration if they have lower complexity?

The remainder of this chapter addresses these questions. In Section 6.2, we present heuristics to improve sieving algorithms, by determining with high probability whether vectors are pairwise reduced without computing the complete inner product of vectors. Various other improvements are presented throughout the chapter, along with parallel variants of sieving algorithms. We also positively answer the question of whether sieving algorithms can be implemented in a way that it achieves high scalability, by presenting scalable implementations of ListSieve, GaussSieve, HashSieve and LDSieve in the following sections. This is achieved with techniques that explore the mathematical properties of the algorithms. In Section 6.4, we cover a very important problem, orthogonal to all sieving algorithms, which pertains to memory usage. We show that sieving algorithms are less practical than enumeration algorithms primarily because sieving algorithms are memory-bound, and memory is a bottleneck resource in modern computing architectures.

Contributions. The main contribution of the work conducted in this thesis is to revert the belief that sieving algorithms are not scalable, by presenting scalable versions of ListSieve, GaussSieve, HashSieve and LDSieve. In particular, the techniques here presented leverage the characteristics of sieving algorithms, including the possibility of missed reductions with negligible harm in the convergence rate of the algorithms and no harm at all in convergence. The implementation described in Section 6.3.4, together with critical improvements shown in Section 6.4, holds the record of the highest random lattice dimension where the SVP was solved, for published sieving algorithms. Other contributions, including heuristics to accelerate sieving algorithms in practice, by reducing the number of reductions by looking at vectors superficially, are described in Section 6.2.

6.2 Heuristics to speed up sieving algorithms

Many of the ideas presented in this Section have been published in [35].

Vector reduction is the core routine of sieving algorithms. Two vectors \mathbf{v} and \mathbf{p} are pairwise reduced if $\min(\|\mathbf{v} \pm \mathbf{p}\|) \geq \max(\|\mathbf{v}\|, \|\mathbf{p}\|)$. One can also determine whether \mathbf{v} and \mathbf{p} are Gauss-reduced by considering the angle θ between them. It follows from elementary Euclidean geometry that if $|\frac{\pi}{2} - \theta| \leq \arcsin(r/2r')$, \mathbf{v} and \mathbf{p} are Gauss-reduced. In ListSieve and GaussSieve, the overwhelming majority of vector pairs that are considered for reduction, are already Gauss-reduced. This means that attempts to reduce two different vectors fail, and the computation incurred (mostly the inner product) is wasted. In practice, to determine whether two vectors \mathbf{v} and \mathbf{p} are Gauss-reduced, we evaluate

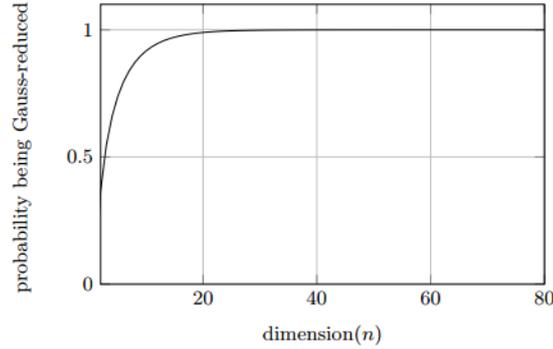


Figure 6.1: Example probabilities of a priori Gauss-reduction [35].

$2 \times |\langle \mathbf{v}, \mathbf{p} \rangle| > \|\mathbf{v}\|$ (recall the “Common concepts” in Section 2.1.1), and if this holds, the reduction will be successful. Reductions account for a significant portion of the execution time of these algorithms (e.g. [51, 75] report overall speedups of 2x when vectorizing the reduction kernel only), yet only a small part of this is in fact useful computation. It would thus be beneficial to determine (possibly with a small margin of error) whether a pair of vectors is already Gauss-reduced or not, using less computation than the inner product between the vectors. This can be achieved with an *approximate Gauss-reduction*, which we presented in [35].

The idea behind an *approximate Gauss-reduction* is that as long as it is possible to estimate θ efficiently (rather than actually computing it entirely), it is possible to say, with a good degree of confidence whether \mathbf{v} and \mathbf{p} are indeed Gauss-reduced. This can be achieved by exploiting correlations between the inner product of the sign bits of a pair of vectors and the angle between them. This stems from the following assumption: for two vectors \mathbf{v} and \mathbf{p} with “similar” norms, sampled uniformly at random from the set of all vectors of a given lattice, the distribution of the normalized sign inner product of these vectors and the angle between them can be approximated by a bivariate Gaussian distribution [35]. Put simply, based on this assumption, one can use the sign inner product of \mathbf{v} and \mathbf{p} as a first approximation to the angle θ between them, when their norms are relatively similar.

The motivation behind using this method as an approximation to the angle between two vectors is that (i) most vectors are likely to be already Gauss-reduced, which is shown in Figure 6.2 and (ii) computing an inner product of binary numbers can be implemented very efficiently on modern computer architectures. For instance, to calculate the (normalized) sign inner product of two vectors, one can load the sign bits of each vector to SIMD registers, take the bitwise XOR of these registers and perform a population count (and divide the result by n). The result yielded by this operation will tell us how many coordinates of both vectors, for the same positions, differ in sign. In high dimensions, when sampling uniformly at random, we expect the number of such individual XORs to be $n/2$. We can then determine how many vectors deviate from this and consider that if they do not deviate from $n/2$ by more than k coordinates, then they are pairwise reduced and we do not compute their inner product to try to reduce them. It is important to stress that although this heuristic works well in practice, it is purely guided by intuition. As shown in [35], k is to be chosen empirically, with values of 6 or 7 appearing to work best for the lattice dimensions experimented. If k is too small or too big, too many false negatives crop up, thereby leading to a decreased speed up. On the benchmarks we carried out in [35], we obtained speedups of about 3x using this optimization alone.

6.3 Parallelization techniques for shared-memory machines

Although ListSieve, GaussSieve, HashSieve and LDSieve share core rationales, their workflow is considerably different and so they have to be parallelized in different ways. This section introduces techniques to parallelize these algorithms, some of which leverage the fact that reductions may be missed occasionally. These techniques include lock-free and probable lock-free data structures which scale well on shared-memory systems. In particular, they achieve almost linear, linear and super-linear scalability depending upon the context.

6.3.1 Relevant sieving implementations

In 2011, Milde and Schneider studied the parallelization of GaussSieve, and proposed a parallel implementation with limited scalability [86]. We refer to this implementation as *Milde2011* from now on. In brief, they implement a ring structure where each thread has a local list and executes the entire GaussSieve algorithm, but hands out the sample vectors to the next thread to process (samples are stored in a local list once the entire ring is travelled). The ring structure, which also includes queues to buffer the flow of vectors across the different threads, is shown in Figure 6.2. The problem is that some vectors in the ring structure will never encounter one another, and many reductions are missed, thus increasing the necessary iterations for convergence, and lowering scalability.

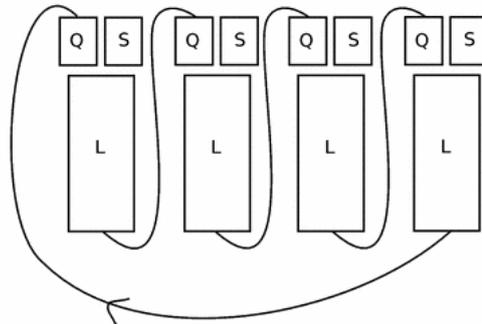


Figure 6.2: Ring structure implemented in *Milde2011*, taken from [86].

Milde2011 relaxes the properties of GaussSieve in the sense that several pairs of vectors might never be pairwise-reduced during the execution of the algorithm. Let us consider a scenario with 2 threads, where a given vector \mathbf{v} and a given vector \mathbf{p} are released, at the same time, by threads 1 and 2, respectively. If the vector \mathbf{p} is reduced against the vectors in the local list of thread 1 before \mathbf{v} is in that list (\mathbf{v} is for example travelling the ring), \mathbf{p} and \mathbf{v} will never be reduced against each other (*missed reduction*). \mathbf{v} and \mathbf{p} will eventually be added to the local lists of threads 1 and 2, respectively. Each vector will possibly fluctuate between the local list of the thread that released it and the private stack of that same thread, but \mathbf{v} and \mathbf{p} will never be reduced against each other.

In 2014, Ishiguro et al. presented a distributed-memory implementation of GaussSieve which we refer to as *Ishiguro2014* [51]. The authors claim that their implementation scales reasonably well with the number of processes, and it has successfully broken a 128-dimensional ideal lattice, as their implementation can also take advantage of ideal lattices. The list of vectors is not split among the processes, and each process has access to a global list. The implementation generates samples in batches

(of a parameterizable size r) and synchronizes the processes between iterations. Unfortunately, there are some important further details pertaining to synchronization that are not disclosed in the paper, thereby making it impossible to reimplement their version. The source code is not disclosed either.

In 2014, Bos et al. proposed another implementation of GaussSieve, which integrates a new approach to take advantage of ideal lattices [15]. The parallelization consists in passing the same vector to different processes, which hold chunks of one big list, thereby splitting the list among different processes and making sure that the same vector passes through every process. The implementation chops up the list and distributes it among different computing units. Vectors are generated in batches and reduced with respect to the individual lists of each process, based on their sizes. Those that “survive” are pairwise reduced and therefore added to the list of one process. Those that do not, are reduced and moved to a different list (where eventually only the shorter - or minimal representative - of each vector is kept), which is concatenated with a global stack and used to feed the algorithm in the next iteration. This implementation uses synchronization at various steps, including broadcasting samples, agreeing on minimal representatives, and concatenating lists.

6.3.2 Loss-tolerant linked lists for ListSieve

Many of the ideas presented in this Section have been published in [75].

In [86], Milde and Schneider suggested that the parallelization of ListSieve would be of interest, as GaussSieve did not scale well, posing the question of whether a scalable parallel implementation of ListSieve would surpass GaussSieve. We answered this question in [75], by showing that, admitting some scenarios, including the loss of some vectors, ListSieve can indeed scale better than the GaussSieve implementation proposed by Milde and Schneider.

Both ListSieve and GaussSieve make use of a linked list of vectors. However, ListSieve only accesses the list once per sample, as it only reduces the samples against the vectors in the list (and not the vectors in the list against sample vectors, as GaussSieve does). This key difference eases the parallelization of ListSieve, as this lowers the number of possible data races if many threads are used to work on the list. Intuitively, if there was a way to add elements to the list in parallel, ListSieve could be parallelized out right, as each thread would reduce its own samples and add them to a global list, in parallel. In fact, one can have multiple threads adding elements to a shared linked list, and as long as they write in different positions. However, in ListSieve, this is not possible as the list L is sorted by increasing norm, and therefore the elements may be added to the same list location. To fully ensure that no data races arise, one can either use locks or assume that elements may be lost: if two threads try to add an element to a given position k in the list at the same time, only one element (the last) will subsist.

Parallelizing ListSieve with the latter approach, i.e. admitting loss of vectors, comes at two different costs: (i) thread j may miss vectors during its reduction process (as it happens in *Milde2011*), as other threads may insert vectors in a part of the list that thread j has already passed by and (ii) vectors are lost for good when two threads try to add samples in the same position of the list (as only one vector subsists, in this case). At first glance, it may look that this version may suffer from similar problems as *Milde2011*. However, there are two very important differences between the algorithms that should be accounted for: (i) in ListSieve, there are more unsuccessful reductions than in GaussSieve, and (ii) in ListSieve the list is bigger than in GaussSieve, and in particular the list size is much bigger than the number of threads (the difference grows with the lattice dimension), and so the probability of adding vectors to the same position

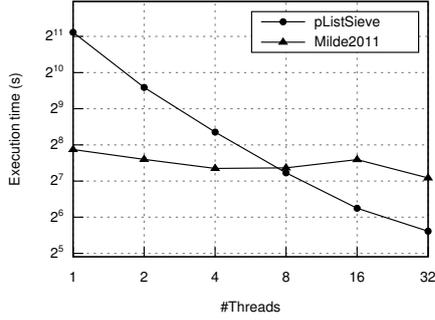


Figure 6.3: Comparison of our ListSieve implementation (*pListSieve*) against *Milde2011* for a BKZ-reduced lattice with blocksize 20 in dimension 60, on Peacock.

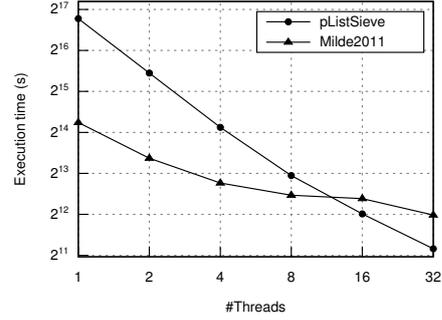


Figure 6.4: Comparison of our ListSieve implementation (*pListSieve*) against *Milde2011* for a BKZ-reduced lattice with blocksize 20 in dimension 70, on Peacock.

- and thus losing them - is very small. With such a loss-tolerant list, ListSieve becomes an heuristic, i.e. it relaxes the original properties of the algorithm. We showed such an implementation in [75]. The list L was implemented as a singly linked list, where each element points to its successor. Each thread executes the workflow of the original algorithm: they sample a vector \mathbf{p} , reduce it against \mathbf{v} , $\forall \mathbf{v} \in L$, thereby generating \mathbf{p}' , and insert \mathbf{p}' in the list L . Threads insert elements between two given vectors \mathbf{v} and \mathbf{w} in the list L , by setting \mathbf{p}' 's *next* pointer pointing to \mathbf{w} and then setting \mathbf{v} 's *next* pointer pointing to \mathbf{p}' .

Figures 6.3 and 6.4 compare this implementation to that of Milde et al., showing that it scales linearly (super-linearly on some instances) and surpasses *Milde2011* for more than 8 threads. The stopping criterion of the implementation was set with $\alpha = 0.1$ and $\beta = 200$. This set of benchmarks was run on Peacock (cf. Table 2.1). The code was compiled with `g++ 4.6.1` (since NTL, used for BKZ, is written in C++) with the optimization flag `-O2`, which turned out to be slightly better than `-O3`. For these tests, we used Goldstein-Mayer lattices, reduced with BKZ (with blocksize 20). Every experiment was repeated three times and the best sample was chosen. The elapsed time of lattice reduction is not included and target norms were never used (a target norm causes the algorithm to stop as soon as a vector with norm smaller or equal to it is found). The output of this implementation was always the shortest vector. This is not completely surprising, as (1) missing reductions do not seem problematic because reductions will at most be missed in a specific iteration (and the reduction could actually be unsuccessful in first place) and (2) losing vectors is very unlikely to happen, due to the length of the list (and thus the low probability of inserting two vectors in the same/continuous positions), and the fact that threads are not likely to insert vectors in L at the same time.

Summary

This section introduced a scalable implementation of ListSieve, which admits loss of vectors and missed reductions. The core idea of the proposed implementation is a linked list that can be used by many threads, but only one thread wins the race if two threads want to insert vectors in the same positions of the list. However, we argue and prove that losing vectors and occasional missed reductions are not problematic in this specific case. Our implementation scales considerably better than *Milde2011*, attaining super-linear speedups (as the reduction process becomes lighter for larger numbers of threads) and surpassing *Milde2011* in execution time for 8 threads onwards, thus answering positively to the question posed in

Milde2011, of whether a scalable version of ListSieve could overperform GaussSieve.

