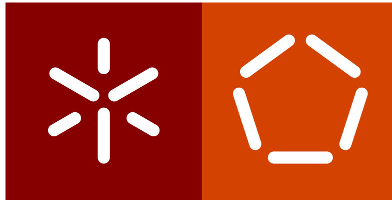
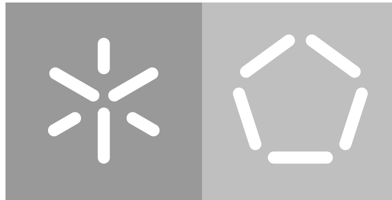


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Hélder José Alves Gonçalves

Towards an efficient lattice basis reduction implementation

October 2016



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Hélder José Alves Gonçalves

Towards an efficient lattice basis reduction implementation

Master dissertation
Master Degree in Computer Science

Supervisor: Alberto José Proença
External Advisor: Artur Miguel Matos Mariano

October 2016

ACKNOWLEDGEMENTS

Quero agradecer ao meu orientador, Alberto Proença, por todo o esforço despendido para a realização desta dissertação, em que o seu acompanhamento constante e rigor exigido foram imprescindíveis. Também quero agradecer ao meu orientador externo, Artur Mariano, pela oportunidade de efectuar um estágio no âmbito da dissertação na Alemanha.

I would also like to thank the Institute for Scientific Computing for receiving me in Darmstadt. A special thanks to Fábio Correia for every discussions on the subject or not, during my internship. I also would like to thank Professor Christian Bischof, Florian Göpfert and Dominique Metz.

I would also like to acknowledge Professor Shizhang Qiao and Rui Ralha for all suggestions.

Também quero agradecer aos meus colegas de curso e amigos Fábio Gomes e Duarte Duarte pelo ajuda dada na revisão da escrita da dissertação e por todas as conversas fantásticas que se sucederam no Skype durante este último ano.

Um especial obrigado à Márcia Couto por todo o suporte dado durante este último ano.

Finamente mas não menos importante, quero agradecer aos meus pais por todo o suporte, esforço e dedicação prestados não só neste último ano mas por todos os anos que me trouxeram a este ponto.

ABSTRACT

The security of most digital systems is under serious threats due to major technology breakthroughs we are experienced in nowadays. Lattice-based cryptosystems are one of the most promising post-quantum types of cryptography, since it is believed to be secure against quantum computer attacks. Their security is based on the hardness of the Shortest Vector Problem and Closest Vector Problem.

Lattice basis reduction algorithms are used in several fields, such as lattice-based cryptography and signal processing. They aim to make the problem easier to solve by obtaining shorter and more orthogonal basis. Some case studies work with numbers with hundreds of digits to ensure harder problems, which require Multiple Precision (MP) arithmetic. This dissertation presents a novel integer representation for MP arithmetic and the algorithms for the associated operations, MpIM. It also compares these implementations with other libraries, such as GNU Multiple Precision Arithmetic Library, where our experimental results display a similar performance and for some operations better performances.

This dissertation also describes a novel lattice basis reduction module, LattBRed, which included a novel efficient implementation of the Qiao's Jacobi method, a Lenstra-Lenstra-Lovász (LLL) algorithm and associated parallel implementations, a parallel variant of the Block Korkine-Zolotarev (BKZ) algorithm and its implementation and MP versions of the the Qiao's Jacobi method, the LLL and BKZ algorithms.

Experimental performances measurements with the set of implemented modifications of the Qiao's Jacobi method show some performance improvements and some degradations but speedups greater than 100 in Ajtai-type bases.

RESUMO

Atualmente existe um grande avanço tecnológico que poderá colocar em causa a segurança da maioria dos sistemas informáticos. Sistemas criptográficos baseados em reticulados são um dos mais promissores tipos de criptografia pós-quântica, uma vez que se acredita que estes sistemas são seguros contra possíveis ataques de computadores quânticos. A segurança destes sistemas está baseada na dificuldade de resolver o problema do vetor mais curto e o problema do vetor mais próximo.

Algoritmos de redução de bases de reticulados são usados em muitos campos científicos, tais como criptografia baseada em reticulados. O seu principal objetivo é tornar o problema mais fácil de resolver, tornando a base do reticulado mais curta e ortogonal. Alguns casos de estudo requerem o uso de números com centenas de dígitos para garantir problemas mais difíceis. Portanto, é importante o uso de módulos de precisão múltipla. Esta dissertação apresenta uma nova representação de inteiros para aritmética de precisão múltipla e todas as respetivas funções de um módulo, *'MpIM'*. Também comparamos as nossas implementações com outras bibliotecas de precisão múltipla, tais como *'GNU Multiple Precision Arithmetic Library'*, em que obtivemos desempenhos semelhantes e em alguns casos melhores.

A dissertação também apresenta um novo módulo para a redução de bases de reticulados, *'MpIM'*, que inclui uma nova e eficiente implementação do *'Qiao's Jacobi method'*, o algoritmo *'Lenstra-Lenstra-Lovász'* (LLL) e respectiva implementação paralela, uma variante paralela do algoritmo *'Block Korkine-Zolotarev'* (BKZ) e a sua versão sequencial e versões the precisão múltipla do *'Qiao's Jacobi method'*, LLL e BKZ.

Trabalhos experimentais mostraram que a versão do *'Qiao's Jacobi method'* que implementa todas as modificações sugeridas mostra ganhos e degradações de desempenho, contudo com aumentos de desempenho superiores a 100 vezes em bases *'Ajtai-type'*.

CONTENTS

1 INTRODUCTION 2
1.1 Motivation 4
1.2 Contribution 4
1.3 Roadmap 4
2 BACKGROUND AND SETUP 6
2.1 Multiple precision 7
2.1.1 Current libraries 7
2.1.2 Integer Representation 9
2.1.3 Addition and Subtraction 11
2.1.4 Multiplication 12
2.1.5 Division 17
2.1.6 Newton's method 19
2.1.7 Hensel's division 20
2.2 Lattice basis reduction 21
2.2.1 Basic Concepts 22
2.2.2 Lenstra–Lenstra–Lovász 24
2.2.3 Hermite-Korkine-Zolotarev 26
2.2.4 Block-Korkine-Zolotarev 26
2.2.5 Qiao's Jacobi method 28
2.2.6 Measuring Basis Quality 30
2.3 Experimental environment 31
2.3.1 Non-Uniform Memory Access 31
2.3.2 Vectorization 33
2.3.3 Methodologies 33
3 THE MULTIPLE PRECISION INTEGER MODULE 35
3.1 Addition and Subtraction 35
3.1.1 Addition Vectorization 36
3.1.2 Increment and Decrement 36
3.2 Multiplication 37
3.2.1 Long multiplication 37
3.2.2 Karatsuba 38
3.3 Division 39
3.4 Other Functions 39
3.4.1 Logical Shifts 39

3.4.2 And/Or/Xor 42
3.4.3 Pseudo-Random Number Generator 42
3.4.4 Compare 42
3.5 Evaluation Results 43
4 THE QIAO'S JACOBI METHOD 48
4.0.1 Vectorization 50
4.0.2 Evaluation Results 51
4.1 Parallel Version 54
4.1.1 Evaluation Results 56
4.2 Basis Quality Assessment 57
5 BKZ, LLL AND QIAO'S JACOBI METHOD 60
5.1 Towards parallel approaches 60
5.1.1 Parallel LLL algorithm 60
5.1.2 Parallel BKZ algorithm 62
5.2 BKZ w/ Qiao's Jacobi method 63
5.3 Reducing L-reduced bases 63
6 CONCLUSIONS & FUTURE WORK 66

LIST OF FIGURES

Figure 1 SVP panorama in three layers 6
Figure 2 Binary representation of a large number with 3 *limbs*. 10
Figure 3 Addition with a carry digit in a large number with 2 *limbs*. 11
Figure 4 The best algorithm to multiply two numbers of x and y *limbs*. bc is long multiplication, 22 is Karatsuba's algorithm and 33, 32, 44 and 42 are Toom variants (from [Brent and Zimmermann (2010)]). 13
Figure 5 Long multiplication algorithm (from Intel documentation). 14
Figure 6 Multiplication step (from Intel documentation). 14
Figure 7 Lattice reduction in two dimensions: the black vectors are the given basis for the lattice, the red vectors are the reduced basis (from Wikipedia). 21
Figure 8 The first two steps of the Gram–Schmidt orthogonalization (from Wikipedia). 23
Figure 9 Examples of GM matrices. 24
Figure 10 Chess tournament with $n = 8$ (from [Jeremic and Qiao (2014)]). 29
Figure 11 Shared memory system (from Google Images). 32
Figure 12 One possible architecture of a NUMA system (from Advanced Architectures slides). 32
Figure 13 Scalar implementation vs vector implementation (from Google Images). 33
Figure 14 Simple logical right shift with the insertion of a zero on the left. 40
Figure 15 Simple logical left shift with the insertion of a zero on the right. 40
Figure 16 Right shift of 2 in a 3-limb large number. 40
Figure 17 Left shift of 2 in a 3-limb large number. 41
Figure 18 Comparison between the 5 addition implementations of the MpIM. 43
Figure 19 Comparison of MpIM's addition to other libraries. 43
Figure 20 Comparison of MpIM's subtraction to other libraries. 44
Figure 21 Comparison between the long multiplication and the Karatsuba implementations of the MpIM. 44
Figure 22 Comparison of MpIM's multiplication to other libraries. 44
Figure 23 Comparison of MpIM's division to other libraries. 45
Figure 24 Comparison of MpIM's right shift to other libraries. 46
Figure 25 Comparison of MpIM's left shift to other libraries. 46

Figure 26 Comparison of MpIM's *'or'* function to other libraries. 46
Figure 27 Comparison of MpIM's *'and'* function to other libraries. 46
Figure 28 Comparison of MpIM's *'xor'* function to other libraries. 47
Figure 29 Execution times of sequential LLL_XD and Qiao's Jacobi method in GM bases. 52
Figure 30 Execution times of sequential LLL_FP and Qiao algorithm in Ajtai-type bases. 53
Figure 31 Number of necessary sweeps to converge to a solution in Ajtai-type bases. 53
Figure 32 Speedups comparison between sequential LLL and Qiao's Jacobi method implementations in Ajtai-type bases. 54
Figure 33 Number of necessary sweeps to converge to a solution in GM bases. 55
Figure 34 Execution times of first parallel approach in GM bases. 57
Figure 35 Execution times of second parallel approach in GM bases. 57
Figure 36 Hermite factor of output basis from LLL algorithm and Qiao's Jacobi method in Ajtai-type bases. 59
Figure 37 Average of the norms of the output basis from LLL algorithm and Qiao's Jacobi method in Ajtai-type bases. 59
Figure 38 Sequence of the GS norms from LLL algorithm and Qiao's Jacobi method in Ajtai-type bases. 59
Figure 39 Last GS norms from LLL algorithm and Qiao's Jacobi method in Ajtai-type bases. 59
Figure 40 Execution times of the Qiao's Jacobi Method, the LLL and BKZ algorithms. 63
Figure 41 Hermite factor of output basis from LLL and BKZ algorithms and Qiao's Jacobi method. 64
Figure 42 Average of the norms of output basis from LLL and BKZ algorithms and Qiao's Jacobi method. 64
Figure 43 Sequence of the GS norms of the output bases from the LLL and BKZ algorithms and the Qiao's Jacobi method. 65
Figure 44 Last GS norms of output basis from LLL and BKZ algorithms and Qiao's Jacobi method. 65

LIST OF ALGORITHMS

2.1 Integer Addition, presented in [Brent and Zimmermann (2010)]. 12
2.2 Long Multiplication, presented in [Brent and Zimmermann (2010)]. 14
2.3 Karatsuba's Algorithm, presented in [Brent and Zimmermann (2010)]. 15
2.4 Toom-Cook 3-Way Algorithm, presented in [Brent and Zimmermann (2010)]. 16
2.5 Long Division, presented in [Brent and Zimmermann (2010)]. 18
2.6 Long division (binary version), from Wikipedia. 19
2.7 Division By a *Limb*, presented in [Brent and Zimmermann (2010)]. 20
2.8 LLL algorithm, presented in [Nguyen and Stehlé (2006)]. 25
2.9 BKZ algorithm, presented in [Chen and Nguyen (2011)]. 27
2.10 Qiao's Jacobi Method, presented in [Qiao (2012)]. 29

3.1 Integer Increment 37

4.1 Proposed Qiao's Jacobi method 49

5.1 Gram-Schmidt process for k 61

1

INTRODUCTION

For years the cryptography community has been searching for more resistant cryptosystems. However, only in last decades there have been an intensive search for cryptosystems that would be resistant against quantum computers attacks. This necessity is explained by the vulnerability of the current popular cryptosystems, whose security relies on (i) the integer factorization problem, (ii) the discrete logarithm problem or (iii) the elliptic-curve discrete logarithm problem. Unfortunately, these three hard mathematical problems are no longer hard to solve on a sufficiently large quantum computer running Shor's algorithm [Shor (1997), Bernstein (2009)].

Nowadays, lattice-based cryptosystems are one of the most promising post-quantum types of cryptography, due to its inherent computational hardness and fully-homomorphic properties. Lattices are rich algebraic structures that have many applications in computer science, namely integer programming [Kannan (1983)], communication theory [Agrell et al. (2002), Nguyen (2010)] and number theory [Cassels (2012), Siegel (2013)].