6.3.3 Lock-free linked lists and optimizations for GaussSieve

Many of the ideas presented in this Section have been published in [75].

Although we have shown that ListSieve can scale better than *Milde2011*, GaussSieve is still way more efficient than ListSieve with a single thread (see Figures 6.3 and 6.4). Therefore, is it natural to wonder whether a scalable implementation of GaussSieve can be devised, a challenge we took on in [75].

As shown in the previous section, ListSieve could be trivially parallelized either if either threads could write and read concurrently on a list, or we assume that elements may be lost. GaussSieve is more complex in that regard, as vectors that are successfully reduced against sample vectors are constantly removed from the list, which introduces a second level of concurrency. The question now comes down to whether there is a way to concurrently write (both add and remove) and read from a shared linked list. In 2001, Harris proposed a lock-free linked list that achieves this [45], which we have used as an essential building block of our GaussSieve implementation.

Our implementation replicates the GaussSieve kernel among many different threads, similarly to *Milde2011*, which share a global lock-free Harris list. Each thread executes the original workflow of the algorithm: they sample a vector \mathbf{v} , reduce it against every vector \mathbf{p} in L , obtaining \mathbf{v}' , and reduce every vector \mathbf{p} in L against \mathbf{v}' . Each thread has also a private stack S , where $\mathbf{p}' = \text{Reduce}(\mathbf{p}, \mathbf{v}')$ is moved onto, whenever $\mathbf{p}' \neq \mathbf{p}$. In contrast to *Milde2011*, we keep the vectors in a single, central list L . Therefore, vectors are likely to encounter one another during the reduction process, because they are physically close. In particular, not only are two vectors \mathbf{v} and \mathbf{p} likely to encounter each other during the reduction process, but reduced versions of these vectors are also likely to encounter one another later on. In addition, this approach is better than *Ishiguro2014* because (1) it does not need extra parameters for which the optimal values are not known upfront and (2) it stops as soon as the threshold of collisions is reached.

Enhanced lock-free list

We implemented the Harris lock-free linked list described in [45], with some modifications and extensions that are tailored towards GaussSieve. Each node in the list represents a vector, and includes an array `data[n]`, which represents the coordinates of the vectors, a `long norm`, which holds the norm of the vectors, and a pointer `Node *next` to the next element in the list. n is the dimension of the lattice. We note that this is also the representation we used for the ListSieve implementation presented before (although stored in a common, i.e. not lock-free linked list), and the sieving implementations presented in the next sections. The list is ordered by increasing norm, similarly to `key` in [45]. For the atomics, we used compiler built-in functions. For additional detail, the reader is referred to [75].

The vectors in the list cannot be concurrently modified by different threads. If a thread is to reduce a vector, it first removes the vector from the list, reduces it and adds its reduced version to the list afterwards. To this end, we split the *Reduce* kernel into two kernels: one that tests whether the reduction is successful and another that actually reduces the vector (the latter is only called if the first yields true). This avoids removing vectors that are not reducible.

Both the `insert` and `remove` methods in Harris' linked list use an internal search method, which searches for a given key (a vector norm in our case) from the beginning of the list. However, in the

lock-free GaussSieve implementation, we do not need to search for the desired norm from the beginning of L . If during the traversal, a given vector \mathbf{p} is to be removed, its location is known (and the same happens for the insertion of samples). We extended Harris' linked list to support insertions and removals without traversing the list from scratch, but by providing a pointer to the previous element in the list. However, it is important to note that the known locations cannot be used blindly, since other threads may change the list concurrently in the meantime.

Therefore, we implemented methods to insert and remove elements after a given pointer. These methods receive an extra parameter, `searchPointer`, which is ideally (note that due to concurrent insertions/removals this may not always hold true) the designated predecessor of the new node (for an insert) or the predecessor of the node to be removed (for a remove). These methods are similar to the original insert and remove methods, but call a modified version of the search method, which begins the search from the given pointer, instead of from the beginning of the list. If the list is changed in the meantime, we use the original search method. For additional detail on the extensions we provided, the reader is referred to [75].

Relaxation of GaussSieve properties

Similarly to *Milde2011*, we relax the properties of GaussSieve, although to a much smaller degree. In our implementation, it is possible that a given vector \mathbf{p} is reduced against the elements in the lock-free list L , while another vector \mathbf{v} is already in the system but not in the list L . For instance, \mathbf{v} may lie on the private stack S or under the reduction stage of another thread. If this occurs, \mathbf{p} will not be reduced against \mathbf{v} , but it is possible that \mathbf{v} is reduced against \mathbf{p} . This will happen if \mathbf{p} remains unchanged and is still in L when \mathbf{v} is later on reduced against all the elements in L . In fact, this is likely to happen, because if \mathbf{v} lies on the stack of one thread, it means that it will soon be reduced against all the elements in L . Assuming this scenario, where \mathbf{v} changes to \mathbf{v}' , it is also possible that a reduced version of \mathbf{p} , \mathbf{p}' , is later on reduced against \mathbf{v}' , or vice versa. In fact, this is very likely to happen, because every vector fluctuates between the private stack S of each thread and the lock-free list L . When the element is picked from the private stack of one thread, it is reduced against all the elements in L . In a nutshell, while it is possible that a vector \mathbf{p} is not reduced against another vector \mathbf{v} when it should be, it is likely that reduced versions of these vectors are eventually reduced against one another, unlike *Milde2011*.

The impact of this relaxation on the convergence rate is minimal, otherwise the scalability of our parallel version would be considerably affected, as in *Milde2011*. As the number of missed reductions grows with the number of threads in our approach, it may happen that a very conservative stopping criterion has to be used for a large number of threads. However, the output of our implementation was, for all our experiments (up to 64 threads), identical to the sequential version. Our experiments show that linear speedups are achieved in several scenarios. Carrying out an analysis on the likelihood of missed reductions is a very complicated task, as they depend strongly on many variables, such as the lattice basis itself.

Code optimizations

The dominant kernel of the implementation is the calculation of the dot product $\langle \mathbf{p}, \mathbf{v} \rangle$, which is used to determine if a vector \mathbf{p} should be reduced against a vector \mathbf{v} . We vectorized this kernel for vectors with both *integer* and *short* entries, using 128-bit registers from SSE 4.2 (4 integers or 8 shorts are packed per

register). While *integer* entries did not result in overflow during our experiments, *short* entries did. To overcome this, we used the instruction `PMADDWD`, which multiplies point-wise *short* entries, producing temporary signed, doubleword results. The adjacent doubleword results are then summed up and stored in the destination operand, thus keeping overflow losses. We show performance results pertaining to the vectorization of this kernel in the subsection *Performance of parallel lock-free GaussSieve*.

Another relevant optimization that improved our implementation by up to 15%, is to reduce the number of (dynamic) memory allocations and deallocations of vectors. As we developed our own module for stack S , we save one memory allocation when removing a vector from a list and inserting it on the stack S . In essence, we keep the already allocated vector and we update the stack pointers to point to this memory location.

Algorithmic optimizations

Similarly to enumeration algorithms, which have been optimized with techniques such as extreme pruning, sieving algorithms can also be modified to converge faster. We observed that GaussSieve converges in fewer iterations when: (1) the samples used during the sieving process are short and (2) the reduction of the samples is primarily done against vectors that are short themselves. From here on, these cases will be referred to as *opt1* and *opt2*, respectively. In order to attain *opt1*, we changed parameter d , in Klein’s algorithm, to $\log(n)/70$, thus forcing it to sample shorter vectors. This was addressed in [51, Section 5.4] first hand. It is known that the shorter the samples in sieving algorithms, the faster the algorithms converges. Although our experiments confirmed the performance gains reported in [51], we noticed that, with this modification, the sampler can become a very heavy or even the dominant kernel within the algorithm. Moreover, with this optimization, the default stopping criterion becomes insufficient for lattices in dimensions up to 60, a problem that was not addressed in [51].

opt2 can be achieved by ordering the reduction of sampled vectors differently. According to the description of the algorithm in [83] the reduction of a sampled vector abides by the following condition:

$$\text{while } (\exists \mathbf{v}_i \in L : \|\mathbf{v}_i\| \leq \|\mathbf{p}\| \wedge \|\mathbf{p} - \mathbf{v}_i\| \leq \|\mathbf{p}\|) \text{ do } \mathbf{p} \leftarrow \mathbf{p} - \mathbf{v}_i. \quad (6.1)$$

In the *gsieve*¹ library, the possible reduction of a sample \mathbf{p} is tested against every element in L , and the process restarts from the beginning of the list only (1) after testing the reduction of \mathbf{p} against every element in L and (2) if at least one reduction is successful. Our implementation, on the other hand, restarts the process from the beginning of the list whenever a reduction is successful, therefore forcing the algorithm to use the shorter vectors in L in the first place. Despite of this difference, both implementations abide by Equation 6.1, but our reduction process is faster.

Performance of parallel lock-free GaussSieve

The analysis was carried out with several random lattices, generated with Goldstein-Mayer bases, in multiple dimensions, available on the SVP-Challenge² website. All lattices were generated with seed 0. We used both Peacock and Lichtenberg for these benchmarks (cf. Table 2.1), so that different core counts could be tested. Except for the scalability tests, which were performed on both Peacock and Lichtenberg,

¹<https://cseweb.ucsd.edu/~pvoulgar/impl.html>

²<http://www.latticechallenge.org/svp-challenge/>

	icpc 13.1.3		g++ 4.6.1	
	Time (s)	CPEs	Time (s)	CPEs
Not hand-vectorized				
integers	9.618	3.126	6.900	2.242
shorts	7.012	2.279	7.000	2.274
Hand-vectorized				
integers	0.698	0.227	1.910	0.621
shorts	0.364	0.118	0.982	0.320

Table 6.2: Runtime, in seconds, and Cycles Per Element (CPE) of the dot product kernel, compiled with both `icpc 13.1.3` and `g++ 4.6.1`, for 100 million runs. Memory is 8-bytes aligned.

the results were obtained on Peacock. Peacock runs Ubuntu 11.10, kernel 3.0.0-32-generic, whereas Lichtenberg runs SUSE Linux Enterprise Server 11 SP3, kernel 3.0.101-0.29-default.

For the experiments regarding vectorization and compiler’s impact, we used both Intel `icpc 13.1.3` and GNU `g++ 4.6.1`. For the remaining tests, we used Intel `icpc 13.1.3` with the `-O2` optimization flag on both compilers, since it was slightly better than `-O3`. Every experiment was repeated three times and the best sample was chosen, although the runtimes usually were quite stable among different runs. The elapsed time of lattice reduction is not included in the results.

Vectorization and compiler’s impact. This section shows a quantitative performance evaluation of the vectorization of the kernel that computes the dot product $\langle \mathbf{v}, \mathbf{p} \rangle$, the dominant kernel of the proposed implementation, using SSE 4.2. The kernel was isolated and ran on synthetic vectors, in dimension 80, both 8- and 16-bytes aligned. In order to obtain solid numbers, the kernel was run 100 million times and the average performance was calculated.

Table 6.2 presents the results of the benchmarks when the `data` array is 8-byte aligned. It includes the number of Cycles Per Element (CPE), where an element is a multiplication of \mathbf{v}_i and \mathbf{p}_i , in the dot product $\langle \mathbf{v}, \mathbf{p} \rangle$. The results differ considerably for different data types and between hand- and compiler-vectorized code. Both compilers perform identically on code that is not hand-vectorized for `short` arrays, whereas `g++ 4.6.1` performs better than `icpc 13.1.3` for not hand-vectorized code on `int` arrays, by a factor of $\approx 1.39x$. This picture changes for hand-vectorized code: `icpc 13.1.3` performs better than `g++ 4.6.1` for `integer` arrays, by a factor of $\approx 2.74x$, and for `short` arrays, by a factor of $\approx 2.70x$.

For memory that is 16-byte aligned, the difference between the performance of both compilers is very similar to the results with 8-byte aligned memory. With integers, the performance of all scenarios and both compilers is actually, for two decimal places, the same as with memory that is 8-byte aligned. When it comes to `short` arrays, `icpc 13.1.3` performs worse in code that is not hand-vectorized, but maintains the very same levels of performance in hand-vectorized code. `gcc 4.6.1`, on the other hand, performs worse in both hand and not hand-vectorized code. Based on these results, the experiments that follow were performed with `icpc 13.1.3`, except when said otherwise, and 8-byte aligned data.

Scalability. The scalability of our implementation on Peacock was measured for random lattices in dimensions 60, 70 and 80. Lower dimensions are either solved very quickly or the lattice reduction process finds the shortest vector per se, rendering a scalability analysis worthless. Running the implementation in higher dimensions, on the other hand, is impractical for a single thread.

We conducted two sets of experiments. In the first set, only `opt1` was activated. In the second set, both `opt1` and `opt2` were activated. BKZ ran with block-size 20 for dimensions 60 and 70, and with

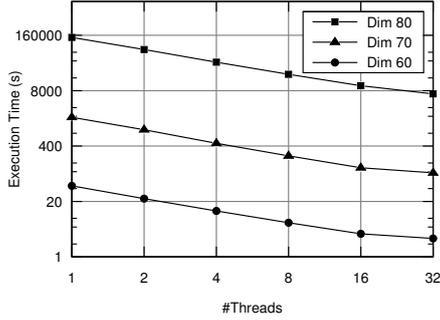


Figure 6.5: Scalability of our implementation on Peacock (with SMT) for 1-32 threads. Results for lattices in dimensions 60, 70 and 80. BKZ’s block-size 32. *opt1* is turned off.

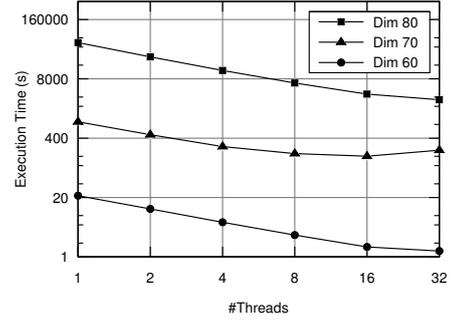


Figure 6.6: Scalability of our implementation on Peacock (with SMT) for 1-32 threads. Results for lattices in dimensions 60, 70 and 80. BKZ’s block-size 32. *opt1* and *opt2* turned on.

block-size 32 for dimension 80. Running the implementation on lattices in dimensions 60 and 70 with the same block-size of 32 is particularly fast, and no significant conclusions can be drawn about the results. Moreover, the parameter d , in dimension 60, was $\log(n)/30$, because the default stopping criterion is insufficient if d is $\log(n)/70$ instead. The `data` array, which holds the coordinates of the vectors, was set to hold `shorts` in both sets of experiments.

Figure 6.5 shows the execution time for the first set of experiments, for 1-32 threads. The application scales linearly for up to 16 threads. The speedup for 16 threads, in both dimensions 60 and 80, is almost linear. The use of two sockets (which involves the use of interconnecting CPU buses) does not seem to impair the scalability of our implementation, since it scales linearly for a lattice in dimension 70. The scalability is limited for the lattice in dimension 60, probably due to the small workload that is entailed. For the lattice in dimension 80, it is possible that the scalability is hurt by the relaxation of the GaussSieve properties on this particular lattice. As shown in Table 6.3, efficiency levels are very high for the three cases, varying between 83.50% and 102.25%. In fact, superlinear speedups are achieved for dimension 70 in 2 cases. Moreover, the implementation benefits from SMT (rows are highlighted in light gray in Table 6.3), since a considerable part of the workflow is memory-bound.

Threads	Dimension 60		Dimension 70		Dimension 80	
	S	E	S	E	S	E
First set of trials (<i>opt1</i> on, <i>opt2</i> off)						
2	2.00x	100%	1.96x	98.00%	1.92x	96.00%
4	3.88x	97.00%	4.09x	102.25%	3.82x	95.50%
8	7.35x	91.88%	8.06x	100.75%	7.35x	91.88%
16	13.36x	83.50%	15.36x	96.00%	13.58x	84.88%
32	17.18x	53.69%	20.25x	63.28%	21.20x	66.25%
Second set of trials (<i>opt1</i> and <i>opt2</i> on)						
2	1.83x	91.85%	1.91x	95.50%	1.93x	96.50%
4	3.84x	96.00%	3.48x	87.00%	3.83x	95.75%
8	7.34x	91.75%	4.97x	62.13%	7.22x	90.25%
16	13.32x	83.25%	5.66x	35.38%	12.64x	79.00%
32	16.41x	51.29%	4.20x	13.13%	16.82x	52.56%

Table 6.3: Speedup (S) and Efficiency (E) of our implementation running on three random lattices (dimensions 60, 70 and 80). BKZ’s block-size was set to 32. SMT is used in grayed out rows.

Threads	Dimension 76		Dimension 78	
	S	E	S	E
8	7.08x	89%	7.74x	97%
16	14.84x	93%	14.86x	93%
32	32.02x	100%	29.65	93%
64	63.64x	99%	53.96x	84%

Table 6.4: Speedup (S) and Efficiency (E) of our implementation on Lichtenberg, a 64-core machine. BKZ’s block-size is 32. *opt1* is turned off and *opt2* is turned on.

Figure 6.6 shows the results for the second set of experiments. In dimension 60, linear speedups are achieved for up to 8 threads and almost linear speedups are achieved for 16 threads. Dimension 70 unveiled a problem that might occur depending on the parameterization of the sampler. *opt1* forces Klein’s kernel to output shorter vectors, which renders the kernel heavier and less scalable. The scalability of this kernel is hurt by the use of, among others, a `rand()`-alike function. As this becomes the dominant kernel with this optimization, the scalability of the whole implementation is reduced. In fact, this is the only case where our implementation does not benefit from SMT. This problem is mitigated for higher dimensions, where the sampler is no longer the dominant kernel, as proven by the results in dimension 80. It is unclear if higher dimensions might benefit from even more strict parameters in Klein’s algorithm, which might speed up GaussSieve but shift the computation weight to the Klein’s algorithm. Either way, we emphasize that (1) a more scalable and efficient kernel of Klein’s algorithm must be developed (a problem which we revisited in [70]) and (2) the proposed implementation of the GaussSieve kernel can be seen as a highly efficient and scalable building block in future implementations.

We also tested our implementation on Lichtenberg (cf. Table 2.1), for 8, 16, 32 and 64 threads. This corresponds to the use of one, two, four and eight CPU-chips, respectively. As we are primarily interested in the scalability of our GaussSieve kernel, *opt1* was deactivated in these tests. The version of `icpc` on this machine is 14.0.2 and the code was also compiled with `-O2`. The speedups and efficiency are shown in Table 6.4. Our implementation scales linearly for a lattice in dimension 76 and almost linearly for a lattice in dimension 78. The running times are considerably slower than on Peacock due to the differences in the microarchitectures.

Comparison with *Milde2011*

We ran both implementations with 1-32 threads on a random lattice in dimension 70. Solving lattices in higher dimensions is impractical for less than 32 threads. In this comparison, the lattice was BKZ-reduced, with block-size 32, and we deactivated *opt1* in our implementation, since it degrades the scalability of the GaussReduce kernel, as mentioned in the previous section.

As shown in [76, Figure 6], not only the single-core performance of our implementation is faster than *Milde2011*, by a factor bigger than 10x, but it also scales much better. In particular, our implementation achieves efficiency levels of 98%, 102.25%, 100.75%, 96% and 63.28% (the latter with SMT), whereas *Milde2011* achieves only 92%, 69.56%, 42.75%, 22.62% and 14.92% (the latter with SMT) for 2, 4, 8, 16 and 32 threads, respectively.

Comparison with *Ishiguro2014*

The *Ishiguro2014* implementation is not disclosed, and several implementation details are omitted, which makes it impossible to re-implement their approach. Nevertheless, we can still compare our results with the execution times reported in the paper, since we use the same CPU-chip model. We also replicated the test environment: we ran our implementation with 32 threads, the execution time of the lattice reduction (BKZ with block-size 30) was not measured.

Using 32 threads and $r = 8192$, the authors reported an execution time of 0.9 hours, i.e., 54 minutes or 3240 seconds, for the execution on a random lattice (seed 0) from the SVP-Challenge, in dimension 80 (see [51, Section 5.3, Table 2]). The execution times for both a random and an ideal lattice are exactly the same, which is surprising, considering that substantial speedups (e.g. $>50x$) can be achieved for GaussSieve on ideal lattices [109].

Despite of this, our implementation solves the very same lattice in 2896 seconds, i.e. less than 48 and a half minutes (or ≈ 0.8 hours), which represents an improvement of nearly 12%. In fact, running the *Ishiguro2014* implementation with an optimal value for r would still require not less than 45 minutes, i.e. 2700 seconds (see Section 5.1, Figure 3(a)). Since r directly influences the number of iterations required for convergence, one can compare this result to the most relaxed stopping criterion in our implementation, which is to set a target norm, as in [108]. In this case, with the same 32 threads and BKZ's block-size 30, our implementation runs in 1788 seconds, which represents an improvement factor of more than 1.5x.

The authors of *Ishiguro2014* do not present or comment on the impact of BKZ's block-size on the performance of the GaussSieve, and therefore we assume that 30 is the optimal choice for this parameter on their implementation. On the contrary, we did assess the impact that different block-sizes in BKZ have on the performance of our implementation [76, Section 5.2.2].

Summary

This section introduced a scalable parallel implementation of GaussSieve. The core idea of the proposed implementation is a lock-free list that holds the vectors in the system, combined with hand-vectorized and hand-optimized code. By slightly relaxing the properties of GaussSieve, we achieved almost linear and linear speedups up to 64 cores, depending on the tested scenario.

In comparison to the previously proposed parallel implementations of GaussSieve, our implementation performs and scales much better than *Milde2011*, and outperforms *Ishiguro2014*, by factors of between nearly 1.12x and 1.50x for lattices that are BKZ-reduced with block-size 32, and between nearly 1.39x and 1.70x for lattices that are BKZ-reduced with block-size 34.

6.3.4 Probable lock-free hash tables for HashSieve

Many of the ideas presented in this Section have been published in [73].

In most sieving algorithms, including GaussSieve, searches for nearby vectors, to reduce samples, are done in a brute-force manner; given a list of vectors L and a target vector \mathbf{v} , searching for list vectors $\mathbf{w} \in L$ which are close to this target vector \mathbf{v} is performed by simply going through the list and comparing its elements, one by one, to \mathbf{v} . Laarhoven showed that a well-known method from the field of nearest neighbor search, called locality-sensitive hashing, can be used to significantly speed up the search step in sieving, presenting an algorithm known as HashSieve [59]. The preliminary experiments in the paper

indicated that HashSieve might be faster than the fastest sieving algorithm at that point in time, GaussSieve, already in low dimensions. However, as mentioned in [59], these preliminary results were based on a comparison of naïve implementations of both algorithms. Thus, it was of major relevance to conduct a high performance, parallel implementation of HashSieve, a challenge we took on in [73], and present in this section.

A (probable) lock-free parallel variant of HashSieve

As shown in the previous section, coarse-grained parallelization is a good strategy (if not the best) for implementing sieving algorithms on shared-memory architectures. In such a scheme, each thread executes the sequential sieving kernel, i.e., generation of a sample (either from scratch or popped from stack), reduction against existing samples, and storing in memory (e.g. in a global list, as in GaussSieve). However, this results in multiple concurrent memory accesses, which must be handled via some sort of synchronization (in Section 6.3.3, we showed that for GaussSieve, this can be done with a lock-free list).

Concurrency in parallel HashSieve If we parallelize HashSieve in a coarse-grain fashion, as we did for GaussSieve, it becomes considerably more difficult to ensure correctness without resorting to very conservative (and computationally expensive) locking mechanisms. There are two concurrent operations in such a scheme:

- The concurrent insertion and removal of vectors in each bucket, which was commented on before, in Section 6.3.3.
- The concurrent use of multiple vectors throughout the execution, for actual reductions. Given that the same vector is stored in all hash tables, it is common to have a single representation of the vector in memory and then use pointers to this location, in all the hash tables, thus reducing memory considerably [73, 59]. Since every hash table has one pointer to every vector in the system, several threads can access (and potentially write) the same vector at the same time, either via the same hash table, or via different hash tables. In short, different threads can access the same buckets and vectors concurrently (even if they are working with different hash tables and different buckets).

A (probable) lock-free mechanism To address concurrency in our parallel HashSieve implementation, we implemented a probable lock-free mechanism, i.e., a synchronization mechanism implemented with locks that will likely act as a lock-free mechanism. That is, although locks are always used to access data structures, chances are that two different threads never use the same lock at the same time, which would cause them to spin, because contention is very low. To implement this probable lock-free mechanism, we introduced a variable per vector and per bucket, which is atomically updated whenever a vector is used (both to read and write). For the buckets, each thread loops until the variable is successfully set to “1”, which happens only when the value is originally “0”, as in a spin-lock. This resolves two sources of concurrency: concurrent management of buckets and concurrent accesses to the same vector (both by different threads) through the same hash table.

For vectors, which can still be accessed concurrently through different hash tables, given that our implementation also uses T pointers to each vector, as shown in Figure 6.7, each thread tries to set the variable atomically to “1” as well, but in this case the vector is ignored if the operation is not successful,

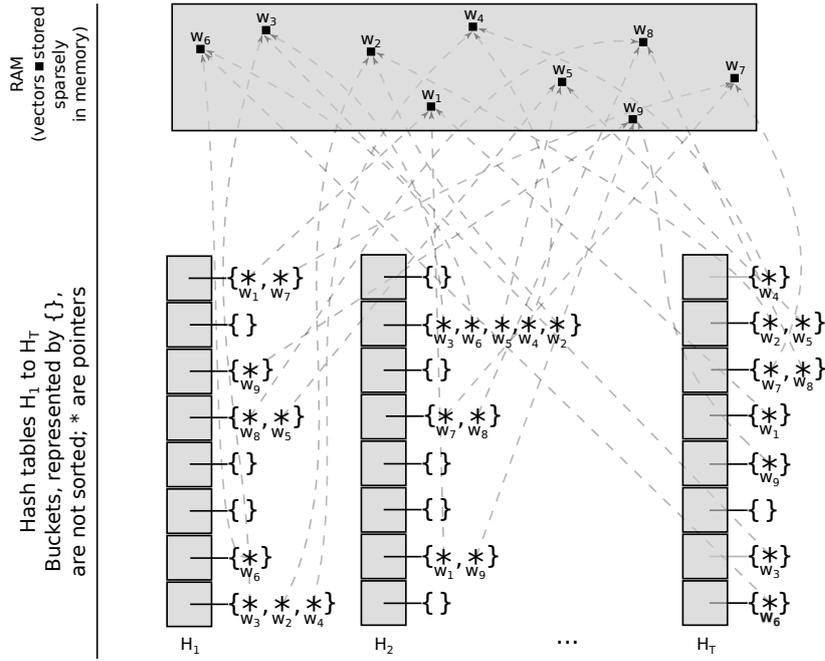


Figure 6.7: Hash tables and buckets, containing pointers to vectors in memory, in our HashSieve implementation. Example with 9 vectors in the system.

and the next vector in line is considered (technically, this is also a lock). An illustrative example of this case is the reduction of a sample \mathbf{v} against a given set of candidates $\mathbf{w}_1, \dots, \mathbf{w}_n$. If the candidate vector $\mathbf{w}_k: 1 \leq k \leq n$ is “locked” (which means that it is either being read or written by another thread), the reduction of \mathbf{v} against \mathbf{w}_k will not be attempted, and the executing thread will continue the process from \mathbf{w}_{k+1} onwards. This means that, in this specific iteration, the reduction between \mathbf{v} and \mathbf{w}_k will not be revisited again. The update of these variables is done once the vector is not needed any longer, and no atomic updates are used. This guarantees that the same vector is only accessed by one thread at a time.

There are two caveats that need to be addressed in this process. First and foremost, we refer to our implementation as a relaxed variant of HashSieve, since occasional missed reductions might occur in specific iterations, if different threads try to access the same vectors concurrently (one gets “the lock” and the others vectors are discarded, thus missing a reduction), as mentioned before. We believe that this is not too much of a problem because (1) the probability of having different threads accessing the same vectors is very low and (2) although vectors within buckets are more likely to be reducible against one another, they can still fail to be reduced. This is known from the experiments with relaxed versions of sieving algorithms, including those in Sections 6.3.2 and 6.3.3, which showed that disregarding vectors at some point does not increase the convergence time of the algorithms unless those vectors are ignored from that moment on, which is not the case in our implementation. The strongest evidence that missing reductions occasionally is not a serious issue is HashSieve itself, since HashSieve is already a somewhat relaxed version of GaussSieve [59]. In HashSieve, however, it must be noted that the probability of tested vectors to be suitable candidates is higher than in GaussSieve. Thus, our probable lock-free mechanism introduces a second level of relaxation. Again, we stress that this relaxation is not problematic, due to the aforementioned reasons, which is ultimately demonstrated by our experiments, as our implementation delivered the optimal solution with all numbers of threads and in all lattices.

The second caveat is the reason why threads will most likely never spin. This is because the number of buckets per hash table grows exponentially with the dimension of the lattice, according to the optimal values of T and K (which we used, whenever possible). For instance, we use 2^{12} buckets per hash table for a lattice in dimension 60, whereas that number increases to 2^{18} for a lattice in dimension 90. For a sensible number of threads in shared-memory systems (e.g. <128), the probability of having two or more threads accessing the same bucket is very small (plus, threads do many more operations than accessing buckets).

Figure 6.7 shows a sketch of our implementation. Our implementation represents buckets with arrays, which are more cache-friendly but also more expensive to manage, in comparison to linked lists, since we tested various data-structures and arrays delivered the best results. The arrays start with a pool of pointers, to avoid several (expensive) singular allocations, and are resized whenever needed. Inner products were vectorized with SSE 4.1 and the algorithm stops once c collisions, where $c = \alpha \times nv + \beta$, where nv is the number of vectors in the system, are generated (in the experimental section, we define both α and β). To generate samples, we used Klein’s algorithm, as implemented in GaussSieve shown in the previous section.

Experiments and Results

An analysis was carried out with several random lattices, generated with Goldstein-Mayer bases, in multiple dimensions, available on the SVP-Challenge website (all of which of seed 0). We carried out these experiments on Peacock (cf. Table 2.1). The code was compiled with Intel icpc 13.1.3. We used the `-O2` optimization flag, since it was slightly better than `-O3`. Every experiment was repeated three times and the best sample was chosen, except when said otherwise. The elapsed time of lattice reduction is not included in the results. Target norms were never used, and the norm of the output vector of each and every run (sequential and parallel) was always the same. We used the default stopping criterion, i.e. $\alpha = 0.1$ and $\beta = 200$, and we used version 5.5.2 of NTL for basis reduction with BKZ.