The security of these cryptographic techniques is based on very strong security proofs based on the hardness of worst-case problems. Thus, breaking a cryptographic construction is probably at least as hard as solving several lattice problems in the worst-case.

Most current computer architectures support operations between numbers with up to 64 bits of precision. However, there are cases in cryptography where numbers with hundreds of digits (that cannot be represented as primitive data types) are required to ensure harder problems. Therefore, it is important to resort to Multiple Precision (MP) arithmetic to solve this kind of problems.

The bold face is used to represent vectors and matrices in this dissertation, where vectors are in lower-case and matrices are in upper-case, e.g., vector $\mathbf{v}$ and matrix $\mathbf{M}$. The transpose of a matrix is given by $\mathbf{M}^T$ and the dot product of two vectors $\mathbf{v}$ and $\mathbf{p}$ is denoted by $\langle \mathbf{v}, \mathbf{p} \rangle$. Finally, $\lceil a \rfloor$ rounds the value a to the nearest integer number and $|a|$ gives the absolute value of a.

Lattices are simple algebraic structures based on familiar concepts to any user with basic training in algebra. The conceptual simplicity of these cryptographic techniques is associated with simple matrix computations. A lattice $\mathcal{L}$ in $\mathbb{R}^n$ is generated for all possible linear

combinations with integer coefficients of any basis in $\mathbb{R}^n$, where a basis $\mathbf{B}$ is a set of linear independent vectors $(\mathbf{b}_1, ..., \mathbf{b}_n)$ and $\mathbf{Z}$ are all possible linear combinations, given by:

$$\mathcal{L} = \mathbf{BZ} = \sum_{i=0}^{n} \mathbf{b}_i \mathbf{z}_i, \ \mathbf{z}_i \in \mathbb{Z} \tag{1}$$

Take a lattice $\mathcal{L}$ embedded in a metric vector space $\mathcal{A}$. Since $\mathcal{L}$ is contained in $\mathcal{A}$, there is the notion of *size* $\|\mathbf{x}\|$ and the notion of *distance* $\|\mathbf{x} - \mathbf{z}\|$, where $\mathbf{x}, \ \mathbf{z} \in \mathcal{L}$. These notions are enough to define two basic problems in lattices.

The Shortest Vector Problem (SVP) [Hanrot et al. (2011b)] can be informally defined as the search for the shortest vector of a given lattice $\mathcal{L}$ and formally defined as follows: given a basis $\mathbf{B}$ for a lattice $\mathcal{L} = \mathcal{L}(\mathbf{B})$, find a vector $\mathbf{v} \in \mathcal{L}$ such that $\|\mathbf{v}\| = \lambda_1(\mathcal{L})$, where the norm of the shortest vector of the lattice $\mathcal{L}$ is given by $\lambda_1(\mathcal{L})$. In its approximated version (α-SVP), the goal is to search for that vector, this time multiplied by a small α factor[1,2]. On the other hand, the Closest Vector Problem (CVP) is defined as follows: given a basis $\mathbf{B}$ for a lattice $\mathcal{L} = \mathcal{L}(\mathbf{B})$ and a vector $\mathbf{x} \in \mathbb{R}^n$, find a vector $\mathbf{v} \in \mathcal{L}$ such that $\|\mathbf{x} - \mathbf{v}\|$ is minimal. The CVP and SVP problems are closely related.

The efficiency of several classes of algorithms that solve the SVP, such as enumeration, sieving and random sampling algorithms, is inherently connected with the quality of the input basis. Therefore, the development of new algorithms and the proposal of implementations that improve the quality of a basis is imperative.

Nowadays, lattice enumeration algorithms are one of the main techniques to solve hard lattice problems such as SVP. A basic enumeration consists on an exhaustive search for the best combination of basis vectors among all others, leading to a run in exponential time executions.

In order to have a polynomial complexity algorithm we have to limit the algorithm specification to do not necessarily require the shortest vector of the lattice but only a reduced basis. It is here that lattice basis reduction algorithms play an important role, where its goal is to transform a given basis $\mathbf{B}$ of a lattice $\mathcal{L}$ into a close to orthogonal and shorter basis such that $\mathcal{L}$ remain the same. Since the reduced basis is shorter and more orthogonal, the SVP-solvers are capable of solve the SVP in less time, which compensate in most of the cases.

Lattice basis reduction algorithms are used in several applications, not only in the SVP. They have also been used in signal processing applications, such as Global Positioning System (GPS), color space estimation in JPEG pictures, frequency estimation, and particularly data detection and precoding in wireless communications [Wbben et al. (2011), Tian and Qiao (2013)].

1 Lattice challenge - https://www.latticechallenge.org
2 SVP challenge - https://www.latticechallenge.org/svp-challenge/

1.1 Motivation

Lattice-based cryptography has been a hot topic in the past 10 years, because systems based on lattices are believed to be secure against quantum computer attacks. These systems are based on the hardness of the SVP in theory, and of α-SVP in practice. While the SVP has been formulated more than a century ago, the algorithmic study of lattices started only in the early eighties, and the development of parallel algorithms for the SVP is even more recent, with developments in the last five years.

Despite the theoretical and practical hardness of the SVP, it is important to keep searching for new more efficient implementations or algorithms to prove that a particular problem may be easier to solve than the expected. The constant scrutiny of these problems is crucial to the scientific community, where a particular problem may be considered reliable or not. Thus, this dissertation focuses on lattice basis reduction algorithms and one of its key requirement, MP arithmetic.

1.2 Contribution

The work developed during this dissertation targeted performance improvements on lattice basis reduction techniques that lead to scientific contributions. These include:

- Development of an efficient *'Multiple precision Integer Module'* (MpIM)[3] with mathematical operations, namely addition, increment, subtraction, decrement, multiplication, division, left and right shifts, and several logical operations;
- Development of a *'Lattice Basis Reduction'* (LattBRed)[3] module. These include:
  - A novel efficient implementation of the Qiao's Jacobi method;
  - Parallel and MP versions of the Qiao's Jacobi method;
  - MP implementations of the LLL and BKZ algorithms.
- A basis quality assessment of the LLL algorithm and Qiao's Jacobi method for Ajtai-type and Goldstein and Mayer lattice basis.

1.3 Roadmap

This dissertation is structured in six chapters. The first chapter introduces the reader the theme of this dissertation, and briefly explains the relevance of this topic for the scientific community.

3 Module available at `https://github.com/heldergoncalves92`

The next chapter describes the necessary background to quickly understand the main subjects related to lattice basis reduction and describes the computational environment for the experimental work. The current approaches for MP and lattice basis reduction algorithms are presented in this chapter.

Chapter 3 presents the implemented MP operations and compares the module performance with existing libraries.

Chapter 4 is dedicated to the Qiao's Jacobi method. It discusses the performance results achieved with the sequential and parallel versions of the algorithm and assesses a lattice basis quality of the Qiao's Jacobi method.

Chapter 5 describes proposed parallel approaches of the LLL and BKZ algorithms and assesses the quality of the output basis of combining the LLL and the BKZ algorithm with the Qiao's Jacobi method.

Finally, chapter 6 concludes the dissertation taking into account the obtained results, and leave guidelines for future work that could not be finished or covered.

2

BACKGROUND AND SETUP

An SVP-solver searches the shortest non-zero vector of a lattice $\mathcal{L}$. However, they used to be high complexity algorithms and they may run in exponential execution times. Some lattice basis reduction algorithms produce reduced basis in polynomial time [Lenstra et al. (1982a)], but they do not solve the problem. The community have been doing a great effort in the last years in SVP-solvers and lattice basis reduction algorithms, in order to get more efficient solutions. SVP-solvers are a class of techniques that solve the SVP. Enumeration, sieving and random sampling algorithms are three of the main techniques in SVP-solvers.

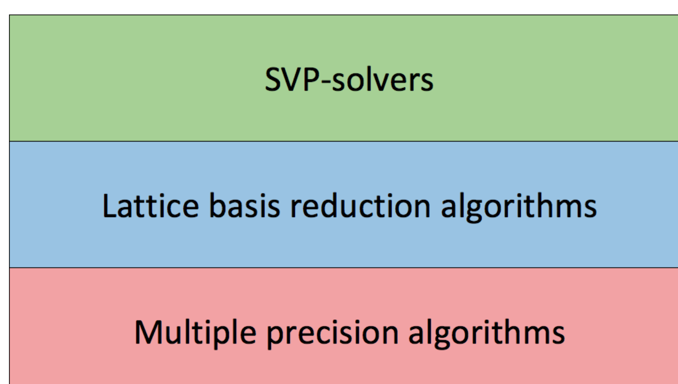


Figure 1: SVP panorama in three layers

Figure 1 illustrates a SVP panorama that this dissertation addresses. It splits the SVP into three different layers.

SVP-solvers and lattice basis reduction algorithms can be used as stand alone algorithms, however they perform better together. The *'SVP-solvers'* layer is on the top because these algorithms always return the shortest non-zero vector of the lattice. Although the lattice basis reduction algorithms can solve the SVP for small basis dimensions, usually they only get a reduced basis which can then be used by a SVP-solver. Thus, they are below the layer *'SVP-solvers'*. Finally, Multiple precision algorithms are in the bottom layer because they are used in both upper layers to represent and perform computations on large numbers that are inherit to the problem. Thus, this layer can be consider as *support* of the others. The dissertation is focused on the two lower layers.

2.1 Multiple precision

Most current computer architectures support operations between integer scalars with up to 64 bits of precision. However, lattices in cryptography require numbers with a larger precision to ensure a better security in some applications. MP arithmetic requires the representation and computation of numbers that do not fit into primitive data types. With this approach it is possible to store and perform calculations on numbers whose precision digits are only limited by the available system memory.

Operations with primitive data types, whose numbers fit into processor registers, are considerably faster than the MP arithmetic. While primitive data types are implemented by hardware, MP arithmetic has to be implemented by software.

The MP history starts with a commercial IBM computer in the 50s[1]. Unlike the current MP, implemented by software, the IBM 702 implemented a integer arithmetic entirely in hardware on digit strings up to 511 digits. Later in the 60s appear the first widespread software MP implementation in MACLISP (a dialect of the Lisp programming language). Already in the 80s, the VAX/VMS and VM/CMS were the first operating systems to offer MP functionalities.

This dissertation is focused in MP integer arithmetic, thus the algorithms here presented are intended to handle large integer numbers.

2.1.1 *Current libraries*

Current MP libraries are available for many programming languages. Languages such as Ruby and Haskell offer built-in support, but its performance decreases. In C and C++, one of the most used libraries is the GNU Multiple Precision Arithmetic Library (GMP)[2].

GMP is a free library for MP arithmetic that was first released in 1991, and it has been updated since then. This library aims to have better implementations than any other MP library, mainly because it (i) uses full words to represent a large number, (ii) uses different algorithms for different operand sizes since the algorithm efficiency depends on the operand, (iii) is specialized for different processor architectures with highly optimized assembly code, and (iv) is continuously updated by the worldwide community.

The Number Theory Library (NTL) is other widely used MP library[3]. Unlike GMP that only implements MP modules, NTL has a strong component in number theory providing data structures and algorithms (e.g. routines for lattice basis reduction, Gaussian elimination). It makes it way more attractive than GMP when the the research target goes beyond performance. The NTL author considers it a high-performance library and to increase its

1 Arbitrary precision arithmetic - https://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic
2 GMP - https://gmplib.org
3 NTL - http://shoup.net/ntl/

performance when using MP integer arithmetic, the author recommends to compile NTL with GMP. It also compares the relative performance of NTL against a similar library [Shoup (2016)].

Class Library for Numbers (CLN) is a MP library for efficient computations[4]. It stands out of the two previously libraries with a rich set of number classes, e.g., rational and complex numbers. As most high-performance libraries, it is implemented with C++ which brings efficiency, algebraic syntax and type safety. The CLN's author claims that it is very efficient in MP integer arithmetic with the use of the Karatsuba algorithm [Karatsuba and Ofman (1962), Karatsuba (1995), Knuth (1997)] and the Fast Fourier Transform (FFT) method [Schönhage and Strassen (1971)]. As most MP libraries, CLN is also dependent of the GMP.

The previous MP libraries were consider for further experimental work to this dissertation due to its performance and MP number type. However, some well rated libraries were not considered for further experimental work, namely Multiple-Precision FP computations with correct Rounding library (MPFR)[5] [Fousse et al. (2007)], Modular-positional Floating-point format (MF-format) [Isupov and Knyazkov (2015)], Multiple Precision Integers and Rationals library (MPIR)[6], Boost[7], Multiple Precision Floating-point Interval library (MPFI)[8] [Revol and Rouillier (2005)], MPFUN2015[9], ARPREC[10] , GNU Multiple Precision Complex library (MPC)[11], GNU Multi-Precision Rational Interval Arithmetic library (MPRIA)[12] and Computer Algebra System (PARI/GP)[13]. The exclusion of these libraries had several reasons: (i) their main functionalities are not relevant in the case study (e.g floating-point arithmetic, complex numbers, interval arithmetic and others), and (ii) several problems occurred when used (e.g. setup or segmentation fault problems and only beta releases).