In [73], we presented a number of results, pertaining to (1) the sequential implementation, where we compare HashSieve against GaussSieve and quantify the overhead of our probable lock-free mechanism, (2) our parallel variant, where we analyze its scalability, compare it with the parallel GaussSieve presented in Section 6.3.3 and test the practicability of our variant regarding the highest possible lattice dimensions.

HashSieve vs GaussSieve and probably lock-free overhead. Regarding the comparison of our HashSieve implementation against the GaussSieve implementation in Section 6.3.3, we concluded that HashSieve is $\approx 2.2x$ to $2.5x$ faster, depending on having the probable lock-free mechanism on or off, respectively. This also indicates that the probable lock-free system causes about a 20% overhead. However, it should be noticed that for parallel executions, some computations will likely be discarded due to the relaxation of the algorithm, thereby amortizing this overhead. For these trials, we used the optimal values of K , T and `BUCKETS`.

Convergence rate and memory. In [73], we showed results pertaining to the convergence rate of GaussSieve and HashSieve, in terms of used vectors, and memory usage. In this dissertation, we only report on the latter, and the reader is referred to [73] for the results pertaining to the convergence rate of GaussSieve and HashSieve. Studying the memory usage of HashSieve is of major relevance, as HashSieve

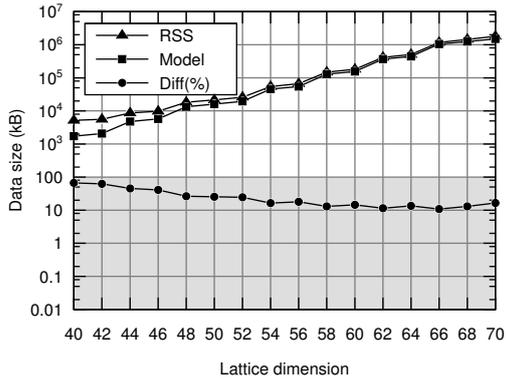


Figure 6.8: Data size stored in RAM and prediction of our model. The grayed out area concerns the percentage difference between both.

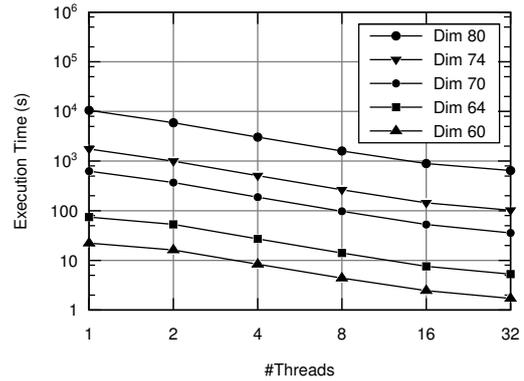


Figure 6.9: Execution time, in seconds, for HashSieve with 1-32 threads, on lattices in dimensions 60-80 (less is better).

also makes use of many hash tables of pointers to vectors, which increases memory consumption. For big dimensions, this becomes a problem, since even 128 GB of RAM are easily exceeded.

In [73], we presented a model for the memory usage of our implementation. We considered that there are two main relevant contributions to the total memory of the application, other than the auxiliary data structures. These are the memory used to store vectors and the arrays that represent the hash table buckets. Using least square fitting, we estimate that the number of vectors is governed by the function $v(n) = 2^{(0.21n+1.95)}$, and the size of each vector is $2n + 17$ bytes, where n is the dimension of the lattice (we use shorts for each coordinate). In our implementation, we extend each bucket in 20 positions whenever they become full.

In order to verify the feasibility of our model, we conducted some experiments to capture the Resident Set Size (RSS), i.e. the amount of process's memory that is held in RAM, of our application. As Figure 6.8 shows, our model gets very close to the actual RSS of our implementation for big lattice dimensions. This is due to the fact that, although our model only accounts for the main (i.e., bigger) data structures in the implementation, they grow exponentially in size with the lattice dimension, and so the remaining data structures become negligible. In particular, Figure 6.8 shows that for lattices bigger than dimension 58, the difference between the actual RSS and our model is always lower than 20%, and most of the times very close to 10%.

Scalability of the parallel implementation. In [73], we also showed the trials that we conducted to quantify the scalability of our implementation on our test platform, for random lattices in dimensions 60, 64, 70, 74 and 80. Lower dimensions are either solved very quickly or the lattice reduction process finds the shortest vector per se, rendering a scalability analysis worthless. Running the implementation for higher dimensions, on the other hand, is impractical for a single thread. We were still able to use the optimal parameters of HashSieve in these dimensions. We BKZ-reduced the lattices up to dimension 70 with $\beta = 20$, and higher dimensions with $\beta = 30$.

As in [76], the time spent within the sampling routine increases for lower values of Klein's algorithm parameter d , which lowers the scalability of the implementation because the sampler routine does not scale well (again, we show a solution to this in [70]). For that reason, and to properly assess the scalability of our implementation, we set the Klein's algorithm parameter d to $\log(n)/20$ for every lattice.

Cores	Dimension 60		Dimension 64		Dimension 70		Dimension 74		Dimension 80	
	S	E	S	E	S	E	S	E	S	E
2	1.37x	69%	1.40x	70%	1.69x	84%	1.76x	88%	1.77x	89%
4	2.66x	67%	2.75x	69%	3.33x	83%	3.43x	86%	3.45x	86%
8	5.07x	63%	5.27x	66%	6.42x	80%	6.64x	83%	6.56x	82%
16	9.05x	57%	9.80x	61%	11.83x	74%	12.16x	76%	11.76x	74%
32	13.01x	41%	14.08x	44%	17.48x	55%	17.14x	54%	16.26x	51%

Table 6.5: Speedup (S) / Efficiency (E) of our HashSieve implementation on 5 random lattices (dimensions 60-80). SMT in grayed out rows.

Figure 6.9 shows the execution time of our implementation, for 1-32 threads. The application scales well for up to 32 threads, for various lattices between dimension 60 and 80. Unfortunately, scalability experiments with low core counts are impractical for higher lattices, since every and each thread set-up is executed three times (e.g., solving a lattice in dimension 84 with 1 thread would take about 38 hours). As shown in Table 6.5, the speedup of our implementation increases, in general, with the dimension, because the higher the dimension, the more time is spent on reducing each sample, rather than on generating more samples. In some cases, our implementation achieves efficiency levels of almost 90%. We believe that the integration of a very scalable sampler in our implementation would increase its scalability. This could also be achieved with larger values for Klein’s parameter d (e.g., $\log(n)$), as mentioned above, but values for d decrease the overall performance of the algorithm, because bigger vectors are sampled and the algorithm converges faster when fed with shorter algorithms.

Practicability experiments. The main limitation we encountered during the trials we conducted was the lack of RAM to use the optimal parameters of HashSieve. Since Peacock we used has 128 GB of RAM, our experiments were limited to $K = 18$, $T = 1828$ and $BUCKETS = 131072$, i.e., far from being the optimal parameters. Unfortunately, the optimal parameters are impractical for machines that are not equipped with unusual amounts of RAM. For instance, the optimal parameters for dimension 96, ($K = 21$, $T = 5345$ and $BUCKETS = 2^{20}$) would require almost 1 TB of RAM only for the data structures that are allocated at launch time. Therefore, we expect that better parameters would result in a much better execution time of our implementation. Yet, by using 32 cores with SMT, we were able to solve the SVP on several lattices, from dimension 86 to dimension 96, all in less than 24h, as Table 6.6 shows.

Dim	86	88	90	92	94	96
Time (h)	1.92	2.38	3.43	7.35	11.07	17.38

Table 6.6: Execution time of our parallel implementation of HashSieve running with 32 threads on the test platform, in hours.

For these experiments, we used BKZ with block size $\beta = 34$, and Klein’s parameter $d = \log(n)/70$ (which is expected to be the best for dimensions higher than 80, even if poor scalability is achieved - a claim that is too time consuming to be verified). Each execution was run only once.

Wrap up

This section summarized the contributions of [73], which presented a parallel implementation of HashSieve. Our implementation scales well on Peacock, a 16-core system with SMT, and outperforms the most

efficient shared-memory GaussSieve implementation, published in [76] and reviewed in Section 6.3.3. Therefore, we were able to verify in practice the results that were expected from theory.

In parallel, HashSieve has more concurrent operations than GaussSieve. Our implementation uses a probable lock-free algorithm, which might cause missed reductions, but chances are that, for large numbers of hash tables and buckets, no threads actually block. Our implementation is able to solve the SVP on an arbitrary lattice in dimension 96 in less than 17.5 hours, with Peacock, our 16-core test machine.

Although HashSieve is considerably more practical than its counterparts on high-dimensional lattices, sieving algorithms are still less efficient than enumeration with extreme pruning on random lattices, yet very promising. We also did a least squares fit of our implementation's execution times, for lattices in dimensions 80-96 (all of which with BKZ with block-size $\beta = 34$, Klein's parameter $d = \log(n)/70$), in seconds. The fit lies between $2^{(0.32n-15)}$ and $2^{(0.33n-16)}$.

The main drawback of HashSieve is the amount of memory used. In fact, this limited our expectation of determining how practical HashSieve is for high dimensional lattices with optimal parameters, since our system limited our experiments to 128 GB. With more RAM, one would be able to determine if HashSieve can outperform enumeration algorithms with extreme pruning.

6.3.5 A parallel variant of LDSieve

Introduced in [10], LDSieve differs from HashSieve in the way vectors are mapped onto buckets and how buckets are searched. Preliminary experiments suggested that LDSieve could surpass HashSieve, but these experiments were done with sequential, not optimized implementations. In [74], we presented a parallel variant of LDSieve, analyzed the algorithm and carried out extensive comparisons against HashSieve.

The variant

The underlying idea of our variant is very similar to HashSieve (i.e. we also used probable lock-freeness, as we introduced in the previous section). Indeed, we implemented LDSieve with HashSieve as a starting point, resulting in equally optimized implementations of the algorithms, thus allowing for a fair comparison. Our implementation of LDSieve is also parallel at a coarse-grained level (i.e. each thread executes the sequential sieving kernel and all threads store vectors in the same, shared, data-structures).

When it comes to concurrency, LDSieve is similar to HashSieve: the concurrent operations are additions/removals of vectors from the same buckets, and the concurrent use (with at least one thread writing) of vectors. In addition, different tables for the inner products between samples and codes have to be used by different threads. Although LDSieve uses a single hash table, in contrast to HashSieve, the problem remains as different threads can access the same buckets and the same vectors simultaneously.

Some vectors in our variant are ignored during the reduction process if they are locked by another thread, similarly to our HashSieve implementation in Section 6.3.4. We employ one efficient lock per vector and bucket, which threads use when accessing these structures. Threads spin when they cannot acquire the locks of the buckets. If the vectors are locked, they are disregarded and the reduction process moves on to the next candidate vector. As mentioned before, the likelihood of successful vector reductions is low, and if one vector is disregarded, the bucket can still contain other vectors to proceed with the reduction process.

Assessment

The original paper about LDSieve reported on experiments with an implementation of the algorithm³, which we refer to as the *baseline implementation* from here on. We carried out an extensive assessment of our implementation, measuring the performance against the baseline implementation, its scalability and other relevant factors, such as the best codesize in practice. Although many other tests could have been conducted, the tests we performed are absolutely essential to provide insight into LDSieve and its behaviour in practice. We used several random lattices, generated with Goldstein-Mayer bases, in multiple dimensions, available on the SVP-Challenge website (all of which generated with seed 0), and we run the tests on Lara (cf. Table 2.1).

Comparison against the baseline We compared the performance of our LDSieve variant, for a single thread (as the baseline is not parallel), in terms of execution time and vectors to reach convergence, with that of the original paper. We showed that our version is as much as 50x faster than the implementation of the original paper, and the speedup increases with the lattice dimension. To achieve this, we capitalize on the vectors that are already in the (reduced) lattice basis by introducing them in the algorithm, apart from the code optimizations we proposed.

Scalability In [74], we showed that our implementation scales very well for up to 64 threads on the machine, for any lattice dimension. After 16 threads we note a decrease on the scalability presumably due to additional overhead of inter-socket communication [74]. Superlinear speedups were achieved in specific instances because thread scheduling can result in different, potentially faster reduction processes. We expect that higher dimensions will result in identical scalability levels, since cache is already exhausted for these dimensions. Lower dimensions might result in weaker scalability, due to lack of enough work to overcome the overhead of thread creation and scheduling, but those are of lesser interest.

Codesize in practice In theory, the ideal codesize is $T = 2^{0.292n+O(n)}$, but it does not mean that this is the best value in practice. In [74], we showed, for the first time, the effect of different codesizes on the execution time of the algorithm, for different lattices, to determine the best parameters in practice. In particular, we tested the implementation running with 64 threads, on lattices in dimensions 76, 80, 84

³<https://github.com/lucas/LDSieve/>

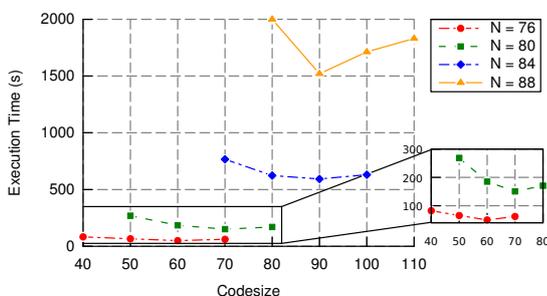


Figure 6.10: Execution time for different codesizes in LDSieve, for lattices in dimensions 76-88. 64 threads. BKZ with $\beta = 34$ and $M = 4$.

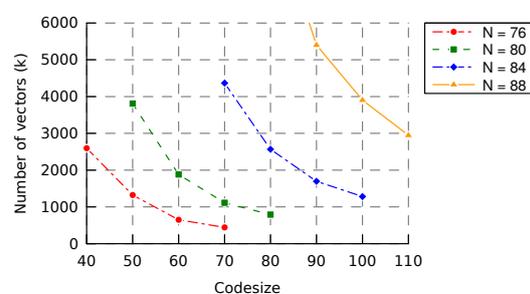


Figure 6.11: Vectors (in thousands) for different codesizes in LDSieve, for lattices in dimensions 76-88. 64 threads. BKZ with $\beta = 34$ and $M = 4$.

and 88, as shown in Figure 6.10; note the zoom-in into the range of N between 40 and 80 on the right side of the figure. The curves of the execution time for different lattice dimensions have the form of a parabola, i.e. the execution time decreases with increased codesize, but starts to increase after some point. In particular, all curves have a well defined minimum (60 for dimension 76, 70 for dimension 80 and 90 for dimensions 84 and 88). The difference in the execution time between the best codesize and the second best varies between 10 and 30%.

The codesize influences the number of vectors that the algorithm uses. However, the codesize that renders LDSieve faster is not necessarily the codesize that leads to fewer vectors. For instance, for the lattice in dimension 84, the implementation used approximately 4.4, 2.6, 1.7 and 1.3 million vectors for codesizes 70, 80, 90 and 100, respectively. Although codesize 100 results in 30% less vectors than codesize 90, it is the latter that provides the best execution time. There are two fundamental reasons for this:

- higher codesizes render the algorithm increasingly selective in the bucket selection, which means that fewer reductions are missed (smaller list size) but also that more buckets must be checked for reductions (more time);
- increasing codesize renders the partial inner product lists bigger, which in turn leads to more cache misses and more requests to higher levels of the memory hierarchy.