In addition to these libraries others were also excluded because (i) we could not find relevant information about them, (ii) we assumed that their performance was lagging behind since they were not updated for several years or benchmarks showed that there are more efficient libraries, and (iii) the target programming language is not C/C++. The list include Fast LIbrary for Number Theory (FLINT)[14], TTMath Bignum Library (TTMath)[15], Arbitrary

---

4 CLN - http://www.ginac.de/CLN/
5 MPFR - http://www.mpfr.org
6 MPIR - http://mpir.org
7 Boost - http://www.boost.org
8 MPFI - http://mpfi.gforge.inria.fr
9 MPFUN2015 - http://www.davidhbailey.com/dhbsoftware
10 ARPREC - http://crd-legacy.lbl.gov/~dhbailey/mpdist/
11 MPC - http://www.multiprecision.org
12 MPRIA - https://www.gnu.org/software/mpria/
13 PARI/GP - http://pari.math.u-bordeaux.fr
14 FLINT - http://www.flintlib.org
15 TTMath - http://www.ttmath.org

precision library (ApFloat)[16], LibTomMath[17], CORE Library (CORE)[18] [Du et al. (2002)], eXact Reals in C (XRC)[19], Multiple-precision Math (MpMath)[20], Software Carry-Save multiple-precision Library (SCSLib) [Defour et al. (2002), Defour and de Dinechin (2003)], Floating-point Arithmetic Library (FpALib)[21], Supporting High Precision on Graphics Processors (GARPREC)[22], CudA Multiple Precision ARithmetic librarY (CAMPARY)[23], General Decimal Arithmetic Specification (MPDecimal)[24], a Multi-precision Number Theory package (MpNT) [Hritcu et al. (2014), Tiplea et al. (2003)], Piologie[25] , BigDigits multiple-precision arithmetic (BigDigits)[26], C for eXtended Scientific Computing (C-XSC) [Hofschuster and Krämer (2004)], Multiple precision Integer and Rational Arithmetic C Library (MIRACL)[27] [Scott (2016)], My Arbitrary Precision Math library (MAPM)[28] [Ring (2001)]and simple and complete bignum C library (bigz)[29].

2.1.2 *Integer Representation*

To represent MP numbers, it is necessary to create a structure that supports all computations. The structure must allow efficient computations over the data.

The Residue Number System (RNS) was created by Sun Tsu Suan-Ching in the 4th century. The RNS is based in the Chinese remainder theorem for its operations. It uses a set of small numbers that fit in the primitive data types to represent a large MP number. As a large MP number is composed of a set of smaller numbers, a MP operation can be performed by compute in parallel and independently each small number.

However, RNS have some limitations, such as the division operation and the comparison of numbers in order to improve the RNS performance several works have been done [Kaltofen and Hitz (1995), Chren (1990), Isupov and Knyazkov (2015)]. RNS cannot efficiently compare two numbers: it has to convert those numbers to other representation to know, for example, which one is greater. To know more about this representational system see [Omondi and Premkumar (2007)].

16 ApFloat - http://www.apfloat.org
17 LibTomMath - http://www.libtom.net
18 CORE - http://cs.nyu.edu/exact/core_pages/intro.html
19 XRC - http://keithbriggs.info/xrc.html
20 MpMath - http://mpmath.org/
21 FpALib - https://sourceforge.net/projects/precisefloating/
22 GARPREC - https://code.google.com/archive/p/gpuprec/
23 CAMPARY - http://homepages.laas.fr/mmjoldes/campary/
24 MPDecimal - http://www.bytereef.org/mpdecimal/
25 Piologie - http://think-automobility.org/geek-stuff/piologie
26 BigDigits - http://www.di-mgt.com.au/bigdigits.html
27 MIRACL - https://www.miracl.com/
28 MAPM - http://www.tc.umn.edu/~ringx004/mapm-main.html
29 Bigz - https://sourceforge.net/projects/bigz/

There are several formats to represent a MP number, but usually it is used an array of integer numbers, where we call *limb* to each position of the array. This dissertation represents each *limb* (usually with 32 or 64 bits) with β. A possible integer representation contains the following fields:

- Number;
- Size;
- Allocated Size;
- Sign.

The field *'Number'* is an array of integer numbers. Each array position (*limb*) represents one part of the large number. The large number is always represented in magnitude for an easier and efficient algorithm implementation without sign verifications. The magnitude of any number is usually called its absolute value or module. In field *'Number'*, one of the most critical choices is related to the primitive data type to be used at each position. A susceptible approach is to sel1ect the *'unsigned long'* data type, in C, for each position, which is represented with 64 bits in a 64-bit processor architecture. Libraries such as GMP use the *'unsigned long'* data type, while other libraries, such as NTL use the *'long'* data type.

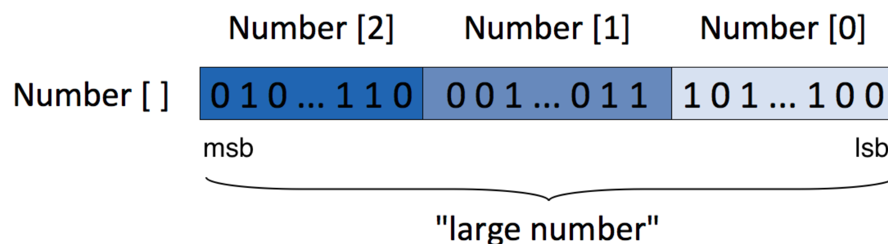


Figure 2: Binary representation of a large number with 3 *limbs*.

Figure 2 illustrates how a large number is represented in an array of integer numbers. It represents a number in binary and its respective positions in *'Number'*. The little-endian format is followed where the least significant *limb* (LSL) is in the beginning of the array, and the most significant *limb* (MSL) is at the last position. Note that the LSL is in the lower memory position. This representation may be considered a reversed list. MP algorithms usually start to compute the LSL. Thus, the representation helps the processor on data prefetching, avoiding several cache misses and hiding memory latency.

The field *'Size'* stores the number of *limbs* used to represent the large number. In Figure 2, this variable would have three as value. To represent the zero number, it sets the variable *'Size'* to zero, avoiding the access to the array where it is represented the number.

The field *'Allocated Size'* contains memory size allocated in bytes for the array *'Number'*. This value is always greater than or equal to the variable *'Size'*. If there is not enough memory allocated, a procedure will do it automatically.

The last field is the *'Sign'*. This variable is a boolean and if it is false the number is positive or zero and if its value is true the number is negative.

There are many ways to represent the number's sign. A simple format is the representation on the GMP. Contrary to our representation the GMP does not use a field to the sign because in its representation, the large number's sign is included in its size variable. A sign and magnitude representation is internally used, and the sign bit is used to know the large number's sign. This representation saves some memory, but to determine which is the sign or size of the large number, it requires a bit more computation than our representation, i.e., *abs* and *xor* functions and some nested-ifs.

Currently, this approach is implemented in the MP module presented in Chapter 3.

2.1.3 *Addition and Subtraction*

In MP, the simplest algorithms are the addition and subtraction algorithms, which have a cost of $O(n)$ to a *n-limb* number. Despite the research of more efficient addition and subtraction implementations, this approach continues until this day, since new efficient algorithms have not yet appeared.

The cost of a multiplication is higher than an addition, so fast multiplication algorithms, such as Karatsuba algorithm, are obtained by replacing multiplications by additions.

In MP arithmetic a carry is a digit that is transferred from one column of digits to another column of more significant digits. The carry is part of the addition algorithm where it starts to compute the LSL and finishes in the MSL.

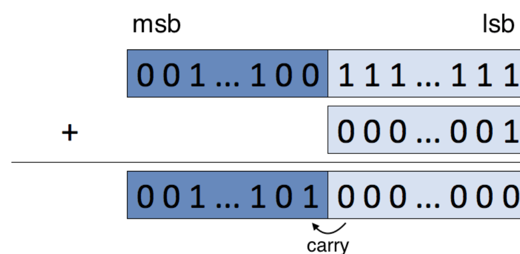


Figure 3: Addition with a carry digit in a large number with 2 *limbs*.

Figure 3 illustrates the addition of one to a *limb* that cannot represent the next integer number. In this case study, a carry is generated and the *limb* is reset to zero. The overflow is detected if the second operand is greater than the result. In this case we add the carry result to the next more significative *limb*, and so on and so forth.

The MP module implements the addition algorithm (Algorithm 2.1). In line 3, an overflow may occur, which in turn may generate a carry. Thus, the addition result cannot fit in the variable s. In this case there are three possibilities to overcome this situation:

- Use a machine instruction that gives the possible carry;
- Compute the module T, $t = a_i + b_i - T$. Then, to verify if the carry occurs, do the comparison $t \leq a_i$;
- Reserve a bit to check the carry occurrence, taking $\beta \leq T/2$.

Algorithm 2.1 Integer Addition, presented in [Brent and Zimmermann (2010)].

Require: $A = \sum_{i=0}^{n-1} a_i\beta^i$, $B = \sum_{i=0}^{n-1} b_i\beta^i$, carry $0 \leq d_{in} \leq 1$
Ensure: $C = \sum_{i=0}^{n-1} c_i\beta^i$, $0 \leq d \leq 1$
1: $d = d_{in}$
2: **for** $i = 0; i < n - 1$ **do**
3: $s = a_i + b_i + d$
4: $d = s \; div \; \beta$
5: $c_i = s \; mod \; \beta$
6: **end for**
7: **return** C, d

The subtraction algorithm is very similar to Algorithm 2.1. The only difference is in line 3, that is stated as '$s = a_i - b_i + d$'.

2.1.4 *Multiplication*

It is common to use algorithms that exchange some multiplications for additions, even if it brings some overhead associated.

In the multiplication, the choice of a particular algorithm is dependent on the input sizes and how fast a particular implementation is. Therefore, we implemented thresholds to determine which algorithm should be used to a certain situation. The thresholds are defined according to the performance of each algorithm. Several factors have an effect in the thresholds, i.e, the addition performance, where the the thresholds are as small as additions are faster. Figure 4 illustrates which is the best algorithm to multiply two numbers of x and y *limbs*. This technique is called of *squaring*.

Most of the proposed algorithms work with operands of the same input-size. However, the multiplications are unbalanced in most real problems. There are two main strategies to face this problem:

- to split the operands into different numbers of *limbs*;

- to split the operands into an equal number of *limbs* of unequal sizes.

```
     4    18    32    46    60    74    88   102   116   130   144   158   172   186   200
  4 bc
 11 bc bc
 18 bc bc 22
 25 bc bc bc 22
 32 bc bc bc bc 22
 39 bc bc bc 32 32 33
 46 bc bc bc 32 32 32 22
 53 bc bc bc bc 32 32 32 22
 60 bc bc bc bc 32 32 32 32 22
 67 bc bc bc bc 42 32 32 32 33 33
 74 bc bc bc bc 42 32 32 32 32 33 33
 81 bc bc bc bc 32 32 32 32 32 33 33 33
 88 bc bc bc bc 32 42 42 32 32 32 33 33 33
 95 bc bc bc bc 42 42 42 32 32 32 33 33 33 22
102 bc bc bc bc 42 42 42 42 32 32 32 33 33 44 33
109 bc bc bc bc bc 42 42 42 42 32 32 32 33 32 44 44
116 bc bc bc bc bc 42 42 42 42 32 32 32 32 32 44 44 44
123 bc bc bc bc bc 42 42 42 42 42 32 32 32 32 44 44 44 44
130 bc bc bc bc bc 42 42 42 42 42 42 32 32 32 44 44 44 44 44
137 bc bc bc bc bc 42 42 42 42 42 42 32 32 32 33 33 44 33 33 33
144 bc bc bc bc bc 42 42 42 42 42 42 32 32 32 32 32 33 44 33 33 33
151 bc bc bc bc bc 42 42 42 42 42 42 42 32 32 32 32 33 33 33 33 33 33
158 bc bc bc bc bc bc 42 42 42 42 42 42 32 32 32 32 32 33 33 33 33 33 33
165 bc bc bc bc bc bc 42 42 32 42 42 42 42 32 32 32 33 33 33 33 33 33 33 33
172 bc bc bc bc bc bc 42 42 42 42 42 42 42 42 32 32 32 32 32 32 44 33 44 44 44
179 bc bc bc bc bc bc 42 42 42 32 42 42 42 42 32 32 32 32 33 32 44 44 33 44 44 44
186 bc bc bc bc bc bc bc 42 42 42 42 42 42 42 42 32 32 32 33 32 44 44 44 44 44 44 44
193 bc bc bc bc bc bc bc 42 42 42 42 42 42 42 42 42 32 32 32 32 32 44 44 44 33 44 44 44
200 bc bc bc bc bc bc bc 42 42 42 42 42 42 42 42 42 32 32 32 32 32 33 44 44 44 44 44 44 44
```

Figure 4: The best algorithm to multiply two numbers of x and y *limbs*. bc is long multiplication, 22 is Karatsuba's algorithm and 33, 32, 44 and 42 are Toom variants (from [Brent and Zimmermann (2010)]).

Long multiplication

Several multiplication algorithms have been studied for decades. The long multiplication algorithm is the most used way to multiply two numbers by hand. It is the same algorithm taught in the elementary school. It is also known as schoolbook or basecase. Currently, it is the most efficient MP algorithm to multiply two operands with a small input size.

In line 3, the multiplication by β^j is simple, because it only needs to shift the result by j *limbs* on the direction of the most significant bit. In this algorithm, the major work is done on the computation of $A \cdot b_j$, and on its accumulation to C. There are many techniques to optimize this step, but the basic one is to save all the multiplication results to an array. The accumulation of results on the array is very heavy to the pipeline, so to decrease its impact, the size of operand A has to be greater than the size of the operand B. For a graphic description of the algorithm, see Figures 5 and 6.

Algorithm 2.2 Long Multiplication, presented in [Brent and Zimmermann (2010)].

Require: $A = \sum_{i=0}^{m-1} a_i \beta^i$, $B = \sum_{i=0}^{n-1} b_i \beta^i$
Ensure: $C = \sum_{i=0}^{m+n-1} c_i \beta^i$, $0 \leq d \leq 1$
1: $A = A \cdot b_0$
2: **for** $j = 0; j < n - 1$ **do**
3: $\quad C = C + \beta^j (A \cdot b_j)$
4: **end for**
5: **return** C

Grid method multiplication

Grid method multiplication, also known as box method, is other algorithm taught at elementary school. It breaks the addition and multiplication of the long multiplication in two steps, which makes it less efficient than the long multiplication.

Booth's multiplication

In 1950, Andrew Donald Booth invented a new multiplication algorithm [Booth (1951)]. Booth's multiplication algorithm improves the multiplication performance, where the shifting operations are faster than adding operations. There is no gain of performance in modern computers once shifting and adding operations take the same amount of time.

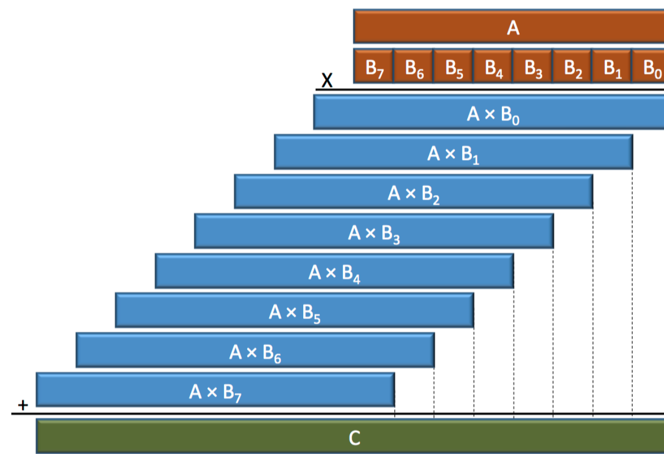


Figure 5: Long multiplication algorithm (from Intel documentation).

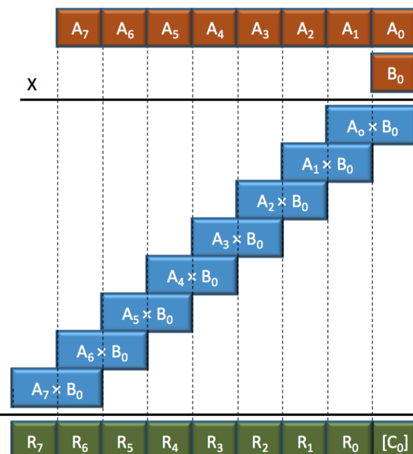


Figure 6: Multiplication step (from Intel documentation).

Gauss' complex multiplication

The first fast multiplication algorithm was discovered in 1805 by Carl Friedrich Gauss [Knuth (1997)]. Gauss's complex multiplication algorithm uses three multiplications and five additions instead of four multiplications and two additions. It is used to complex mul-

tiplications, which is not relevant for this dissertation. However, it was the beginning of the fast multiplication algorithms.

Karatsuba

The Divide and Conquer Algorithms (DCA) have a considerable value in MP arithmetic, i.e., Karatsuba and Toom-Cook algorithms [Knuth (1997), Mel (2007), Bodrato (2007)]. DCA are algorithms based on multi-branched recursion. These algorithms work by recursively breaking down a problem into two or more sub-problems of the same type, until these become simple enough so no more breakdowns make sense. The solution of the original problem is given by the combination of the solution of all the sub-problems generated.

Algorithm 2.3 Karatsuba's Algorithm, presented in [Brent and Zimmermann (2010)].

Require: $A = \sum_{i=0}^{m-1} a_i \beta^i$, $B = \sum_{i=0}^{n-1} b_i \beta^i$
Ensure: $C = \sum_{i=0}^{m+n-1} c_i \beta^i$, $0 \leq d \leq 1$
1: **if** $n < n_0$ **then return** BaseCaseMultiply(A, B)
2: **end if**
3: $k = \lceil n/2 \rceil$
4: $A_0 = A \; mod \; \beta^k$
5: $B_0 = B \; mod \; \beta^k$
6: $A_1 = A \; div \; \beta^k$
7: $B_1 = B \; div \; \beta^k$
8: $s_A = sign(A_0 - A_1)$
9: $s_B = sign(B_0 - B_1)$
10: $C_0 = $ Karatsuba(A_0, B_0)
11: $C_1 = $ Karatsuba(A_1, B_1)
12: $C_2 = $ Karatsuba$(|A_0 - A_1|, |B_0 - B_1|)$
13: **return** $C = C_0 + (C_0 + C_1 - s_A s_B C_2)\beta^k + C_1^{2k}$

Karatsuba algorithm is one of the most important fast multiplication algorithms. Not only because of its good performance for small input sizes but because it opened the door to several algorithms and implementations. This algorithm is a divide and conquer algorithm for multiplication of integers discovered in 1960 by Anatoly Karatsuba. Since its publication several works have been done, such as parallel implementations [Kuechlin et al. (1991), Char et al. (1994)] and its analysis in distributed memory architectures [Cesari and Maeder (1996)], generalizations [Weimerskirch and Paar (2006), Nursalman et al. (2014)] and FPGAs [von zur Gathen and Shokrollahi (2006)]. Its goal is to reduce the number of multiplications on a multiplication of two n-digit numbers to at most $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.585})$ single-digit multiplications. In practice, it reduces a multiplication of length n to three multiplications of length $n/2$, plus some overhead. Depending on the processor's architecture, its optimal threshold can vary from 10 to 100 *limbs*.

There are many versions of the Karatsuba algorithm, where the addictive and subtractive versions are the most known. Despite the small difference between them, the subtractive version is more attractive since it avoids possible carries. Thus, it is not necessary to have carry verification step, which makes the subtractive version more efficient.

Algorithm 2.3 illustrates the subtractive Karatsuba version. The different between both versions is stated as $|A_0 - A_1|$ and $|B_0 - B_1|$ instead of $A_0 + A_1$ and $B_0 + B_1$, at line 12. In lines 4-7, the operations *mod* and *div* are not executed in the implementation. It is only a mathematical way to indicate where the large number is split. Here, A_0 and B_0 have always the same number of *limbs*, but A_1 and B_1 can deal with different sizes.

Toom-Cook k-way

Toom-Cook k-way algorithms also follow the divide and conquer strategy. Its general idea is to split the problem in k sub-problems in each iteration. This new low complexity algorithm was proposed by Andrei Toom in 1963 [Toom (1963)], and later Stephen Cook cleaned the description [Stephen A. Cook (1969)]. The most known version is 3-way, also known as 'Toom-3'. In practice, it reduces a multiplication of length n to five multiplications of length $n/3$, plus some overhead. In the end it has a complexity of $\Theta(n^{\log_3 5}) \approx \Theta(n^{1.465})$. In general, Toom-Cook k-way goal is to reduce the number of multiplications on a multiplication of two n-digit numbers to $2k-1$ products of about n/k *limbs*, thus it complexity is $\Theta(n^v)$ where $v = \frac{log(2k-1)}{log(k)}$. An interesting fact is that the 2-way version is known as the Karatsuba algorithm, where the Toom-Cook is its generalization. Toom-Cook algorithm is typically used for intermediate input-size multiplications, because of the overhead associated that makes it slower than Karatsuba and long multiplication algorithms.

Algorithm 2.4 Toom-Cook 3-Way Algorithm, presented in [Brent and Zimmermann (2010)].

Require: $A = \sum_{i=0}^{m-1} a_i\beta^i$, $B = \sum_{i=0}^{n-1} b_i\beta^i$, $n_1 \geq 3$
Ensure: $C = \sum_{i=0}^{m+n-1} c_i\beta^i$
1: **if** $n < n_0$ **then return** KARATSUBA(A, B)
2: **end if**
3: write $A = a_0 + a_1x + a_2x^2$, $B = b_0 + b_1x + b_2x^2$ with $x = \beta^k$
4: $v_0 =$ TOOMCOOK3(a_0, b_0)
5: $v_1 =$ TOOMCOOK3$(a_0 + a_1 + a_2, b_0 + b_1 + b_2)$
6: $v_{-1} =$ TOOMCOOK3$(a_0 + a_2 - a_1, b_0 + b_2 - b_1)$
7: $v_2 =$ TOOMCOOK3$(a_0 + 2a_1 + 4a_2, b_0 + 2b_1 + 4b_2)$
8: $v_\infty =$ TOOMCOOK3(a_2, b_2)
9: $t_1 = (3v_0 + 2v_{-1} + v_2)/6 - 2v_\infty$
10: $t_2 = (v_1 + v_{-1})$
11: $c_0 = v_0$, $c_1 = v_1 - t_1$, $c_2 = t_2 - v_0 - v_\infty$, $c_3 = t_1 - t_2$, $c_4 = v_\infty$
12: **return** $C = c_0 + c_1\beta^k + c_2\beta^{2k} + c_3\beta^{3k} + c_4\beta^{4k}$, where $k = \lceil n/3 \rceil$

Algorithm 2.4 uses 5 evaluation points $(0, 1, -1, 2, \infty)$ and tries to optimize the evaluation and interpolation expression. The division in line 9 and 10 needs to be exact. The division operation is a heavy operation, but as the dividend is a 6 it is possible to do the division by shifting the number, followed by a division by three.

FFT-based

Despite Karatsuba and Toom-Cook algorithms have a good performance in its sequential version, Fagin claimed that they are not good candidates for parallel implementations due to its divide and conquer strategy, which requires a lot of interprocess communication [Fagin (1992)]. He also claims that the FFT-based algorithms are more suitable in parallel implementations, where several studies have been done [Jamieson et al. (1986), Johnsson et al. (1988)]. In addition to the inherit parallel properties, FFT-based algorithms are the more suitable algorithms for input-sizes with thousands of digits. Currently, there are two main FFT-based algorithms used in MP integer arithmetic.

Schönhage–Strassen algorithm is a FFT-based algorithm that was developed in 1971 by Arnold Schönhage and Volker Strassen [Schönhage and Strassen (1971)]. Currently, it is the most used FFT-based algorithm for large MP numbers, because of its low asymptotic complexity $\Theta(n \cdot log(n) \cdot log(log(n)))$. In practice, it uses recursive FFTs in rings with $2^n + 1$ elements. Until 2007, when the Fürer's algorithm was published [Fürer (2007), Covanov and Thomé (2015)], the Schönhage–Strassen was the algorithm with the lowest asymptotic complexity. Anindya De was the first to purpose a similar approach that relies on modular arithmetic [De et al. (2008)]. In 2014, the asymptotic complexity of $O(n \cdot log(n) \cdot 2^{2log^*(n)})$ was achieved by David Harvey with a modification to Fürer's algorithm [Harvey et al. (2016)]. However, it only gets an advantage for considerable large MP numbers, which makes it unpractical.

2.1.5 *Division*

The division is one of the most important algorithms to be optimized, because it uses to be one of the most heavy operations. A good strategy is to replace divisions by multiplications (e.g. precomputing the divisor's inverse). Usually, MP division algorithms perform more multiplications than divisions. Thus, the multiplication algorithms, such as Karatsuba, have an important role in the division algorithms, since its performance has a direct impact. Therefore, it is important to optimize well the multiplication.

As multiplication, there are two types of division algorithms. Slow division algorithms obtain a *limb* of the final result at each iteration. On other hand, fast division algorithms start with an approximation of the final number and compute a more accurate number after each iteration, i.e., Newton–Raphson and Goldschmidt algorithms.

In all division algorithms the divisor must be normalized. A number is normalized when its most significant *limb* satisfies $b_{n-1} \geq \beta/2$.

Algorithm 2.5 Long Division, presented in [Brent and Zimmermann (2010)].

Require: $A = \sum_{i=0}^{n+m-1} a_i\beta^i$, $B = \sum_{i=0}^{n-1} b_i\beta^i$, B normalized, $m \geq 0$
Ensure: $Q = \sum_{i=0}^{m-1} q_i\beta^i$, $0 \leq R < \beta$
1: **if** $A \geq \beta^m B$ **then** $q_m = 1$, $A = A - \beta^m B$
2: **else** $q_m = 0$
3: **end if**
4: **for** $i = m - 1;\ i \geq 0$ **do**
5: $\quad q_i^* = \lfloor (a_{n+i}\beta + a_{n+i-1}/b_{n-1}) \rfloor$
6: $\quad q_i = min(q_i^*, \beta - 1)$
7: $\quad A = A + q_i\beta^i B$
8: $\quad$ **while** $A < 0$ **do**
9: $\quad\quad q_i = q_i - 1$
10: $\quad\quad A = A + \beta^i B$
11: $\quad$ **end while**
12: **end for**
13: **return** Q, $R = A$

Long division

The long division algorithm [Knuth (1997)], also known as schoolbook division, is the standard division algorithm taught in the elementary school. As the DCA, it breaks a large division in a set of smaller problems, allowing the computation of large MP numbers. There are several variants of the long division such as the short division (used when the divisor is 1 *limb* size), and chunking method that is less efficient than the current long division algorithm introduced in 1600 by Henry Briggs. This algorithm has an asymptotic complexity of $\mathcal{O}(n^2)$.