For high codesizes, the inner product lists become a dominant factor in memory usage. In particular, the memory usage grows substantially with the codesize, in all dimensions. For instance, for a lattice in dimension 88, the implementation requires approximately 52.7 GBs of memory with codesize 80, and over 122 GBs for codesize 110.

LDSieve vs HashSieve

The main motivation for implementing and assessing the practicability of LDSieve is that both theoretical analyses and preliminary experiments suggest that LDSieve can outperform the best sieving SVP algorithm in practice, HashSieve, for a sufficiently large lattice dimension. First-hand experiments with LDSieve, reported in the original paper [10], suggest that, for sequential implementations, LDSieve outperforms HashSieve for lattices in dimension 72 and onwards. Therefore, it is of major relevance to verify this claim when highly (but equally) optimized, parallel versions of LDSieve and HashSieve are compared against each other.

To carry out this comparison, we used the HashSieve implementation presented in Section 6.3.4. Figure 6.12 shows the execution time of our parallel implementation of LDSieve, with $M=3$ for lattices in dimension 81, 87 and 90 (which are divisible by 3), and $M=4$ for lattices in dimension 80, 88 and 92 (which are divisible by 4), against the aforementioned HashSieve implementation. The lattice in dimension 84, which is divisible both by 3 and 4, was solved with LDSieve set both with $M=3$ and $M=4$. We did not test higher dimensions as HashSieve requires more memory than that available in the system (e.g. in dimension 94, HashSieve requires about 800GBs of memory). Probing is a technique which reduces memory consumption (i.e. fewer hash tables and buckets) at the cost of increased execution time (e.g. more buckets are checked) [59, Section 5]. We decided not to compare HashSieve and LDSieve in these conditions given that, as LDSieve does not require probing for these tests, the comparison would be unfair.

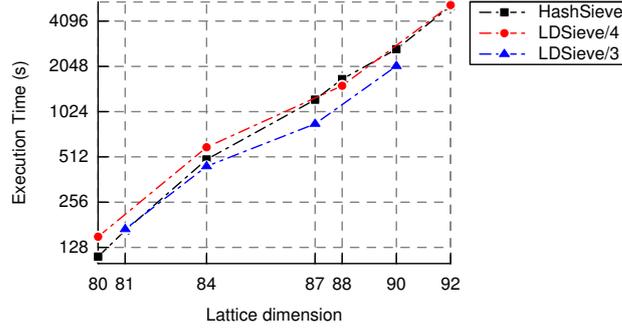


Figure 6.12: Comparison of our parallel implementation of LDSieve against HashSieve, for lattices in dimensions 80-92, running with 64 threads. Lattice bases are reduced with BKZ, with block-size $\beta = 34$. $M = 4$ for $N = 80, 84, 88$ and 92 . $M = 3$ for $N = 81, 84, 87$ and 90 .

Our experiments support the claim that LDSieve overcomes HashSieve for high dimensions. In particular, our LDSieve variant performs similarly to HashSieve for the lattice in dimension 81, but beats HashSieve for higher dimensions. This holds true for any of the tested lattice dimensions when M was set to 3. However, with M set to 4, this is not true for all dimensions, which is an interesting phenomenon. For instance, LDSieve with M set to 4 is better than HashSieve for the lattice in dimension 88, but is slightly worse for the lattice in dimension 92. While this could be explained by noise introduced by the lattices themselves, as we tested a single lattice for each dimension, we believe that this happens because LDSieve is more efficient in practice with $M=3$ within the range of lattice dimensions that we tested. Another result suggesting this is that LDSieve is better than HashSieve for the lattice in dimension 87 when M is set to 3, but not when M is set to 4.

Changing M has implications both in the algorithm and the implementation. With regard to the impact in the algorithm, increasing M has both positive and negative consequences. As M is increased, the size CS of the subcode S decreases, and so the overhead of computing block-wise inner products decreases as fewer inner products have to be computed. This means that finding the right buckets becomes cheaper. On the other hand, increasing M makes the product code more structured and therefore less *random*. As a result, the theoretical analysis of [10] starts to be less accurate, and there will be more imbalances in the buckets: many buckets will be empty, and a few buckets will contain many vectors. This means the bucketing strategy becomes less effective, as ideally each bucket has roughly the same number of vectors. Overall, theoretically it is not quite clear which value of M is optimal, and only further extensive experiments can answer this question. As for the implications in the implementation, M changes the size of some data structures, which we showed not to be problematic as far as cache locality is concerned (lattice dimensions with very large M values that could raise cache locality concerns are not practical).

Memory and number of vectors used

To handle concurrency in LDSieve, we replicate some data structures per thread. Even with a thread count as high as 64 threads, our implementation of LDSieve is considerably more memory efficient than HashSieve after dimension 84, both when M is set to 3 and 4⁴, as shown in Table 6.7 (note that dimension 84 was tested with M set to 3 and 4 as it is divisible by both). For instance, our LDSieve variant uses as

⁴We also conducted benchmarks with M set to 2. The performance of the algorithm was lower than with M set to 3 and 4.

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

Table 6.7: Memory (in GBs) and number of vectors used for HashSieve and LDSieve for lattices in dimensions 80-92, running with 64 threads. LDSieve set with $M=3$ and $M=4$, multiple codesizes.

much as 5 times less memory than HashSieve (e.g. in dimension 90). LDSieve has thus a significant edge over HashSieve as it mitigates the key problem of HashSieve, its memory consumption.

Moreover, although our LDSieve variant requires much less memory than HashSieve, it uses substantially higher numbers of vectors than HashSieve to reach convergence, as also shown in Table 6.7. For instance, in dimension 92, our LDSieve implementation uses 5 times more vectors than HashSieve, but the execution time of both implementations is identical, as shown in Figure 6.12. It is important to point out that the number of vectors stored in memory is not the main contributor to the overall memory consumption (of either implementation). In both implementations, most of the used memory is used to store the (various) Hash table(s) and respective buckets.

Good choices of the codesize and M are essential to achieve both low memory consumption levels and satisfactory numbers of vectors used (as we showed in Subsection 6.3.5). It is hard to infer from theory good values for these parameters, as they depend on many factors and implementation details.

Wrap up

This subsection summarizes fundamental questions pertaining to the practicability of LDSieve, which we addressed in [74]. We presented a very efficient parallel variant of the algorithm, which we used as a fundamental tool to address fundamental questions around the algorithm, and which achieves speedup factors of 50x over the original implementation of LDSieve, for the same parameters. The first and perhaps most relevant question around LDSieve is whether it can beat HashSieve, the best sieving algorithm for the SVP to this day, in practice. In particular, it is relevant to assess this claim (1) with equally optimized sequential implementations of both algorithms, (2) verifying if LDSieve lends itself to parallelism and finally (3) whether a parallel implementation of LDSieve compares well to a parallel version of HashSieve, which is known to scale well on shared-memory systems.

To verify this, we conducted a thorough analysis of the behaviour of our parallel implementation LDSieve, in comparison to HashSieve. Our LDSieve variant scales linearly on shared-memory systems (at least up to 16 threads) and is better than the best HashSieve implementation to this day, when $M=3$, for dimensions 81-90, being competitive when $M=4$. In addition, we conclude that there are both pros and cons of LDSieve over HashSieve. An advantage of LDSieve is that it spends considerably less memory for very high dimensions (>92), and if memory becomes a limitation in the system, LDSieve is preferable over HashSieve. Probing is expected to affect both algorithms to the same extent, LDSieve will have an advantage over HashSieve. On the other hand, a disadvantage of LDSieve is that multiple

parameters have to be selected to get optimal performance (note that the optimal parameters in theory for HashSieve perform very well in practice). In particular, a wrong codesize selection may render the algorithm impractical. Unfortunately, the best codesize in practice can only be found through empirical benchmarks, which are time-consuming. In addition, if a new lattice dimension is to be solved, there is no point in running preliminary benchmarks to find out the best codesize, as one would already have the solution after running the preliminary benchmarks. Thus, we have also provided insight, for the first time, on how the codesize should grow as a function of the lattice dimension.

6.4 The memory problem

As we showed in the previous sections, the memory consumption of HashSieve and LDSieve is very high, and eventually it actually becomes an impediment to execute the algorithms in higher lattice dimensions. In [70] and [72] we studied the optimization of the memory access pattern and memory utilization of HashSieve, which serves as an excellent example of a large, irregular parallel application. Our optimizations are also extensible to the other sieving implementations we presented, and other classes of algorithms. This section summarises the findings reported in those papers.

6.4.1 Improving memory efficiency through code optimizations

Many of the ideas presented in this Section have been published in [70].

Sieving algorithms are memory-bound, i.e. their execution time is mainly governed by the time that is spent in the memory hierarchy, both retrieving and storing data. HashSieve is no exception. In [70] we computed the arithmetic intensity of HashSieve.

Kernel	Dot product	Add	Hash Val
Operations	$2n$	$n + 3$	$3n/2 + 4$
Load/stored bytes	$12n + 16$	$6n + 36$	$10n + 8$
Arithmetic intensity	$\approx 1/6$	$\approx 1/6$	$\approx 1/7$

Table 6.8: CPU operations and bytes fetched from memory for the three kernels of HashSieve, for $n=80$.

Table 6.8 presents the arithmetic intensity (ratio of CPU and memory operations) of the main kernels of the algorithm for $n = 80$, imputing a maximum arithmetic intensity of $\approx 1/6$, which is rather low. Furthermore, the arithmetic intensity decreases with the lattice dimension. The actual arithmetic intensity is way below this mark: these kernels aside, the algorithm essentially fetches high volumes of data from memory, so that these kernels can be executed. In particular, the algorithm (1) loads hash buckets, (2) adds and (3) removes vectors to/from hash buckets. These procedures are difficult to bound in terms of memory loads and stores, but they will only contribute to lower the overall arithmetic intensity. More than that, they contribute to very high cache miss ratios.

This suggests that there is a window of opportunity to optimize sieving algorithms (and the HashSieve implementation presented in Section 6.3.4 in particular) through improved memory handling. An effective way to enhance the memory access pattern and memory utilization of an implementation is through code modifications. This section shows a number of techniques, published in [70], which we used to improve the HashSieve implementation presented in [73] and the LDSieve implementation presented in [74]. We run the algorithms on Peacock, with 32 threads, and we reduced the lattices with BKZ, with block-size 34.

Memory and object pools

The implementation proposed in [73] and presented in Section 6.3.4 implements hash buckets with pools (of pointers to vectors), an attempt to mitigate the cost of adding vectors to the hash tables. These pools are re-sized whenever needed, with *realloc*. For samples, no pools are used: when vectors are sampled, they are stored with a *malloc* call. This is inefficient for a few reasons. First, this might lead to memory fragmentation and reduced locality of reference because there is no assurance that these vectors will be stored contiguously in memory. Second, this incurs additional overhead because each of these *malloc* calls is an operating system call and they are invoked in parallel by several threads, which requires a locking mechanism to control concurrency.

We improved this with another pool of vectors, private per thread, this time used to store sample vectors. Essentially, we allocate a big array of vectors, at the beginning of the application, along with its size and its latest used position:

```
Vector *pool = malloc(POOL_SIZE*sizeof(Vector));
int pool_size = POOL_SIZE, latest_pos = 0;
```

Whenever a vector of the pool is used, the `pool_size` variable is decremented and the `latest_pos` variable is incremented. When a vector is used, the size of the pool is checked, and the pool is resized when it is empty. Such a vector pool minimizes the number of *malloc* calls and improves spatial locality, since vectors are consecutively stored in memory.

With this vector pool, the performance grew by over 10%. In general, the speedup grows with the lattice dimension. We tested this optimization for up to dimension 86, as it is impractical to extend these experiments to higher dimensions. It is hard to speculate if the speedup will continue to grow for higher dimensions, because the benefit depends both on the number and the timing of each *malloc* call, which varies between executions. The gains are thus larger for more overlaps between these calls.

Generic memory allocators

Generic memory allocators are an alternative to memory pools. Although these mechanisms are considerably different in terms of aim and design, both usually accelerate memory allocation. Generic memory allocators are usually more scalable and more efficient than the default *malloc* system call, especially on multi-threaded applications. `Hoard`⁵, and `tcmalloc`⁶ are among the most popular generic memory allocators. Some, as `tcmalloc`, are drop-in replacements of *malloc* and other system calls, while others provide an API.

Our HashSieve implementation allocates memory in parallel in three different stages: (i) during the initialization, where all the hash tables and buckets are allocated, (ii) to create the various object pools (as discussed in the previous subsection), private to each thread, and (iii) to extend the memory reserved to every bucket (with *realloc*). The latest is the most problematic among them, because even for dimension 80, millions of *reallocs* are in fact called. The number of *reallocs* is much bigger than the number of vectors used, which means that one vector is moved around in many hash tables throughout the application.

⁵<http://www.hoard.org/>

⁶<http://goog-perftools.sourceforge.net/doc/tcmalloc.html>

As shown in [70], the use of `tcmalloc` is beneficial for our application and this benefit increases with the lattice dimension. Again, we conducted experiments for up to dimension 86. In some instances, we obtained speedups of over 5%. However, the actual speedup boost depends not only on the number of allocations that are performed, but also on the number of overlapping mallocs that are avoided, which varies from execution to execution. It should be noted that the algorithm performs differently for different lattices, even in the same dimension.

Manual prefetching

The last of the optimizations we applied was software-based prefetching, with hand-inserted prefetch directives. Although it is usually claimed that achieving speedups with such a scheme is rather hard for irregular applications [134, 33], we did use prefetching successfully in HashSieve.

There are many opportunities on HashSieve to prefetch data, which are not captured by the compiler because they depend on runtime values of subsequent iterations. A representative example is the removal of one vector from all the hash tables in the system, which is shown in the code below.

```
for(int t = 0; t < T; t++){
    hash_value = hash_function(w, t);
    bucket_remove(&HashTables[t][hash_value], w);
}
```

As the compiler does not know the subsequent value of `hash_value`, it cannot prefetch the data where `HashTables` of iteration $t+1$ resides at. As programmers, we can not only make it more explicit to the compiler, as we can also prefetch the data. This is achieved by replacing the previous loop with:

```
int hash_values[T];
for(int t = 0; t < T; t++){
    hash_values[t] = hash_function(w, t);
}
for(int t = 0; t < T-1; t++){
    //Prefetch HashTables[t+1][hash_values[t+1]]
    bucket_remove(&HashTables[t][hash_values[t]], w);
}
bucket_remove(&HashTables[T][hash_values[T]], w);
```

This calculates all the indices upfront and prefetches the data needed in iteration $t+1$ in iteration t . As such, when iteration $t+1$ is executed, the data will already be in cache (or the latency will be small, since data is requested before). This can be (and is) replicated throughout the code, for insertion and removal of vectors from hash buckets, and other minor operations. This optimization delivered a speedup of about 10-15%, depending on the lattice dimensions [70].

6.4.2 Improving memory access for NUMA machines

Many of the ideas presented in this Section have been published in [72].

Improving the memory access behaviour and usage of parallel applications is of prime importance in high-performance computing. This is especially relevant on Non-Uniform Memory Access (NUMA) architectures, which have multiple memory controllers (each of which represents a NUMA node). In this

context, there are two fundamental issues: the *locality* of memory accesses, and the *balance* of memory accesses among controllers. Memory accesses are said local if they are served by local NUMA nodes, and remote otherwise. The balance of memory accesses among controllers has to do with the number of accesses each controller serves. Perfect balance is achieved when all NUMA nodes serve the same number of memory access requests.

HashSieve falls within the class of applications with large memory footprints and irregular memory access patterns. Optimizing memory accesses is thus very important (as we also showed in the previous subsection), because these applications spend a considerable fraction of their execution time accessing memory. In the following, we show an analysis of HashSieve in terms of thread communication, memory access locality and balance of memory accesses, on NUMA machines, and we show how these metrics can be improved, so the overall performance of the algorithm is improved as well. These results are based on [72].

HashSieve's memory access behaviour

In [72], we performed a memory access characterization of HashSieve with a custom memory tracer built with the Pin Dynamic Binary Instrumentation (DBI) tool. We used two versions of HashSieve: *the baseline version*, which is the implementation presented in Section 6.3.4, and *the optimized version*, which is the baseline version with the memory improvements, through code optimization, which we described in Section 6.4.1. We ran both versions with 64 threads on Adriana (cf. Table 2.1) and recorded all memory accesses by all threads, at the page granularity. We then analyzed the memory access behaviour based on the trace.

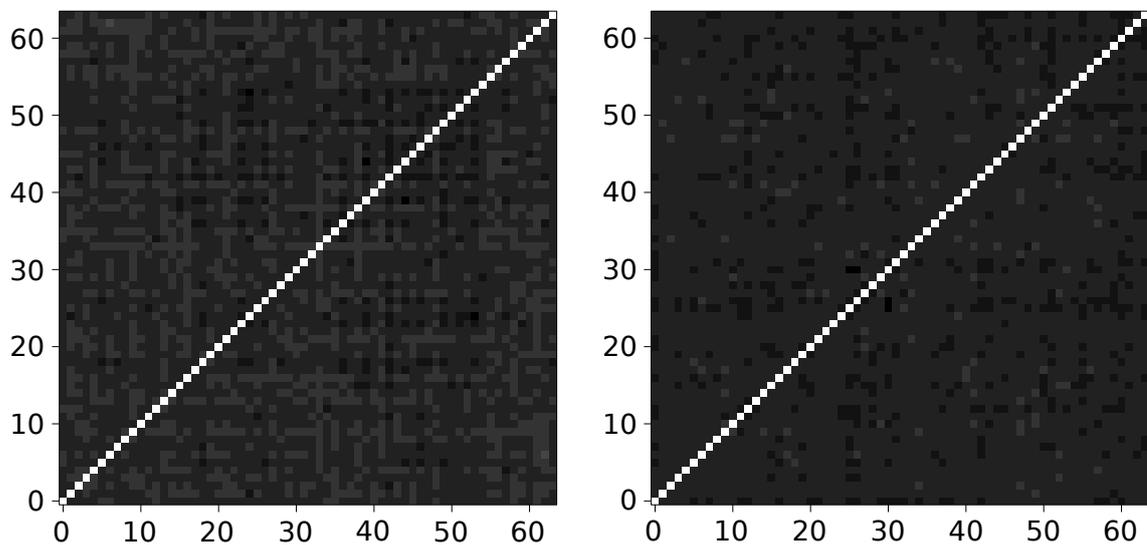


Figure 6.13: Communication patterns of HashSieve (baseline version on the left and optimized version on the right). Axes show the thread IDs, with 64 threads. Cells show the amount of communication between all pairs of threads. Darker cells indicate more communication.

The communication matrices of HashSieve are shown in Figure 6.13. In the matrices, darker cells illustrate larger amounts of communication between threads. The baseline version has very similar levels of communication between all threads, which suggests an unstructured behaviour. As both matrices are normalized, the slightly darker pattern of the optimized version indicates a small increase in the amount of

communication. Although difficult to see in the figure, the optimized version also has a more structured communication behavior, with more communication between neighbouring threads. For the baseline version, we expect only small gains from a thread mapping policy, as no thread mapping can optimize the overall communication. However, the optimized version is more suitable for thread mapping due to its slightly more structured behaviour.

Locality of memory accesses To determine the locality of memory accesses of HashSieve, we calculated the exclusivity of the memory accesses to each page, as well as the average (weighted) exclusivity for the whole application. Higher exclusivity values indicate a higher suitability for locality-based data mapping. The results of the exclusivity analysis are shown in Figure 6.14. In the figure, each dot represents a page. The horizontal axis shows the exclusivity, while the vertical axis shows the number of memory accesses to each page.

For the baseline version of HashSieve, the results show a lot of pages with a very low exclusivity (towards the minimum exclusivity of 25%). Most of these shared pages are allocated on the first NUMA node and tend to have more memory accesses than the pages with a higher exclusivity. Some pages have the maximum exclusivity of 100%. These pages correspond to memory areas private to threads, such as the memory pools of each thread. In the optimized version of HashSieve, the average exclusivity increases slightly, from 62.2% to 64.3%. We can see that the overall access distribution is pushed towards pages with a higher exclusivity. We also note that the balance of the pages is improved with the optimized version, which will be discussed in the next section. Despite the increased exclusivity, the value is still low enough such that we expect only limited improvements from a locality-based policy.

Balance of memory accesses To analyze memory access balance, we evaluate how many pages are allocated on each NUMA node with a first-touch policy, as well as the number of memory accesses to the nodes. Higher imbalance indicates higher suitability for balance-based data mapping. The balance is also illustrated in Figure 6.14. The total number of memory accesses by the nodes is shown above each plot.

For the baseline version of HashSieve, Figure 6.14 shows that the majority of pages (about 92%) will be allocated on NUMA node 1 with a first-touch policy. These pages also receive a large number of memory accesses, leading to a severely imbalanced distribution of memory accesses between NUMA nodes (47% of memory accesses are handled by node 1). In the optimized version, the distribution of pages is considerably better, since both the number of pages decreased on node 1 and increased on the other nodes, as shown in Figure 6.14. This leads to an improved distribution of memory accesses (41% are handled by node 1). Despite these improvements, the overall behaviour is still significantly imbalanced, suggesting that an improved mapping could result in further gains.

Results with different data mapping policies The results showed that both page exclusivity and memory access balance have been improved in the optimized version of HashSieve. However, exclusivity and balance remain quite low, since most scientific applications have exclusivities well above 80% [31]. Therefore, we do not expect big improvements from locality-based data mapping policies. Balance-based policies might deliver better results, since the application incurs considerable memory access imbalance.

We conducted the experiments on Adriana (cf. Table 2.1). The system has a NUMA factor of 1.5 and page size of 4 KB. The NUMA factor, defined as the latency between memory accesses to remote NUMA

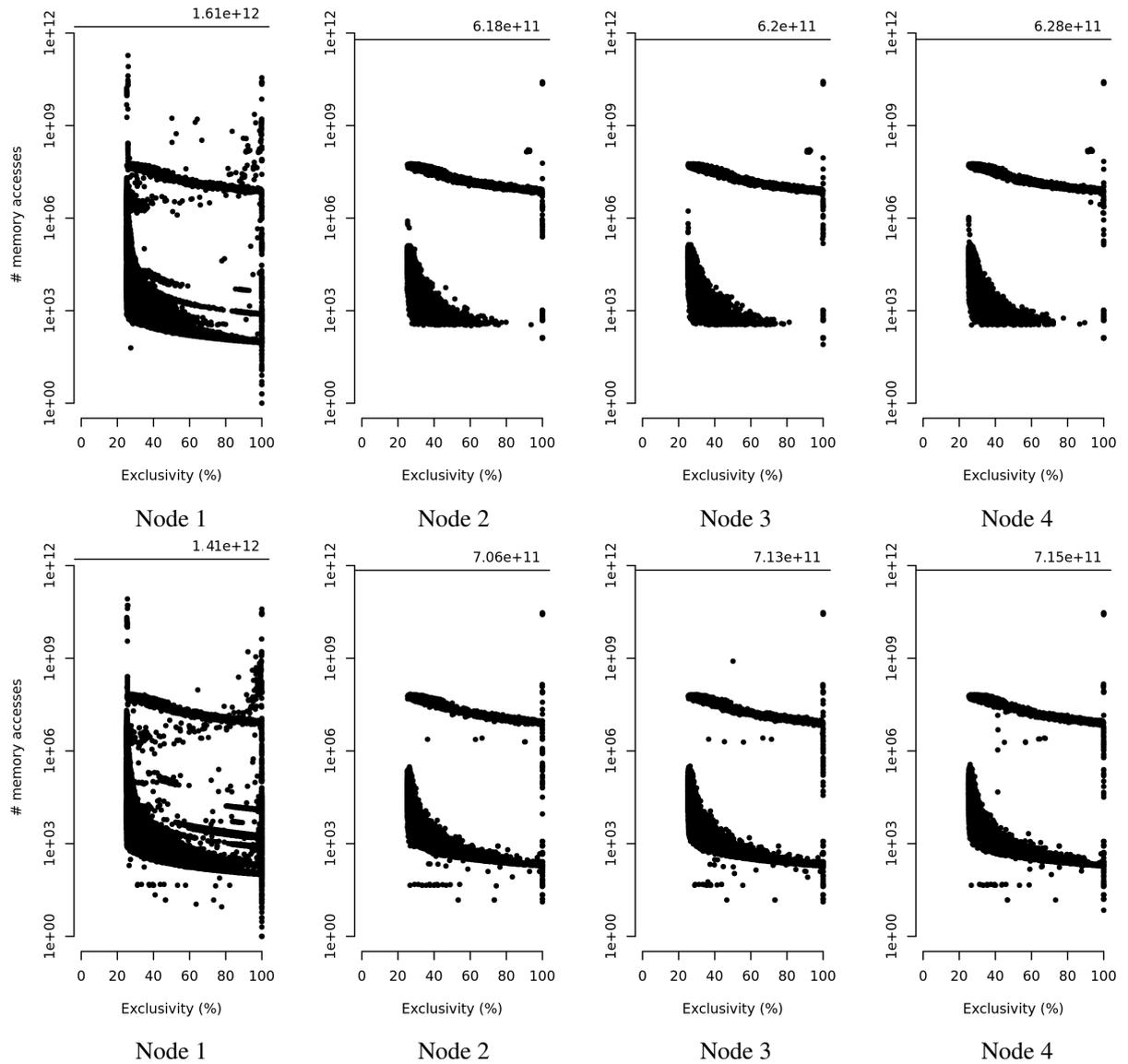


Figure 6.14: Memory access pattern of HashSieve (baseline version on top, optimized version on the bottom). Each dot represents a page. Pages are distributed to NUMA nodes according to a first-touch policy. The vertical axis displays the number of memory accesses to each page in logarithmic scale. The horizontal line on top of the graph indicates the total number of memory accesses performed by each node.

nodes divided by the latency to access local nodes, was measured with the Lmbench tool⁷. HashSieve was executed with 64 threads, given that the machine has 64 cores.

Mapping Policies We compare five mapping policies: OS, Compact, Interleave, kMAF and NUMA Balancing. The three first policies perform no page migrations during execution. kMAF and NUMA Balancing are dynamic mechanisms that migrate pages between NUMA nodes at runtime.

For the *OS* mapping, we run an unmodified Linux kernel (version 3.8), and use its default first-touch mapping policy. The NUMA Balancing mechanism [24] is disabled in this configuration. The *Compact* thread mapping is a simple mechanism to improve memory affinity by placing threads with neighbouring

⁷www.bitmover.com/lmbench/

IDs (such as threads 0 and 1) close to each other in the memory hierarchy, such as on same cores. In the *Interleave* policy, pages are assigned to NUMA nodes according to their address. This policy ensures that memory accesses to consecutive addresses are distributed among the nodes. Interleave is available on Linux with the `numactl` tool.

We also use the *NUMA Balancing* mechanism [24] of the Linux kernel. The mechanism uses a sampling-based next-touch migration policy. NUMA Balancing gathers information about memory accesses from page faults and uses them to balance the memory pressure between NUMA nodes. To increase detection accuracy, the kernel periodically introduces extra page faults during execution. The *kMAF* mechanism [30] performs a locality-based thread and data mapping of parallel applications. Similar to NUMA Balancing, this mechanism uses page faults to determine memory accesses behaviour, but keeps an access history to reduce unnecessary page migrations. Pages are migrated to nodes which access them the most.

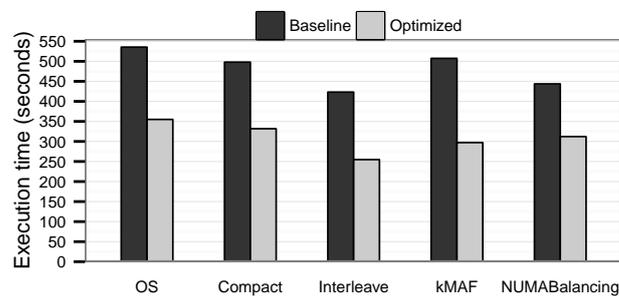


Figure 6.15: Execution time results of the two HashSieve implementations, for different mapping policies.

Figure 6.15 shows the execution time results of HashSieve with different mapping policies. All the values are averages of 3 executions. The results were very stable, with a difference between the maximum and minimum of less than 2% of the total execution time. The code optimizations we conducted resulted in a speedup between 40-70%, depending on the mapping policy. Compared to the OS, the Compact thread mapping only reduces execution time slightly for both versions of HashSieve (less than 7%). This is due to the unstructured memory access pattern and low exclusivity figures of the implementations, as reported before. For the baseline version of HashSieve, which has very low exclusivity, kMAF is only able to improve performance slightly, since both threads and data are moved around based on locality. As the mechanism detects a low page exclusivity, only few actions are taken. Both NUMA Balancing and the Interleave policies reduce execution time substantially (by 17% and 21%, respectively) for the baseline version, since, as shown in Figure 6.14, the implementations incur a high imbalance, which is resolved by these policies.

Our optimized version of HashSieve is substantially faster than the original version, even with the OS mapping. Nevertheless, the optimized version can benefit further from better data mapping. kMAF and NUMA Balancing deliver better results than OS mapping. kMAF delivers about 17% speedup (which compares to $\approx 5\%$ for the baseline version) whereas NUMA Balancing delivers about 11%, in comparison to about 17% for the baseline version. The reason for this is that, as locality became better, kMAF is more effective, but since balancing also became better, there is less room for improvement for the NUMA Balancing policy. Similar to the baseline version, the Interleave policy provides the highest improvements among all policies, reducing execution time by 28%. Overall, the performance was improved by a factor of $2.1\times$, from the baseline version running with the OS mapping to the optimized version running with

the Interleave policy.

6.4.3 Summary

We showed that memory is a key aspect of sieving algorithms. Number one, the algorithms require very high memory usage levels, causing memory to become a bottleneck eventually. Number two, the memory access pattern of the algorithms, and HashSieve in particular, is very irregular and unstructured, which is not ideal for modern computer architectures. In this section, we showed how memory usage can be enhanced in sieving algorithms, using HashSieve as a representative example, both with code optimizations (e.g. with memory pools and manual prefetching) and general improvements on the memory access patterns (e.g. by exploring memory access balancing and mapping policies), when running the algorithms on NUMA machines. This was primarily shown for HashSieve, although the results are extensible to other sieving algorithms and even other classes of algorithms. Although not presented in this dissertation, we introduce a methodology to improve memory access, using similar processes to those described here, in [72].