Division by a limb

The single *limb* division is used when the divisor is represented with only one *limb*. It allows further optimizations. It can be both implemented with hardware division instructions or a multiplication by inverse [Moller and Granlund (2011)]. The choice of the method depends on the hardware. For example, CPUs with low latency multipliers can perform the main operation much faster than a hardware divide instructions. However, due to the cost of calculating the inverse, it compensates for input-sizes larger than 4-5 *limbs*.

Exact division

The exact division algorithm is used when it is known that the remainder of certain division is zero. This knowledge allowed Jebelean to do some significant optimizations [Jebelean (1993), Krandick and Jebelean (1996)]. This algorithm can be used within fast multiplication algorithms, such as Karatsuba algorithm and Toom-Cook generalizations [Jebelean (1996)].

Algorithm 2.6 Long division (binary version), from Wikipedia.

Require: $A = \sum_{i=0}^{n+m-1} a_i\beta^i$, $B = \sum_{i=0}^{n-1} b_i\beta^i$, B normalized, $m \geq 0$
Ensure: $Q = \sum_{i=0}^{m-1} q_i\beta^i$, $0 \leq R < \beta$
1: $Q = 0$
2: $R = 0$
3: $b =$ GETNUMBEROFBITS(A)
4: **for** $i = b - 1;\ i \geq 0$ **do**
5: $\quad R = R << 1$
6: $\quad R(0) = N(i)$
7: $\quad$ **if** $R \geq B$ **then**
8: $\quad\quad R = R - B$
9: $\quad\quad Q(i) = 1$
10: $\quad$ **end if**
11: **end for**
12: **return** Q, R

Usually, if the divisor is larger than a certain threshold the division is done with a divide and conquer algorithm [Burnikel et al. (1998), Moenck and Borodin (1972), Jebelean (1997), Hart (2015)]. Unlike the long division that determines a *limb* of the final result at each iteration, the divide and conquer division tries to get several *limbs* at once. It divides the original MP number in smaller MP numbers. Thus, it is possible to speedup the main division by using fast multiplication algorithms in smaller operands.

2.1.6 *Newton's method*

Newton's method, also known as Newton–Raphson method, is a fast division algorithm with the best asymptotic complexity [Schulte et al. (1994)]. It was created by Isac Newton and Joseph Raphson. Newton's method is widely used in number theory to solve several problems, such as the computation of roots. This method finds a reciprocal of the divisor and multiply it by the dividend. This, it successively finds better approximations of the final quotient.

Barret's division

Barret's division is a reduction algorithm created by Paul Barrett in 1986 [Barrett (1987)] to speedup the RSA encryption algorithm on an 'off-the-shelf' digital signal processing chip. It is designed to replace division by multiplications. Its first version just uses a single *limb*. However, this version is not able to perform MP divisions. Therefore, Barret propose a second version of his algorithm that approximates to the single *limb* implementation [Menezes et al. (1996)].

2.1.7 *Hensel's division*

Classical division algorithms usually cancel the most significant part of the MP number. However, Hensel's division algorithm cancels the least significant part of the number. The big difference of this strategy is that it is not necessary a correction step, since carries go in the opposite direction of the classical algorithms. There are cases, where only the remainder is desirable. In that cases this algorithm is known as Montgomery reduction [Knezevic et al. (2010)].

With so many algorithms available it is hard to select which one is the best for certain operation. In 2003, Karl Hasselström compared some of the most prominent MP division algorithms for several input-sizes [Hasselström (2003)].

Algorithm 2.7 Division By a *Limb*, presented in [Brent and Zimmermann (2010)].

Require: $A = \sum_{i=0}^{m-1} a_i\beta^i$, $0 \leq c < \beta$
Ensure: $Q = \sum_{i=0}^{m-1} q_i\beta^i$, $0 \leq b < 1$
1: $d = 1/c \ mod \ \beta$
2: $b = 0$
3: **for** $i = 0;\ i < n-1$ **do**
4:     **if** $b \leq a_i$ **then**
5:         $x = a_i - b$
6:         $b^{'} = 0$
7:     **else**
8:         $x = a_i - b + \beta$
9:         $b^{'} = 1$
10:     **end if**
11:     $q_i = dx \ mod \ \beta$
12:     $b^{''} = (q_i c - x)/\beta$
13:     $b = b^{'} + b^{''}$
14: **end for**
15: **return** $\sum_0^{n-1} q_i\beta^i$, b

2.2 LATTICE BASIS REDUCTION

Lattice basis reduction is a subgroup of problems in lattices. As mentioned before, the lattice basis reduction goal is to transform a given basis $\mathbf{B}$ of a lattice $\mathcal{L}$ into a closer to orthogonal and shorter basis such that $\mathcal{L}$ remains the same. The quality of a basis depends on the shortness and orthogonality of the basis vectors and other factores [Xu (2013)]. In order to reach that, it is possible to use the following operations:

- Swap two vectors of the basis. The swapping changes only the order of vectors in the basis it is trivial because L is not changed;
- Replace $\mathbf{b}_j$ by $-\mathbf{b}_j$. It is trivial because $\mathcal{L}$ remains the same;
- Subtracting or adding to a vector $\mathbf{b}_j$ a combination of other vectors of the basis $\mathbf{B}$. The lattice is not changed because when it is used an arbitrary vector that belongs to the lattice $\mathcal{L}$, it is achieved another vector that belongs to $\mathcal{L}$. Mathematically, if a vector is replaced by $\mathbf{b}_j \leftarrow \mathbf{b}_j + \sum_{i \neq j} z_i \mathbf{b}_i$, a new basis is obtained that will generate the same lattice $\mathcal{L}$.

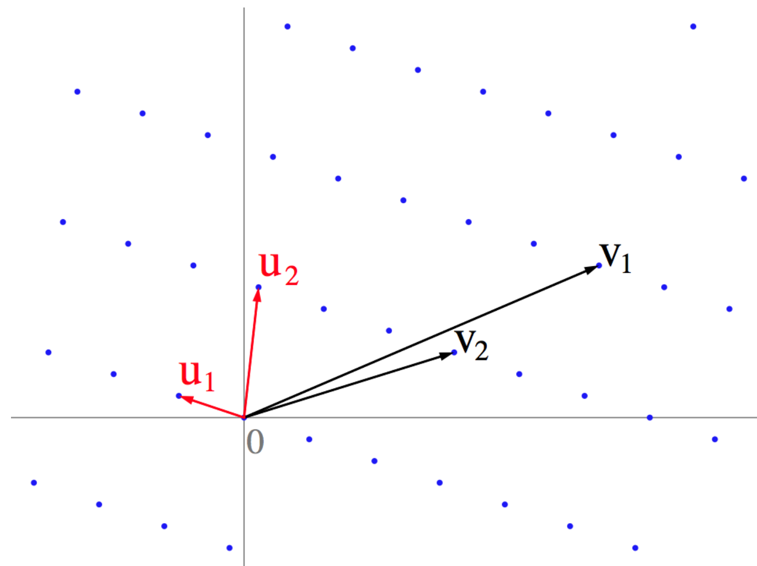


Figure 7: Lattice reduction in two dimensions: the black vectors are the given basis for the lattice, the red vectors are the reduced basis (from Wikipedia).

Lattice basis reduction is used to achieve the shortest vector of a basis when its rank is small. To higher ranks there is not known any algorithm to solve the SVP in polynomial time, but some lattice reduction algorithms can find a nearly short vector in polynomial time [Lenstra et al. (1982a)], which is enough to some applications. The figure 7 shows a basis reduction example, where $\mathbf{v}_i$ is the vector of a basis $\mathbf{B}$, and $\mathbf{u}_i$ are the resultant vectors of the lattice basis reduction.

Finding a good reduced basis has proved helpful in many fields of computer science and mathematics, particularly in cryptology. A good example is the execution time of the SVP-solvers, where they took less time to finish in good quality basis.

2.2.1 *Basic Concepts*

This section contains some concepts that are important to easily understand about lattices and its inherit problems.

Determinant of a lattice

An interesting feature of a lattice $\mathcal{L}$ is its determinant ($det\ \mathcal{L}$), a relevant numerical invariant. Thus, two different basis with the same lattice $\mathcal{L}$ will have the same determinant because it does not depend on the choice of a basis $\mathbf{B}$. Geometrically, the determinant is the volume of the parallelepiped spanned by the basis.

In a full rank basis, where the number of basis vector is equal to the spanned dimension, the determinant of basis $\mathbf{B}$ is the volume of the parallelepiped spanned by its vectors. Besides, if the number of vectors is less than the dimension of the underlying space, then volume is $\sqrt{det(\mathbf{B}^T\mathbf{B})}$. In resume for a full rank lattice, we have:

$$det(\mathcal{L}) = det(\mathbf{B}) = \sqrt{det(\mathbf{B}^T\mathbf{B})} \tag{2}$$

Gram Matrix

The Gram Matrix of a set of vectors $\mathbf{B}$ is a square matrix composed of all possible inner product entries (Equation 3). This matrix is symmetric which means that $\mathbf{G} = \mathbf{G}^T$. It has important applications, such as the computation of linear independences, where a set of vectors is linearly independent if its determinant is different from zero. It will be widely used in a lattice reduction algorithm that will be presented ahead.

$$\mathbf{G}_{ij} = \langle \mathbf{B}_i, \mathbf{B}_j \rangle \tag{3}$$

Gram-Schmidt coefficients

Gram-Schmidt (GS) orthogonalization is a process to orthogonalize a set of vectors, i.e., a lattice basis. It computes an orthogonal basis $\mathbf{B}^*$ for the same vector space, where all vectors are orthogonal to all previous basis vectors. During the GS process the GS coefficients

and norms are computed. For advantage of some lattice reduction algorithms, the GS orthogonalization is computed iteratively by:

$$\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=0}^{j<i} \mu_{i,j}\mathbf{b}_j^*, \ \textit{where} \ \mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle} \quad (4)$$

The GS coefficients (μ) and its norm vectors are widely used in some lattice reduction algorithms, since it helps to get a more orthogonal basis. Notice that the orthogonal basis cannot belong to the lattice $\mathcal{L}$. Figure 8 illustrates an example of the first two steps of the GS orthogonalization, where $\mathbf{e}_i$ are normalized vectors.

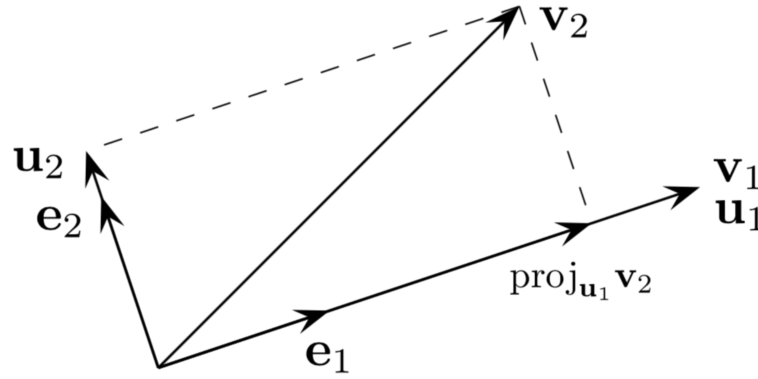


Figure 8: The first two steps of the Gram–Schmidt orthogonalization (from Wikipedia).

Lattice basis type

Lattice reduction algorithms can have different behaviours depending on the type of input basis. Therefore, it is important to study the behaviour of different algorithms in different types of basis.

There are ways to generate lattices that converge to an uniform distribution, accordingly to the Haar measure[30], when the integer parameters grow to infinity. Goldstein and Mayer (GM) are an example of a basis that converge to an uniform distribution [Goldstein and Mayer (2003)]. This type of bases follows the next steps to generate a basis of dimension n (i) choose a large prime integer p, (ii) choose randomly $n-1$ numbers (x_i) where x_i are integers in the range $0 \leq x_i < p$. Figure 9 illustrates some examples of GM matrices.

We also performed tests in Ajtai-type bases [Ajtai (1996)]. Ajtai introduces similar bases [Ajtai (2003)] to prove a lower-bound on the quality of Schnorr's block-type algorithms [Schnorr (1987)]. These bases are upper-triangular matrices where (i) it is chosen a random parameter a, (ii) $b_{i,i} = \lfloor 2^{(2n-i+1)^a} \rceil$, (iii) the $b'_{i,j}$s where $i > j$ are independent, randomly and uniformly selected in $\mathbb{Z} \cap [-\frac{b_{j,j}}{2}, \frac{b_{j,j}}{2}]$. The advantage of chose $b_{i,i} = \lfloor 2^{(2n-i+1)^a} \rceil$ is that the $\|b_i^*\|$'s decrease quickly, thus the basis is far from being reduced.

30 Haar measure - https://en.wikipedia.org/wiki/Haar_measure

GM bases have to use MP arithmetic. Besides, Ajtai-type bases can be represented in primitive data types.

$$\begin{pmatrix} p & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{pmatrix}, \begin{pmatrix} 1 & x_1 & & \\ & p & & \\ & & \ddots & \\ & & & 1 \end{pmatrix}, \ldots, \begin{pmatrix} 1 & & & x_1 \\ & \ddots & & \vdots \\ & & 1 & x_{n-1} \\ & & & p \end{pmatrix}$$

Figure 9: Examples of GM matrices.