Conclusions and Outline

"The Church says that the Earth is flat, but I know that it is round. For I have seen the shadow of the earth on the moon and I have more faith in the Shadow than in the Church."

Ferdinand Magellan, Portuguese Navigator and Explorer, 1480-1521.

Today, lattice-based cryptography is one of the most prominent type of quantum-immune cryptosystems. The SVP, the CVP and lattice basis reduction are among the key problems underpinning the security of lattice-based cryptosystems. The safety of these cryptosystems is determined by their specific parameters, which are set based on the hardness of these problems. The common way to determine the actual hardness of these problems, so that appropriate parameters for cryptosystems can be chosen, is to solve them in practice, using the best possible algorithms (i.e. SVP- and CVP-solvers) implemented on high-end computer architectures such as multi-core processors. While considerable progress was made on SVP- and CVP-solvers from a theoretical point of view, their practical performance is commonly not well understood, and so it is not possible to set parameters with high confidence. This thesis aimed to fill this gap in knowledge, by analyzing various important algorithms for lattice-based cryptography, and describing techniques to parallelize and optimize them for modern, parallel computer architectures. In particular, this dissertation covered enumeration algorithms in Chapter 4, lattice basis reduction algorithms in Chapter 5, and sieving algorithms, the focus of the thesis, in Chapter 6.

In Chapter 4, we show scalable implementations of ENUM, based on work-sharing and demand-driven parallel execution models. We implemented the first approach with OpenMP and the second with POSIX threads. We conclude that OpenMP is highly convenient but it inevitably duplicates computation for this particular application. Our demand-driven mechanism does not duplicate computation, and attains linear speedups, and compares well to the OpenMP-based implementation, especially for high thread counts, e.g., 64 threads. This implementation is the fastest published parallel full enumeration SVP-solver to date, as it surpasses the state of the art implementation by Dagdelen et al. [29].

In Chapter 5, we show that data structures and their organization in memory are of major importance in order to achieve good performance levels across various algorithms in lattice-based cryptanalysis (cf. e.g., Section 5.2.3). They allow vectorization of kernels such as inner products, which are common in both lattice reduction and sieving algorithms. Therefore, we presented a vectorized, cache friendly LLL implementation, which is especially efficient on high-dimensional lattices (cf. e.g. Section 5.2.3).

We also show that the parallelization of the BKZ algorithm is easily achieved with the scalable ENUM implementation presented in Chapter 4, as ENUM is the dominating kernel in BKZ for high block sizes.

However, the scalability of BKZ in this scenario is limited, as ENUM only scales well for high block-sizes, with which BKZ becomes intractable. For moderate block-sizes, say 30, BKZ is practical, but ENUM does not scale linearly for many threads as there is not enough work to keep all them busy. We showed how to parallelize ENUM in such a way that the inherent load imbalance of the algorithm is controlled, with parameters that determine the number and shape of tasks. The technique we proposed is applicable to pruned ENUM variants as well, such as those in BKZ 2.0. It should be noted, though, that pruned ENUM calls may have trees more imbalanced than not-pruned ones, and so the parameters of the model we propose must be defined accordingly. In fact, it is an interesting question for future work to determine whether there is a relationship between the bounding function used to prune each ENUM call within BKZ 2.0 and the parameters of our parallelization scheme.

Chapter 6, which focuses on sieving algorithms, comprises most of the contributions of this thesis. The chapter refutes the idea that sieving algorithms do not scale well and are impractical. We started by showing various scalable schemes for multiple sieving algorithms, whose underlying ideas are orthogonal to other classes of algorithms. For instance, we showed that the properties of some sieving algorithms can be relaxed, so that high scalability is attained. In particular, we showed that if we admit loss of vectors in ListSieve from time to time, this can be used to implement a parallel heuristic of ListSieve that scales super-linearly. We then exploited the use of lock-free data structures to present scalable versions of other sieving algorithms, such as GaussSieve. We showed that a lock-free linked list is an effective way to aggregate vectors altogether (as opposed to a split list of vectors, which has been shown to be ineffective before), while permitting parallel thread cooperation on inter-vector reduction. Later, we presented the concept of probable lock-free data structures, i.e. data structures that can be used in parallel codes and are likely lock-free throughout the execution, although threads may spin, if need be. We used such concept to implement parallel versions of HashSieve and LDSieve, which scale linearly even with high thread counts. These implementations are currently the most efficient sieving parallel implementations published in the literature. We also identified and described many memory problems pertaining to sieving algorithms, and methods to mitigate or even correct them. For instance, as sieving algorithms are memory bound, we showed that most sieving codes can profit e.g. from memory and object pools and manual prefetching, among other techniques. On NUMA machines, we showed the impact of many page mapping policies on performance, and how to select one over the others.

Generalization of results. Although the main goal of this thesis is to show the full potential (both in terms of performance and scalability) of the algorithms studied, there are two important generalizations that can be drawn from the set of results that we presented. The first pertains to lattice-based cryptanalysis, as we showed that sieving algorithms are more practical and scalable than previously believed. Given that sieving has lower complexity than enumeration ($2^{\mathcal{O}(n)}$ vs. $2^{\mathcal{O}(n \log n)}$), its complexity should indeed be used for parameter selection in lattice-based cryptosystems, based on our results. In fact, we have found that sieving algorithms are suited for parallel computing on shared-memory systems, if different threads work on a global, tightly coupled list of vectors, even if the properties of the algorithms are somewhat relaxed. We expect that our results with sieving will motivate the community to give them closer consideration.

The second generalization pertains to parallel programming. We argue that many of the techniques we proposed for parallelizing SVP-solvers can be used for other classes of algorithms. Regarding enumeration algorithms, we showed that lower-level handling of parallel tasks with POSIX threads brings

wider flexibility, but at the cost of complex mechanisms that are transparent to the programmer in the OpenMP runtime system. The suitability of each paradigm depends upon the concrete application, and all variables (programming time, performance, etc) should be taken into account when choosing the right paradigm. Regarding sieving algorithms, we believe that both the relaxation of the properties of algorithms and the concept of probable lock-freeness, presented in Section 6.3.4, may be useful for other applications.

We also note two other relevant points when assessing the contributions of this thesis in the realm of SVP-solvers. First, while this thesis was conducted, Random Sampling algorithms became progressively more important. For instance, the RS algorithm by Fukase et al. was published in 2015 and used to consistently break the SVP-Challenge for several dimensions higher than 120 [36]. Later on, a new RS algorithm was reported to be used to break dimensions 134-148, thus currently holding the SVP-Challenge record, but the algorithm is yet to be published. We have not worked with RS-based algorithms, as the work of Fukase et al. reported a parallel version that attained good results. We do note that RS algorithms can also benefit from the work developed in the context of this thesis, as they also include some sort of lattice basis reduction, for which we proposed parallel, efficient models.

Second, it is common to use lattice basis reduction algorithms to solve the SVP-Challenge in high dimensions. In general, lattice basis reduction algorithms do not find a shortest vector. However, the shortest vectors of the lattices in the challenge are not known, and so the challenge accepts solutions that are 5% off the Gaussian heuristic. More often than not, lattice basis reduction algorithms find solutions within this interval. As of July 2016, all entries for lattices in dimension 120 and onwards were solved with either Random Sampling algorithms or lattice basis reduction algorithms (primarily BKZ 2.0). We stress that there is, from a theoretical standpoint, no reason to believe that our models to parallelize ENUM cannot be extended to BKZ 2.0. However, we anticipate that the balancing parameters of our model have to be set based on the load imbalance of the pruned calls inside the algorithm.

Future work. Throughout this thesis, it became clear that most of the algorithms studied are suited for shared-memory parallel systems. In order to solve e.g. the SVP on high-dimensional lattices, one needs to access massive computing capability, which is typically provided by clusters of Symmetric Multi-Processing (SMP) systems. However, the advances in processor design suggest that shared-memory systems will continue to evolve, integrating more cores on the chip, which favours our implementations. A recent example of this is the new Chinese supercomputer Sunway TaihuLight, which leapfrogged the competition in this year's TOP500 list. It is equipped with homegrown SW26010 processors, with 260 cores per chip^{1,2}. Porting these algorithms to clusters of SMP systems is, nevertheless, an interesting and relevant line for future work. This may require a redesign of the algorithms themselves; we have tried to port some of our own implementations to distributed memory systems, but this task has shown to be complex, due to the nature of the algorithms.

As for other future lines of work, we want to extend our optimistic parallelization methods for sieving algorithms to GPUs and other accelerators, and devise heterogeneous CPU+GPU implementations. We also want to investigate bounding functions for pruned versions of enumeration, to include in recent variants of BKZ, such as BKZ 2.0. It will be interesting to assess the performance of our parallel enumeration implementations in such scenarios; we anticipate that dynamic setting for balancing parameters will be needed for optimal performance, and therefore we want to explore the possibility of a connection between these parameters and the bounding functions.

¹<https://www.top500.org/news/china-tops-supercomputer-rankings-with-new-93-petaflop-machine/>

²<https://www.top500.org/lists/2016/06/>

Bibliography

- [1] Erik Agrell, Thomas Eriksson, Alexander Vardy, and Kenneth Zeger. Closest point search in lattices. *IEEE Transactions on Information Theory*, 48(8):2201–2214, 2002.
- [2] M. Ajtai. Generating hard instances of lattice problems (extended abstract). In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 99–108, New York, NY, USA, 1996. ACM.
- [3] Miklós Ajtai. The shortest vector problem in L_2 is NP-hard for randomized reductions (extended abstract). In *STOC*, pages 10–19, 1998.
- [4] Miklós Ajtai and Cynthia Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 284–293, New York, NY, USA, 1997. ACM.
- [5] Miklós Ajtai, Ravi Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *STOC*, pages 601–610, 2001.
- [6] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Mathematical Cryptology*, 9(3):169–203, 2015.
- [7] Yoshinori Aono, Yuntao Wang, Takuya Hayashi, and Tsuyoshi Takagi. Improved progressive BKZ algorithms and their precise cost estimation by sharp simulator. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I*, pages 789–819, 2016.
- [8] Thomas Arnreich. A comprehensive comparison of BKZ implementations on multi-core CPUs. B.S. Thesis, Technische Universitaet Darmstadt, Germany, 2014.
- [9] Sanjeev Arora, László Babai, Jacques Stern, and Z. Sweedyk. The hardness of approximate optima in lattices, codes, and systems of linear equations. *J. Comput. Syst. Sci.*, 54(2):317–331, 1997.
- [10] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *SODA*, 2016.
- [11] Anja Becker, Nicolas Gama, and Antoine Joux. A sieve algorithm based on overlattices. In *ANTS*, pages 49–70, 2014.

- [12] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. In *UCSD CSE 207 Course Notes*, page 207, 2005.
- [13] E. 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.
- [14] Dan Boneh. Twenty Years of Attacks on the RSA Cryptosystem. *NOTICES OF THE AMS*, 46:203–213, 1999.
- [15] Joppe Bos, Michael Naehrig, and Joop van de Pol. Sieving for shortest vectors in ideal lattices: a practical perspective. Cryptology ePrint Archive, Report 2014/880, 2014.
- [16] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. On the Security of 1024-bit RSA and 160-bit Elliptic Curve Cryptography. Cryptology ePrint Archive, Report 2009/389, 2009. <http://eprint.iacr.org/>.
- [17] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery. Solving a 112-bit prime elliptic curve discrete logarithm problem on game consoles using sloppy reduction. *IJACT*, 2(3):212–228, 2012.
- [18] Johannes Buchmann and Christoph Ludwig. *Algorithmic Number Theory: 7th International Symposium, ANTS-VII, Berlin, Germany, July 23-28, 2006. Proceedings*, chapter Practical Lattice Basis Sampling Reduction, pages 222–237. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.
- [19] Narasimham Challa and Jayaram Pradhan. Summary performance analysis of public key cryptographic systems RSA and NTRU. *IJCSNS International Journal of Computer Science and Network Security*, 7(8), 2007.
- [20] Yuanmi Chen. *Reduction de reseau et securite concrete du chiffrement completement homomorphe*. PhD thesis, Université Paris Diderot, Paris, France, 2015.
- [21] Yuanmi Chen and Phong Q. Nguyen. *Advances in Cryptology – ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, chapter BKZ 2.0: Better Lattice Security Estimates, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [22] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.
- [23] J. H. Conway, N. J. A. Sloane, and E. Bannai. *Sphere-packings, Lattices, and Groups*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.
- [24] Jonathan Corbet. Toward better NUMA scheduling, 2012.
- [25] Fabio Correia. Assessing the hardness of SVP algorithms in the presence of CPUs and GPUs. Master’s thesis, University of Minho, Braga, Portugal, 2014.

- [26] Fabio Correia, Artur Mariano, Alberto Proença, Christian H. Bischof, and Erik Agrell. Parallel Improved Schnorr-Euchner Enumeration SE++ for the CVP and SVP. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 596–603, 2016.
- [27] James Cowie, Bruce Dodson, R. Marije Elkenbracht-Huizing, Arjen K. Lenstra, Peter L. Montgomery, and Jörg Zayer. *Advances in Cryptology — ASIACRYPT '96: International Conference on the Theory and Applications of Cryptology and Information Security Kyongju, Korea, November 3–7, 1996 Proceedings*, chapter A World Wide Number Field Sieve factoring record: On to 512 bits, pages 382–394. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.
- [28] Ronald Cramer and Victor Shoup. Signature schemes based on the strong RSA assumption. *ACM Trans. Inf. Syst. Secur.*, 3(3):161–185, August 2000.
- [29] Özgür Dagdelen and Michael Schneider. *Euro-Par 2010 - Parallel Processing: 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II*, chapter Parallel Enumeration of Shortest Lattice Vectors, pages 211–222. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [30] Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Hei. kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 277–288, 2014.
- [31] Matthias Diener, Eduardo H. M. Cruz, Larcio L. Pilla, Fabrice Dupros, and Philippe O. A. Navaux. Characterizing Communication and Page Usage of Parallel Applications for Thread and Data Mapping. *Performance Evaluation*, 88-89(June):18–36, 2015.
- [32] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006.
- [33] John Mellor-Crummey et al. Improving memory hierarchy performance for irregular applications using data and computation reorderings. In *International Journal of Parallel Programming*, pages 425–433, 2001.
- [34] 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.
- [35] Robert Fitzpatrick, Christian Bischof, Johannes Buchmann, Özgür Dagdelen, Florian Göpfert, Artur Mariano, and Bo-Yin Yang. Tuning GaussSieve for speed. In *LATINCRYPT*, pages 284–301, 2014.
- [36] Masaharu Fukase and Kenji Kashiwabara. An accelerated algorithm for solving SVP based on statistical analysis. *JIP*, 23(1):67–80, 2015.
- [37] V. I. Levenshtein G. A. Kabatiansky. On bounds for packings on a sphere and in space. *Problems Inform. Transmission*, 14(1):3–25, 1978.
- [38] Nicolas Gama, Phong Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *EUROCRYPT*, pages 257–278, 2010.

- [39] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178, 2009.
- [40] Arash Ghasemmehdi and Erik Agrell. Faster recursions in sphere decoding. *IEEE Trans. Information Theory*, 57(6):3530–3536, 2011.
- [41] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '97, pages 112–131, London, UK, UK, 1997. Springer-Verlag.
- [42] Oded Goldreich, Daniele Micciancio, S. Safra, and J-P. Seifert. Approximating shortest lattice vectors is not harder than approximating closest lattice vectors. 71(2):55–61, 1999.
- [43] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Algorithms for the shortest and closest lattice vector problems. In *IWCC*, pages 159–190, 2011.
- [44] Guillaume Hanrot and Damien Stehlé. *Advances in Cryptology - CRYPTO 2007: 27th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2007. Proceedings*, chapter Improved Analysis of Kannan's Shortest Lattice Vector Algorithm, pages 170–186. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.
- [45] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, DISC '01, pages 300–314, Berlin, Heidelberg, 2001. Springer-Verlag.
- [46] Bettina Helfrich. Algorithms to Construct Minkowski Reduced an Hermite Reduced Lattice Bases. *Theor. Comput. Sci.*, 41:125–139, 1985.
- [47] Ryan Henry and Ian Goldberg. Solving discrete logarithms in smooth-order groups with cuda. In *Workshop Record of SHARCS 2012: Special-purpose Hardware for Attacking Cryptographic Systems*, pages 101–118. 2012.
- [48] Jens Hermans, Michael Schneider, Johannes Buchmann, Frederik Vercauteren, and Bart Preneel. Parallel shortest lattice vector enumeration on graphics cards. In *AFRICACRYPT*, pages 52–68, 2010.
- [49] C. Hermite. Extraits de lettres de m. hermite à m. jacobi sur differents objets de la thÉorie des nombres, deuxième lettre. *Reine Angew. Math.*, 40:279–290, 1850.
- [50] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. *Algorithmic Number Theory: Third International Symposiun, ANTS-III Portland, Oregon, USA, June 21–25, 1998 Proceedings*, chapter NTRU: A ring-based public key cryptosystem, pages 267–288. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.
- [51] Tsukasa Ishiguro, Shinsaku Kiyomoto, Yutaka Miyake, and Tsuyoshi Takagi. Parallel Gauss Sieve algorithm: Solving the SVP challenge over a 128-dimensional ideal lattice. In *PKC'14*, pages 411–428, 2014.
- [52] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Math. Oper. Res.*, 12(3):415–440, August 1987.

- [53] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.
- [54] Philip Klein. Finding the closest lattice vector when it's unusually close. In *SODA*, SODA '00, pages 937–941, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.
- [55] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen K. Lenstra, Emmanuel Thomé, Joppe W. Bos, Pierrick Gaudry, Alexander Kruppa, Peter L. Montgomery, Dag Arne Osvik, Herman Te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA Modulus. In *Proceedings of the 30th Annual Conference on Advances in Cryptology, CRYPTO'10*, pages 333–350, Berlin, Heidelberg, 2010. Springer-Verlag.
- [56] Neal Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.
- [57] Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.
- [58] Po-Chun Kuo, Michael Schneider, Özgür Dagdelen, Jan Reichelt, Johannes Buchmann, Chen-Mou Cheng, and Bo-Yin Yang. *Cryptographic Hardware and Embedded Systems – CHES 2011: 13th International Workshop, Nara, Japan, September 28 – October 1, 2011. Proceedings*, chapter Extreme Enumeration on GPU and in Clouds, pages 176–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [59] Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In *CRYPTO*, pages 3–22, 2015.
- [60] Thijs Laarhoven, Michele Mosca, and Joop van de Pol. Solving the shortest vector problem in lattices faster using quantum search. *CoRR*, abs/1301.6176, 2013.
- [61] Thijs Laarhoven, Joop van de Pol, and Benne de Weger. Solving hard lattice problems and the security of lattice-based cryptosystems. *Cryptology ePrint Archive*, Report 2012/533, 2012.
- [62] Thijs Laarhoven and Benne Weger. *Progress in Cryptology – LATINCRYPT 2015: 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, chapter Faster Sieving for Shortest Lattice Vectors Using Spherical Locality-Sensitive Hashing, pages 101–118. Springer International Publishing, Cham, 2015.
- [63] J. C. Lagarias and A. M. Odlyzko. Solving low density subset sum problems. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 1–10, Nov 1983.
- [64] J. C. Lagarias and A. M. Odlyzko. Solving low-density subset sum problems. *J. ACM*, 32(1):229–246, January 1985.
- [65] A.K. Lenstra, H.W.jun. Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Math. Ann.*, 261:515–534, 1982.

- [66] Richard Lindner and Chris Peikert. *Topics in Cryptology – CT-RSA 2011: The Cryptographers’ Track at the RSA Conference 2011, San Francisco, CA, USA, February 14-18, 2011. Proceedings*, chapter Better Key Sizes (and Attacks) for LWE-Based Encryption, pages 319–339. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [67] 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.
- [68] Vadim Lyubashevsky. *Public Key Cryptography – PKC 2008: 11th International Workshop on Practice and Theory in Public-Key Cryptography, Barcelona, Spain, March 9-12, 2008. Proceedings*, chapter Lattice-Based Identification Schemes Secure Under Active Attacks, pages 162–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
- [69] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. *Advances in Cryptology – EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 – June 3, 2010. Proceedings*, chapter On Ideal Lattices and Learning with Errors over Rings, pages 1–23. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [70] Artur Mariano and Christian H. Bischof. Enhancing the Scalability and Memory Usage of Hashsieve on Multi-core CPUs. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 545–552, 2016.
- [71] Artur Mariano, Fabio Correia, and Christian Bischof. A vectorized, cache efficient LLL implementation. In *12th International Meeting on High Performance Computing for Computational Science, Porto, Portugal, June 28th to 30th, 2016*, 2016.
- [72] Artur Mariano, Matthias Diener, Christian H. Bischof, and Philippe O. A. Navaux. Analyzing and Improving Memory Access Patterns of Large Irregular Applications on NUMA Machines. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 382–387, 2016.
- [73] Artur Mariano, Thijs Laarhoven, and Christian Bischof. Parallel (probable) lock-free hash sieve: A practical sieving algorithm for the SVP. In *44th International Conference on Parallel Processing (ICPP)*, pages 590–599, September 2015.
- [74] Artur Mariano, Thijs Laarhoven, and Christian H. Bischof. A Parallel Variant of LDSieve for the SVP on Lattices. In *Technical report*, 2016.
- [75] Artur Mariano, Özgür Dagdelen, and Christian Bischof. A comprehensive empirical comparison of parallel ListSieve and GaussSieve. In *APCI&E*, 2014.
- [76] Artur Mariano, Shahar Timnat, and Christian Bischof. Lock-free GaussSieve for linear speedups in parallel high performance SVP calculation. *SBAC-PAD’14*, 2014.
- [77] Paulo Martins, Artur Mariano, and Leonel Sousa. A survey on fully homomorphic encryption: an engineering perspective. *Submitted.*, 2016.

- [78] Nicholas McDonald. Past, present and future methods of cryptography and data encryption. Research Review. University of Utah.
- [79] Daniele Micciancio. Efficient reductions among lattice problems. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, pages 84–93, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.
- [80] Daniele Micciancio and Shafi Goldwasser. *Complexity of Lattice Problems: a cryptographic perspective*, volume 671 of *The Kluwer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Boston, Massachusetts, March 2002.
- [81] Daniele Micciancio and Oded Regev. *Post-Quantum Cryptography*, chapter Lattice-based Cryptography, pages 147–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
- [82] Daniele Micciancio and Panagiotis Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations. *Electronic Colloquium on Computational Complexity (ECCC)*, 17:14, 2010.
- [83] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *SODA*, pages 1468–1480, 2010.
- [84] Daniele Micciancio and Michael Walter. Fast lattice point enumeration with minimal overhead. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '15, pages 276–294. SIAM, 2015.
- [85] Andrea Miele, Joppe W. Bos, Thorsten Kleinjung, and Arjen K. Lenstra. Cofactorization on graphics processing units. In *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, pages 335–352, 2014.
- [86] Benjamin Milde and Michael Schneider. A parallel implementation of GaussSieve for the shortest vector problem in lattices. In *PaCT*, pages 452–458, 2011.
- [87] Victor S. Miller. *Advances in Cryptology — CRYPTO '85 Proceedings*, chapter Use of Elliptic Curves in Cryptography, pages 417–426. Springer Berlin Heidelberg, Berlin, Heidelberg, 1986.
- [88] Phong Nguyen. *Advances in Cryptology — CRYPTO' 99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings*, chapter Cryptanalysis of the Goldreich-Goldwasser-Halevi Cryptosystem from Crypto '97, pages 288–304. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.
- [89] Phong Nguyen and Jacques Stern. *Advances in Cryptology — CRYPTO '98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings*, chapter Cryptanalysis of the Ajtai-Dwork cryptosystem, pages 223–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.
- [90] Phong Q. Nguyen and Damien Stehlé. An LLL algorithm with quadratic complexity. *SIAM J. Comput.*, 39(3):874–903, 2009.

- [91] Phong Q. Nguyen and Brigitte Valle. *The LLL Algorithm: Survey and Applications*. Springer Publishing Company, Incorporated, 1st edition, 2009.
- [92] Phong Q. Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptology*, 2(2):181–207, 2008.
- [93] Phong Q. Nguên and Damien Stehlé. Floating-point LLL revisited. In Ronald Cramer, editor, *Advances in Cryptology – EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 215–233. Springer Berlin Heidelberg, 2005.
- [94] A. Odlyzko. Cryptanalytic attacks on the multiplicative knapsack cryptosystem and on shamir’s fast signature scheme. *IEEE Trans. Inf. Theor.*, 30(4):594–601, 1984.
- [95] A M Odlyzko. Discrete logarithms in finite fields and their cryptographic significance. In *Proc. Of the EUROCRYPT 84 Workshop on Advances in Cryptology: Theory and Application of Cryptographic Techniques*, pages 224–314, New York, NY, USA, 1985. Springer-Verlag New York, Inc.
- [96] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.
- [97] Chris Peikert. A decade of lattice cryptography. *IACR Cryptology ePrint Archive*, 2015:939, 2015.
- [98] Thomas Plantard and Michael Schneider. Creating a challenge for ideal lattices. *Cryptology ePrint Archive*, Report 2013/039, 2013. <http://eprint.iacr.org/>.
- [99] 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.
- [100] J. M. Pollard. Monte Carlo methods for index computation (mod p). 32(143):918–924, July 1978.
- [101] Xavier Pujol and Damien Stehlé. Solving the shortest lattice vector problem in time $22.465n$. *IACR Cryptology ePrint Archive*, 2009:605, 2009.
- [102] S. Qiao. A Jacobi Method for Lattice Basis Reduction. In *Engineering and Technology (S-CET), 2012 Spring Congress on*, pages 1–4, May 2012.
- [103] O. Regev. The learning with errors problem (invited survey). In *Proc. IEEE Conference on Computational Complexity*, pages 191–204, 2010.
- [104] Oded Regev. New lattice-based cryptographic constructions. *J. ACM*, 51(6):899–942, November 2004.
- [105] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing*, STOC ’05, pages 84–93, New York, NY, USA, 2005. ACM.
- [106] R L Rivest, L Adleman, and M L Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, Academia Press, pages 169–179, 1978.

- [107] Ronald L. Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.
- [108] Michael Schneider. Sieving for short vectors in ideal lattices. In *AFRICACRYPT*, pages 375–391, 2013.
- [109] Michael Schneider and Norman Göttert. *Cryptographic Hardware and Embedded Systems – CHES 2011: 13th International Workshop, Nara, Japan, September 28 – October 1, 2011. Proceedings*, chapter Random Sampling for Short Lattice Vectors on Graphics Cards, pages 160–175. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [110] C. P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theor. Comput. Sci.*, 53(2-3):201–224, August 1987.
- [111] C. P. Schnorr and H. H. Hörner. *Advances in Cryptology — EUROCRYPT '95: International Conference on the Theory and Application of Cryptographic Techniques Saint-Malo, France, May 21–25, 1995 Proceedings*, chapter Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction, pages 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995.
- [112] Claus Peter Schnorr. Lattice reduction by random sampling and birthday methods. In *In Proc. STACS 2003*, Eds. H. Alt and M. Habib, LNCS 2607, pages 145–156. Springer, 2003.
- [113] Claus Peter Schnorr. *The LLL Algorithm: Survey and Applications*, chapter Progress on LLL and Lattice Reduction, pages 145–178. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [114] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(2–3):181–199, 1994.
- [115] C.P. Schnorr. A more efficient algorithm for lattice basis reduction. In Laurent Kott, editor, *Automata, Languages and Programming*, volume 226 of *Lecture Notes in Computer Science*, pages 359–369. 1986.
- [116] P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, SFCS '94, pages 124–134, Washington, DC, USA, 1994. IEEE Computer Society.
- [117] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997.
- [118] Denis Simon. *Selected Applications of LLL in Number Theory*, pages 265–282. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [119] Simon Singh. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. London: Fourth Estate, 1999.
- [120] Nigel Smart. *Cryptography: An Introduction*. McGraw-Hill Education, 2003.
- [121] Damien Stehlé. *The LLL Algorithm: Survey and Applications*, chapter Floating-Point LLL: Theoretical and Practical Aspects, pages 179–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

- [122] Damien Stehlé and Ron Steinfeld. *Advances in Cryptology – EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, chapter Making NTRU as Secure as Worst-Case Problems over Ideal Lattices, pages 27–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.
- [123] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. *Advances in Cryptology – ASIACRYPT 2009: 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, chapter Efficient Public Key Encryption Based on Ideal Lattices, pages 617–635. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.
- [124] Laarhoven Thijs. *Search problems in cryptography: From fingerprinting to lattice sieving*. PhD thesis, Technische Universiteit Eindhoven, The Netherlands, 2016.
- [125] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.
- [126] Brigitte Vallée and Antonio Vera. *The LLL Algorithm: Survey and Applications*, chapter Probabilistic Analyses of Lattice Reduction Algorithms, pages 71–143. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [127] Joop van de Pol. Lattice-based cryptography. Master’s thesis, Technische Universiteit Eindhoven, The Netherlands, 2015.
- [128] Michael Walter. *Information Theoretic Security: 8th International Conference, ICITS 2015, Lugano, Switzerland, May 2-5, 2015. Proceedings*, chapter Lattice Point Enumeration on Block Reduced Bases, pages 269–282. Springer International Publishing, Cham, 2015.
- [129] Xiaoyun Wang, Mingjie Liu, Chengliang Tian, and Jingguo Bi. Improved Nguyen-Vidick Heuristic Sieve Algorithm for Shortest Vector Problem. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS ’11*, pages 1–9, New York, NY, USA, 2011. ACM.
- [130] Erich Wenger and Paul Wolfger. *Selected Areas in Cryptography – SAC 2014: 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers*, chapter Solving the Discrete Logarithm of a 113-Bit Koblitz Curve with an FPGA Cluster, pages 363–379. Springer International Publishing, Cham, 2014.
- [131] S. Wu and C. Tan. A high security framework for SMS. In *Biomedical Engineering and Informatics, 2009. BMEI ’09. 2nd International Conference on*, pages 1–6, Oct 2009.
- [132] Peiliang Xu. Experimental quality evaluation of lattice basis reduction methods for decorrelating low-dimensional integer least squares problems. *EURASIP J. Adv. Sig. Proc.*, 2013:137, 2013.
- [133] Feng Zhang, Yanbin Pan, and Gengran Hu. A three-level sieve algorithm for the shortest vector problem. *IACR Cryptology ePrint Archive*, 2013:536, 2013.

- [134] Zheng Zhang and Josep Torrellas. Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching. In *Proceedings of the 22nd Annual ISCA*, pages 188–200, 1995.