2.2.2 *Lenstra–Lenstra–Lovász*

The Lenstra–Lenstra–Lovász (LLL) was the first prominent lattice basis reduction algorithm to be introduced. The LLL is a polynomial time algorithm invented by Arjen Lenstra, Hendrik Lenstra and László Lovász in 1982 [Lenstra et al. (1982b)]. Currently, the LLL algorithm has been successfully implemented, due to the Lovász condition that controls swapping operation between basis vectors. Therefore, all of the following works are mainly focused on (i) understanding statistical mean running behaviour and average complexity of the LLL algorithm [Nguyen and Stehlé (2006)] and (ii) improving the efficiency and stability of the LLL algorithm [Artur Mariano and Bischof (2016)]. An interesting fact is that many simulations and theoretical analysis confirm that the LLL algorithm performs much better in practice than the worst case bound of complexity.

LLL algorithm

The LLL algorithm is split into two main components. The first one aims to make the basis more orthogonal as possible with the Gram-Schmidt coefficients by computing a size-reduction of the vector $\mathbf{b}_k$. It is a size-reduced basis when $|\mu_{ij}| \leq \frac{1}{2}$, where $(1 \leq j < i \leq n)$ in $\mathbb{R}^n$. Usually the basis is size-reduced, but when $|\mu_{ij}| > \frac{1}{2}$ it replaces b_i with $(b_i - \lceil \mu_{ij} \rfloor b_j)$. The size-reduction component is described in Algorithm 2.8 between lines 4 and 9.

In the second component, it implements the Lovász swapping condition to make the reduced vectors as short as possible. The Lovász condition is denoted by:

$$\delta \|\mathbf{b}_i^*\|^2 \leq \|\mathbf{b}_{i+1}^* + \mu_{(i+1)i}\mathbf{b}_i^*\|^2 \tag{5}$$

where $\delta = 3/4$. A robust swapping condition implies using a larger value for the control parameter δ in the condition, which can be between 0.95 and 0.999. If the swapping is necessary, the vectors b_i and b_{i+1} will exchange themselves and then set the current stage of $(i+1)$ back to i.

Since LLL algorithm was proposed, several works have been done. However, only two major improvements were done. First, a very efficient floating-point version was proposed by Schnorr [Schnorr and Euchner (1994)], allowing to solve some exact problems more efficiently. Avoiding MP arithmetic and using primitive floating-point data types results in faster computations and in a minor number of swaps. Later, other floating-point versions appear with further optimizations [Nguên and Stehlé (2005)] that reduced the asymptotic complexity. Despite this optimization speeds up the LLL algorithm, it needs to be used with caution since it introduces floating-point errors.

Algorithm 2.8 LLL algorithm, presented in [Nguyen and Stehlé (2006)].

Require: *A basis* $(\mathbf{b}_1, ..., \mathbf{b}_n)$ *and* $\delta \in (\frac{1}{4}, 1)$
Ensure: *A* $LLL - reduced\ basis$
1: Compute Gram-Schmidt coefficients and norms
2: $k = 2$
3: **while** $k \leq n$ **do**
4: **for** $i = k - 1$ *to* $i = 1$ **do**
5: $\mathbf{b}_k = \mathbf{b}_k - \lceil \mu_{k,i} \rfloor \mathbf{b}_i$
6: **for** $j = 1$ *to* i **do**
7: $\mu_{k,j} = \mu_{k,j} - \lceil \mu_{k,i} \rfloor \mu_{i,j}$
8: **end for**
9: **end for**
10: $k' = k$
11: **while** $k > 2$ *and* $\delta \mathbf{c}_{k-1} > \mathbf{c}_{k'} + \sum_{i=k-1}^{k'-1} \mu_{k',i}^2 \mathbf{c}_i$ **do**
12: $k = k - 1$
13: **end while**
14: *Insert* $\mathbf{b}_{k'}$ *right before* $\mathbf{b}_k$
15: $k = k + 1$
16: **end while**
17: **return B**

The original LLL algorithm runs in polynomial time but it is just capable of generates a basis with medium quality. It led Schnorr and Euchner to propose the second major improvement. They introduced a LLL algorithm with a *deep insertion* technique. [Schnorr and Euchner (1994)], which allows to find shorter basis vectors, resulting in better reduced bases. In practice, it replaces the swapping step by a deep insertion. As well as the original LLL algorithm, this implementation make use of the Gram-Schmidt coefficients to make the basis as orthogonal as possible, but the Lovász condition is overwritten by a 'deep insertion' to achieve a basis with shorter vectors. Thus, the algorithm computes the following stronger condition:

$$\delta \|\mathbf{b}_i^*\|^2 \leq \|\mathbf{b}_k^* + \sum_{j=i}^{k-1} \mu_{lj} \mathbf{b}_j^*\|^2 \tag{6}$$

where $\delta = 3/4$, until it is true or $i < k$. If the condition is true the algorithm will insert b_k right before b_i. Schnorr and Euchner also proposed using a bigger value of $\delta = 0.99$.

At first, the complexity of the LLL algorithm with deep insertions was super polynomial, with examples showing that its practical running time is longer by a few times than the original LLL algorithm, but Gama and Nguyen [Gama and Nguyen (2008)] reported that the Schnorr version has super exponential complexity.

2.2.3 *Hermite-Korkine-Zolotarev*

The Hermite-Korkine-Zolotarev (HKZ) is a lattice-reduction algorithm that achieves reduced basis with better quality. Its vectors are more orthogonal and shorter than the previous LLL algorithms, but it requires more computation time to converge [Hanrot and Stehlé (2008)].

A basis $\mathbf{B}$ of a lattice $\mathcal{L}$, is HKZ-reduced if its first vector reaches the minimum of $\mathcal{L}$ and if orthogonally projected to $\mathbf{b}_1$ the other vectors $\mathbf{b}_i$'s are themselves HKZ-reduced.

2.2.4 *Block-Korkine-Zolotarev*

Schnorr proposed several works during its career. In 1994, he introduces a new lattice basis reduction algorithm [Schnorr and Euchner (1994)]. The Block-Korkine-Zolotarev (BKZ) combines the quality basis output of the HKZ with the good execution times of LLL. It combines a lattice basis reduction algorithm with an SVP-solver, the LLL algorithm and an enumeration algorithm respectively. In BKZ, the lattice reduction algorithm and the enumeration algorithm are dependents on each other, and the enumeration algorithm operates as a function of the main algorithm.

The BKZ have an extra entry parameter ω that defines the window size. The window corresponds to the block of basis vectors where the enumeration algorithm executes. A bigger block size results in a more reduced basis. However, it is necessary some caution on choosing the window size since the running time increase significantly. It happens because the enumeration algorithm is super-exponential, $(2^{O(\omega^2)})$. The BKZ with $\omega = 20$ is very practical, but when the block size increases to $\omega \geq 25$, its running time increases significantly, which makes any high block size impracticable. This was the Achilles heel of the original BKZ, denying the possibility to operate with bigger blocks size.

The BKZ algorithm starts by calling the LLL algorithm to obtain a LLL-reduced basis and then it behaves like a sliding window over the basis, that will call successively an enumeration function, that returns the shortest vector found in the projected basis. Then, if a shorter vector is found, it is added to the current basis and the LLL algorithm is called again to remove the generated dependency.

Algorithm 2.9 BKZ algorithm, presented in [Chen and Nguyen (2011)].

Require: *A basis* $(\mathbf{b}_1, ..., \mathbf{b}_n)$, *its GramSchmidt orthogonalization, i.e.,* μ *and* $\mathbf{c}_i$, *a block size* $\omega \geq 2$, *and* $\delta \in (\frac{1}{4}, 1)$
Ensure: *A BKZ* $\omega -$ *reduced basis*

1: $z = j = 0$
2: LLL($\mathbf{B}$, δ)
3: **while** $z < n - 1$ **do**
4: $\quad j = (j\ mod(n-1)) + 1$
5: $\quad k = min(j + \omega - 1, n)$
6: $\quad h = min(k + 1, n)$
7: $\quad \mathbf{v} = \text{ENUM}(\mu, \mathbf{c})$
8: $\quad$ **if** $\mathbf{v} \neq (1, 0, ..., 0)$ **then**
9: $\quad\quad z = 0$
10: $\quad\quad$ LLL($(\mathbf{b}_1, ..., \sum_{i=j}^{k} \mathbf{v}_i \mathbf{b}_i, \mathbf{b}_j, ..., \mathbf{b}_h), \delta$)
11: $\quad$ **else**
12: $\quad\quad z = z + 1$
13: $\quad\quad$ LLL($(\mathbf{b}_1, ..., \mathbf{b}_h), \delta$)
14: $\quad$ **end if**
15: **end while**
16: **return B**

BKZ 2.0

Lately the BKZ 2.0 was presented by Chen and Nguyen [Chen and Nguyen (2011)], that made the first experiments in higher blocks size, $\omega \geq 40$. The BKZ 2.0 can be considered an updated BKZ which came with four improvements:

- An early-abort;
- A sound pruning enumeration;
- Preprocessing of local blocks;
- Optimizing the enumeration radius.

The first improvement is simply an early-abort and was based on a theoretical result of Harrot [Hanrot et al. (2011a)]. This improvement results on the addition of a parameter that specifies how many iterations should be performed. The improvement delivers an exponential speed up over BKZ over call with higher blocks size.

The other three improvements are related with the enumeration subroutine. The main modification consists in the incorporation of the sound pruning technique developed by Gama, Nguyen and Regev [Gama et al. (2010)]. The sound pruning uses specific bounding functions to discard some branches where the probability of to find a shorter vector is too small.

The cost of the enumeration subroutine is correlated with the quality of the reduced basis. Unfortunately, the BKZ only guarantees an LLL-reduced basis, which can be too expensive with higher blocks size. Thus, the BKZ 2.0 guarantee a stronger lattice reduction algorithm by preprocessing local blocks.

When the enumeration subroutine starts, the initial radius R used to be initialized as $R = \|\mathbf{b}_j^*\|$. Unfortunately, this radius could be far from the norm of the shortest vector, what will take more computation than if a nearby radius was defined. However, there is no theoretical proof of which size must be the initial radius. Thus, the radius approximation is based in the Gaussian Heuristic (GH), that provides a good estimate for the norm of the shortest vector of the lattice $\mathcal{L}$. The GH is denoted by:

$$GH(\mathcal{L}) = F(\frac{n}{2+1})^{\frac{1}{n}} \times det(\mathcal{L})^{\frac{1}{n}} \tag{7}$$

where $det(\mathcal{L})$ is the determinant of the lattice $\mathcal{L}$, and:

$$F(n) = (n-1)! \tag{8}$$

2.2.5 *Qiao's Jacobi method*

The Jacobi method proposed by Sanheng Qiao in 2012 is a recent algorithm for lattice basis reduction that claims to reduce a lattice basis with better orthogonality in less time than LLL algorithm [Qiao (2012)].

The Jacobi method is a very attractive algorithm because it is inherently parallel, due to matrix computations [Golub and Van Loan (1996)] that are the majority of the algorithm. Thus, there is a great chance to exploit parallel microarchitectures and improve its performance.

Lagrange's algorithm computes a reduced basis in a two-dimensional lattice, where S. Qiao found an algorithm that uses this principle, and given a lattice basis **A**, it gets an unimodular matrix **Z** of the same size, where **AZ** forms a reduced basis. The algorithm consists in applying the two-dimensional Lagrange's algorithm to all possible pair of vectors in original basis **A**. For a detailed description see the original paper [Qiao (2012)].

The Jacobi method output is said to be Lagrange-reduced (L-reduced). Thus, the basis **A** is conspired L-reduced if:

$$\|\mathbf{a}_i\| \leq \|\mathbf{a}_j\| \tag{9}$$

$$|\mathbf{a}_i^T \mathbf{a}_j| \leq \frac{\|\mathbf{a}_i\|^2}{2} \tag{10}$$

where $i < j$.

Algorithm 2.10 Qiao's Jacobi Method, presented in [Qiao (2012)].

Require: $\mathbf{A}_n$, *where n is the lattice dimension*
Ensure: $\mathbf{C}_n = \mathbf{AZ}$
1: $\mathbf{Z} = \mathbf{I}_n$
2: $\mathbf{G} = \mathbf{A}^T\mathbf{A}$
3: **while** $!isLagrangeReduced(\mathbf{G})$ **do**
4: **for** $i = 1$ *to* $n-1$ **do**
5: **for** $j = i+1$ *to* n **do**
6: $\mathbf{Z}' =$ LAGRANGE$(\mathbf{G}, i, j)$
7: $\mathbf{Z} = \mathbf{ZZ}'$
8: **end for**
9: **end for**
10: **end while**
11: **return** $\mathbf{C}$, d
12: **function** $Lagrange(\mathbf{G}, i, j)$
13: $\mathbf{Z}' = \mathbf{I}_n$
14: **if** $G_{ii} < G_{jj}$ **then**
15: swap the ith and jth rows of $\mathbf{G}$
16: swap the ith and jth column of $\mathbf{G}$
17: swap the ith and jth column of $\mathbf{Z}'$
18: **end if**
19: $q = \lfloor G_{ij}/G_{jj} \rceil$
20: $\mathbf{Z} = \mathbf{I}_n$
21: $Z_{ii} = 0,\ Z_{jj} = -q,\ Z_{ij} = Z_{ji} = 1$
22: $\mathbf{G} = \mathbf{Z}^T\mathbf{GZ}$
23: **return** $\mathbf{Z}'\mathbf{Z}$
24: **end function**

Two years later a parallel version and a GPU implementation of the Qiao's algorithm was introduced [Jeremic and Qiao (2014)]. Filip Jeremic and Sanzheng Qiao proposed a parallel version of the Qiao's Jacobi method, where the algorithm is parallelized by carrying out Lagrange's algorithm on two-dimensional sublattices simultaneously. However, there are some limitations that should be taken into account to avoid data hazards and to maintain the algorithm properties.

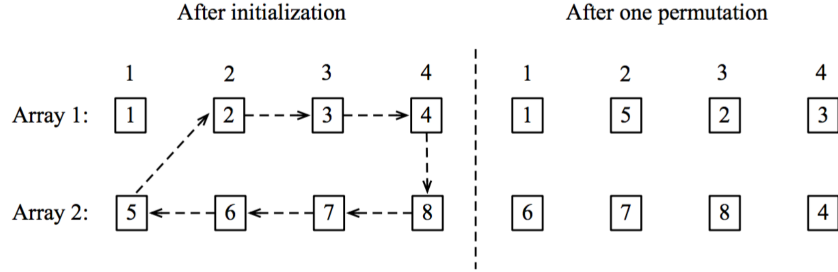


Figure 10: Chess tournament with $n = 8$ (from [Jeremic and Qiao (2014)]).

At each cycle of the Algorithm 2.10, the pair combination (i, j) is reduced, where the the two for-loop are generating all possible pair combinations between i and j. Thus, the parallel version should aim at the maximum of parallel reductions at once. Unfortunately, all the pair combinations cannot be reduced simultaneously.

It cannot reduce the pair combinations (i, j) and (j, k) at once, where $i < j < k$. This approach may generate a data hazard because some elements of the vector j can be reduce by i before this reduces the vector k. In other hand, we also cannot reduce $(i, i+1), (i, i+2), ..., (i, n)$ in parallel, due to data hazards created by swaps. For a more detailed description see [Jeremic and Qiao (2014)].

Jeremic suggests the *chess tournament ordering* [Wang and Qiao (2002)]. It maximizes concurrency while avoids data hazards and race conditions, and aims to parallelize $n/2$ pair combinations at once, wherein a column or row only belongs to a pair combination. This ordering will perform all pair combinations in $n-1$ permutations. The first initialization and the first permutation of the chess tournament ordering are illustrated in Figure 10.

The Algorithm 2.10 reduces i against $i+1$, i against $i+2$, and so on and so forth. Thus, the chess tournament vectors do not follow the same ordering, which may lead to a different output from the original one. Therefore, the algorithm can also take a different number of sweeps to converge to a solution. One sweep is a double for-loop.

2.2.6 *Measuring Basis Quality*

The same lattice can be represented by different basis, but it is also crucial to guarantee only good basis quality. Besides having a good parallel implementation, it is also important to have a good basis since these can significantly speedup some applications, namely the SVP.

A lattice basis reduction aims to make the vectors of the reduced basis as short as possible and as orthogonal as possible. Thus, the basis quality measurements should be related with the lattice reduction goals. It is hard to have a direct evaluation of the output of two different lattice reduction algorithms. To measure the basis quality, we use the following parameters [Mariano (2016)]:

1. HF of a basis;
2. Sequence of the GS norms;
3. Norm of the last GS vector;
4. Average of the norms of a basis;
5. The product of the norms of a basis;
6. The orthogonal defect of a basis;
7. Execution time of a SVP solver on a lattice.

The basis quality is better for lower values in all mentioned criteria, except for the norm of the last GS vector that is best for greater values and the sequence of the GS norms where it decreases slowly for better basis.

The Hermite Factor (HF) is widely used to measure the quality of different basis. The HF of a basis B of rank n can be defined as:

$$\mathcal{H}(\mathbf{B}) = \frac{||\mathbf{b}_1||^2}{vol(\mathcal{L})} \tag{11}$$

where $vol(\mathcal{L})$ is the volume of a lattice and it is equal to $(det(\mathcal{L}))^{\frac{1}{n}} = (det(\mathbf{B}^T\mathbf{B}))^{\frac{1}{n}}$. The volume of lattice is always 1, therefore is directly linked to the square norm of the first reduced vector. The HF can be interpreted to evaluate the mean and the improvements of the length of the shortest reduced vector against the lattice $\mathcal{L}$.

The orthogonal defect, also known as the Hadamard's inequality, is defined as:

$$\delta(\mathbf{B}) = \frac{\prod_{j=0}^{n} \|\mathbf{b}_j\|}{\sqrt{det(\mathbf{B}^T\mathbf{B})}} \tag{12}$$

2.3 EXPERIMENTAL ENVIRONMENT

There are many parallel programming models and a wide variety of implementations, where the purpose of a programming model is to easily adapt a software to multiple platforms. The adaptation to the right architecture and technology for a given problem must be the first big step of anyone that works in computer science.

Parallel programming is not so easy as it seems, because there is an inherent set of difficulties. The biggest disadvantage of parallel programming, but not the hardest to address in most cases, is the created computation, overhead, that aims the synchronization of the running threads. Data races used to be the most common difficulty and sometimes the most complex to solve. Another common problem is the load balance that can have certain running patterns, where certain yarns obtain more work, and ultimately have a severe impact on the performance.

To choose the right computing environment it was necessary a thorough study about the type of algorithms that are presented in this dissertation. Due to several related works, we explored this case study only on shared memory environments.

2.3.1 *Non-Uniform Memory Access*

The concept of shared memory allows that several programs access the same memory positions simultaneously. The shared memory allows programs to efficiently communicate or passing data between them. It can avoids redundant copies if data between several applications.

The hardware to a shared memory system is usually referent to a block of memory. There are three main types of memory organization to use in a shared memory system:

- Cache-Only Memory Architecture (COMA) - The local memories for the processors at each node are used as cache instead of actual main memory;

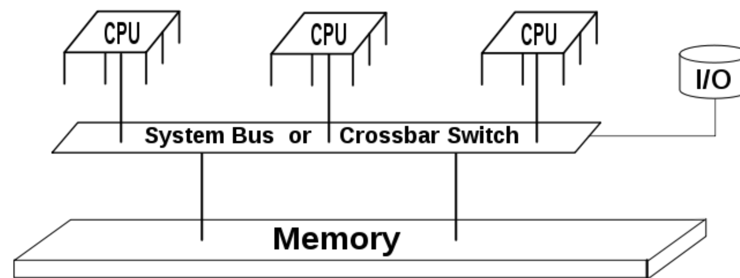


Figure 11: Shared memory system (from Google Images).

- Non-Uniform Memory Access (NUMA) - Memory access time depends on the memory location relative to a processor;
- Uniform Memory Access (UMA) - All the processors share the physical memory uniformly, and the access time does not depends of which processor makes the request;

Although three types of memory organization were presented, only NUMA organization will be described in greater detail.

Just one CPU can access to a shared memory system at time, which results in many race conditions to the possibility of simultaneous memory accesses. Therefore, in attempt to solve this problem it was created a type of memory organization dedicated to shared memory systems called NUMA, that provides separate memory blocks for each CPU, which allows several memory accesses at each moment.

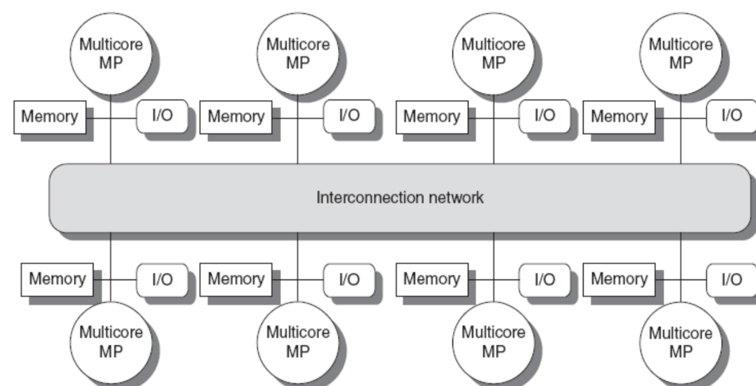


Figure 12: One possible architecture of a NUMA system (from Advanced Architectures slides).

The NUMA concept gives a global space address to each CPU node, then if a processor needs to access a memory block in another node, a copy is made to its own local cache. This is the opposite of a COMA memory organization, where the memory block would be moved instead. Migrate the memory block could bring a better use of memory resources and reduce the number of redundant copies, but it can raise the maintenance routines to

know where is a random memory block at a certain moment. Under a NUMA organization a certain processor can access its own memory much faster than a memory block in another node, where the access time will depend on the distance between both nodes.

2.3.2 *Vectorization*

Early processors had an Arithmetic Logic Unit (ALU) that only could compute one instruction on a pair of operands at once, which makes the processing of huge amounts of data impracticable.

Modern architectures can compute one instruction through n pair of operands simultaneously, depending on the processor architecture. This computer architecture is known by 'Single Instruction, Multiple Data' (SIMD). For a better comprehension look to Figure 13.

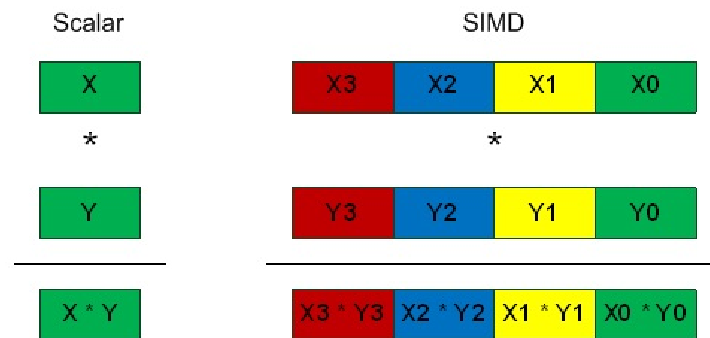


Figure 13: Scalar implementation vs vector implementation (from Google Images).

Unfortunately, vectorize is not always achievable in practice, because it is necessary that certain conditions were met, such as aligned memory.

2.3.3 *Methodologies*

Throughout this dissertation, there are several graphs and tables that convey crucial information to the discussion and conclusions of this dissertation. Depending on the problem and information we want to illustrate, various methodologies to measure the execution times were adopted.

To measure the execution times, we executed several times the same function and saved the best result. However, to guarantee that any evaluation benefits of the data locality, we performed several random operations between each repetition in order to fill all the CPU cache.

Chapter 3 presents several graphs with the MP's execution time of each operation. To do the experimental evaluations it is necessary to measure the time each operation takes. For

example, the addition operation have a brief execution time, about 0.2 ms in some cases. Short execution times are likely to contain measurement errors. To reduce the measurement errors the same operation is repeated several times in a for-loop, always with different inputs. In the experimental evaluation of the chapter 3, we repeated this process 6 times for each operation, with 400 different inputs, and registered the best execution time.

The problem faced in Chapter 4 and 5, with the Qiao's Jacobi method and the LLL algorithm is slightly different from the previous one. Since these algorithms are more complex than a simple MP addition and take long to perform, they are susceptible to contain less measurement errors. Thus, the obtained execution times consist in repeating the same test 10 times and record the best value.

To study the scalability of the Qiao's Jacobi method we used the previous guideline to measure the respective execution times by performing the same tests with different number of threads, in order to obtain the runtime evolution with the increasing number of threads.

The tests to assess the quality of the bases were perform just once, since the result is always the same for a particular case.

The algorithms were tested on a node of the Institute for Scientific Computing at the Darmstadt University of Technology. No other user processes were running on the node. The specification for this node is in Table 1.

Subject	**Description**
CPU Device(s)	2 Intel Xeon CPU $E5$-$2698v3$
Instruction Set	64 bits with 256 bits
Number of cores	2×16
SMT	Hyper-threading
Clock base frequency	2.3 GHz
Max turbo frequency	3.6 GHz
L1(I/D) Cache	32KB / 32KB
L2 Cache	256 KB
Shared L3 Cache	40 MB

Table 1: Node specifications.

The code was compiled with Intel's C++ compiler, version 15.0.5.223, with the -O2 optimization flag.

3

THE MULTIPLE PRECISION INTEGER MODULE

This chapter describes the main steps to implement an efficient MP module, referred as the *'Multiple precision Integer Module'* (MpIM) in the runtime evaluations of section 3.5. Despite the several number of MP algorithms presented in the section 2.1, we will take a closer look at the implemented algorithms.

Every algorithm of the MpIM was subjected to a test validation: to run a given operation with all possible combinations, with 400 different numbers. This test was repeated for different input-sizes.

3.1 ADDITION AND SUBTRACTION

In order to get a more efficient implementation, it is always necessary to perform the calculations with the absolute number. Thus, the MP addition algorithm implements both addiction and subtraction algorithms. Take as example the addition of -8 to 5. Since it is necessary to guarantee the absolute value, it will perform $8 - 5$ and then corrects the sign of the number.

From the three possibilities reffered in the section 2.1.3 to solve the overflow problem, it is not possible to use the third resolution, since the proposed representation uses the *'unsigned long'* as primitive data type. We implemented five versions of the Algorithm 2.1:

1. An implementation with comparisons to detect overflows;
2. An implementation with dedicated Intel intrinsics to MP arithmetic, where the same instruction sum the two operands and the carry if there is an overflow on last addition;
3. An assembly implementation;
4. An assembly version that does two additions sequentially avoiding one overflow test per cycle;
5. Vectorized version with AVX 2.0.

Both assembly implementations use an *add-with-carry* instruction (ADC), and the addition is always done within the CPU registers. Unfortunately, the function inline was not possible in these versions because of register coherence problems.

We implemented three versions of the subtraction algorithm: (i) an implementation with comparisons to detect the borrow, (ii) an implementation with dedicated Intel instructions to MP arithmetic, where the same instruction subtracts the two operands and sum the borrow of the last operation, returning the borrow if it exists in the last subtraction, and (iii) an assembly implementation. The borrow is the subtraction special case for the overflow in the addition.

The second implementation of the addition algorithm uses the *'_addcarry_u64'* Intel intrinsic, and the subtraction algorithm uses the *'_subborrow_u64'* Intel intrinsic.

3.1.1 *Addition Vectorization*

The result of the *limb* i rely on the result of the *limb* $i - 1$ due to overflow problems. There is a chance to vectorize this algorithm, but it could take some overhead.

It is not possible to vectorize the Algorithm 2.1 and keep the cost of $O(n)$ to a *n-limb* number. With AVX 2.0, it is possible to get a good theoretical speedup (four times faster), by vectorizing four additions. If there are no carries on the comparison test, the best scenario is achieved. However, the worst scenario gets more computation and could be slower than the basic algorithm. It happens if a carry is identified in every comparison tests.

AVX and AVX 2.0 do not support comparisons of unsigned numbers, which makes it impossible to implement this strategy. The best chance to implement it, it is with Broadwell microarchitecture that supports AVX-512.

The implemented version does four additions at once. Then it compares the result to the operand, *limb* by *limb* and adds the carry on next *limb* more significant. The intrinsics used were:

- *_mm256_load_si256*;
- *_mm256_store_si256*;
- *_mm256_add_epi64*.

3.1.2 *Increment and Decrement*

The increment/decrement algorithm is similar to the addition/subtraction algorithm. The increment and decrement operators are unary operators that add or subtract one from their operand, respectively. The main difference is that the algorithm only does one addition or

subtraction on the less significant *limb*. If there is no carry on the increment or decrement, the algorithm stops. Since, it just performs one operation the vectorization is not feasible.

Algorithm 3.1 Integer Increment

Require: $A = \sum_{i=0}^{n-1} a_i\beta^i$, carry $0 \leq d_{in} \leq 1$
Ensure: $C = \sum_{i=0}^{n-1} c_i\beta^i$, $0 \leq d \leq 1$
1: $d = d_{in}$
2: $s = a_0 + 1 + d$
3: $d = s\ div\ \beta$
4: $c_0 = s\ mod\ \beta$
5: **for** $i = 1;\ d == 1$ **do**
6: $\quad s = a_i + d$
7: $\quad d = s\ div\ \beta$
8: $\quad c_i = s\ mod\ \beta$
9: **end for**
10: **return** $C,\ d$

The decrement algorithm is very similar to Algorithm 3.1. The difference is in lines 2 and 6, which become '$s = a_0 - 1 + d$' and '$s = a_i - d$', respectively.

Since these algorithms rely on a single operation, a vectorization approach is not feasible.

3.2 MULTIPLICATION

The performance obtained on the multiplication depends on the chosen algorithm that executes a particular pair of operands. The MP module implements two different multiplication algorithms. The long multiplication that used to obtain the best performance results for small input-sizes, and the Karatsuba algorithm used to perform well for larger input-sizes. To define the threshold that choses which algorithm we perform a runtime comparison in Figure 21. Other algorithms like the Toom-Cook k-way and FFT-based algorithm were not implemented because (i) the input-sizes of the tested bases are not suitable in these algorithms and (ii) due to time constraints.

3.2.1 *Long multiplication*

We implemented the long multiplication algorithm (Algorithm 2.2), which is also known by 'schoolbook multiplication' as it mimics the multiplication learned at elementary school.

To ensure that no computation is lost, the result has to be stored on a datatype that is equal to the sum of the input size of the two operands, i.e., if the operand A is length five and B is length four, then C needs to be at least length nine. That does not mean that C cannot be smaller, i.e ($15 \times 1 = 15$). The same happens when a 64 bit multiplication is done.

Thus, the implementation uses the *'unsigned __int128'* data type, in C++, that has 128 bits of precision. This data type is a C++ extension. In reality, it is a struct of two *'unsigned long'*, where simple operations like addition and multiplication are optimized at compiler level.

We implement the long multiplication algorithm in two ways: (i) an implementation that uses the *'unsigned __int128'* data type to save the multiplication result and then accumulate the two halves of the result, and (ii) an implementation with a late Intel instruction, available on the Haswell microarchitecture and folling ones, which aims to increase the performance of MP multiplications. The Intel intrinsic is the *'_mulx_u64'*. This instruction is an extension of the existing multiplication instruction but has two main advantages:

- Greater flexibility in register usage. The default multiply instruction have the destination registers implicity defined. With MULX, the destination registers may be distinct from the source operands, so that the source operands are not overwritten;
- Since no flags are modified, MULX instructions can be mixed with add-carry instructions without corrupting the carry chain;

To optimize even further the long multiplication, it is recommend to allocate a block of memory to store the multiplication result. It allows the loop-unroll of several multiplications in order to maximize multiplication throughput. Unfortunately, this technique was not implemented due to time constraints.

There are two more new instructions to support large integer arithmetic. The ADCX and the ADOX instructions are extension of the existing ADC but these two instructions are designed to support two different carry-chains concurrently. Since these instructions are available from Broadwell microarchitecture, the required hardware is not available.

Currently, it is impossible to vectorize this algorithm with the current *unsigned long* data type because there are any vector instructions that multiply two unsigned 64 bit length numbers and return an unsigned 128 bit length *limb*.

3.2.2 *Karatsuba*

A good implementation of the Karatsuba algorithm must rely on a good use of allocated memory. Thus, a good improvement is to avoid allocating memory for the intermediate results C_0, C_1, C_2, $|A_0 - A_1|$ and $|B_0 - B_1|$, where a possible solution is to allocate a large amount of memory on the first call of the algorithm. Our implementation computes how much memory will be necessary in the beginning and allocates memory just once.

Algorithm 2.3 represents a multiplication where both operands have the same input-size. However, it does not work in most real cases where the operands have different input-sizes. To handle with this problem the line 1 is a bit different. It handles a second condition that

checks how much unbalanced are the input-sizes of the operands. Here, the Karatsuba algorithm is executed if the condition '$4 \times A_{size} < 5 \times B_{size}$' holds true, where $A_{size} > B_{size}$. Otherwise, the long multiplication algorithm is executed.

Since the Karatsuba algorithm always executes the long multiplication algorithm, the performance of both algorithms is linked, which makes imperative to have a good implementation of the long multiplication algorithm.

3.3 DIVISION

The division is one of the most important algorithms to be optimized, because it used to be one of the heaviest and complex operations. The MpIM implements two versions of the long division algorithm. The implemented versions are based in the Algorithms 2.6 and 2.5, respectively. The first version is not the most efficient but is important to understand how a division algorithm works. The second version is more efficient since it performs less operations, however it was not finished due to time constraints.

The MpIM's division execution times stated in the section 3.5 correspond to the binary division version.

3.4 OTHER FUNCTIONS

3.4.1 *Logical Shifts*

A logical shift is a bitwise operation that shifts all the bits of its operand by n bits. In this case study, the described implementations are the logical left shift and the logical right shift. However, keep in mind that this is a different operation, and not an arithmetic or rotate shift.

Logical shifts are capable of perform an efficient multiplication or division of unsigned integers by powers of two. Shifting left by n bits on a signed or unsigned binary number has the effect of multiplying it by 2^n. Shifting right by n bits on an unsigned binary number has the effect of dividing it by 2^n.

Figures 14 and 15 illustrate two logical shifts where the new bits value of the shift are always a zero.

The MP implementation is more complicated than these logical shifts, where they are used as a procedure. Our implementation starts to create a mask that depends on the shift's value. It will be useful to filter the bits that have to be transferred between different *limbs*. In the mask, the bits that have to be transferred to the next/previous *limb* are set to one. However, if the shift value is a 64 multiple, both implementations do a memory move instead. These shifts are explained in greater detail below.

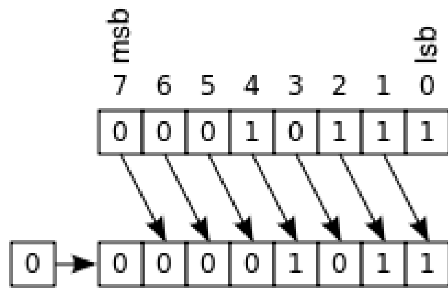


Figure 14: Simple logical right shift with the insertion of a zero on the left.

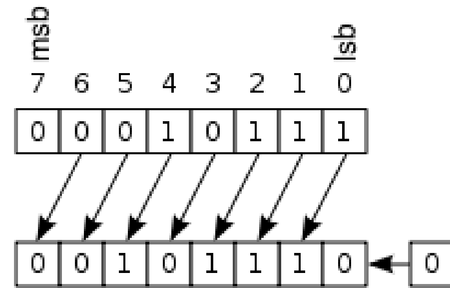


Figure 15: Simple logical left shift with the insertion of a zero on the right.

Right Shift

The right shift algorithm is illustrated in the Figure 16. The algorithm does not have to deal with memory allocations since the output number is always smaller than the input. The same does not happen in the left shift.

Due to data dependencies, the computation starts in the LSL and goes on until the most significant. Before it starts computing each position, the algorithm creates a mask that depends on the shift's value. The mask filters the least significant bits of each *limb*. For each position the algorithm have three main steps: (i) it starts to do the bitwise operation *'and'* between the *limb* $p+1$ and the mask, to store the bits that need to be shifted into a temporary variable, (ii) it does a right shift in the *limb* p, and (iii) it performs the bitwise operation *'or'* between the *limb* p and the temporary variable to store the bits that need to be shifted to *limb* p. The algorithm discards the least significant bits in the LSL.

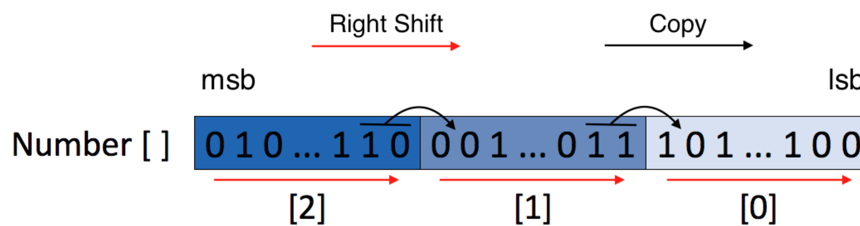


Figure 16: Right shift of 2 in a 3-limb large number.

Left Shift

The left shift algorithm is illustrated in the Figure 17. It is a bit more complicated than the right shift because it has to ensure that enough memory is allocated to handle with the shift's output.

Both logical shifts have similar algorithms. Due to data dependencies, the left shift algorithm starts to compute the MSL and go on until the least significant. Before it starts computing each position, the algorithm creates a mask that depends on the shift's value.

The mask gets the most significant bits of each *limb*. The algorithm starts by test and set the *limb* $Size + 1$ if it is needed. For each position the algorithm have three main steps: (i) it does a left shift in the *limb* p, (ii) it do the bitwise operation *'and'* between the *limb* $p - 1$ and the mask to store the bits that need to be shifted into a temporary variable, and (iii) it saves its values in a temporary variable. Finally, it performs the bitwise operation *'or'* between the *limb* p and the temporary variable to store in the *limb* p the bits that need to be shifted.

This implementation can generate a set of cache misses due to start computing the last position of the array *Number*. To reduce this negative impact in the performance, it is possible to use prefetch instructions to get the data into the cache before it is needed. Due to time constraints this strategy was not implemented.

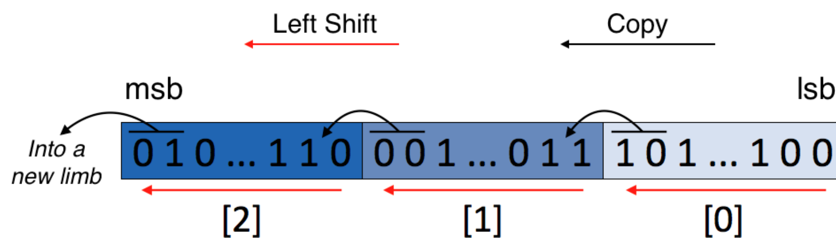


Figure 17: Left shift of 2 in a 3-limb large number.

Vectorization

After a detailed analysis is detected an opportunity to vectorize the two previous approaches in order to achieve better performance runtimes. There is only one data dependency in the right and left shifts, in the previously or in the following *limb*, depending on the shift operation. It is possible to mitigate this dependency with a special temporary variable. It was used the *__m256i* data type, which allows to store the bits that need to be shifted.

The approach uses AVX instructions to broadcast the mask, to load and to store data, and AVX 2.0 instructions to shift and bitwise operations. Due to the dependency $p + 1$ or $p - 1$, the AVX 2.0 instructions vectorize four *limbs* at once, and the large number have to be at least length five. The Intel intrinsics used are:

- *_mm256_load_si256;*
- *_mm256_loadu_si256;*
- *_mm256_store_si256;*
- *_mm256_set1_epi64x;*
- *_mm256_and_si256;*

- *_mm256_or_si256*;
- *_mm256_slli_epi64*;
- *_mm256_srli_epi64*.

3.4.2 *And/Or/Xor*

The logical functions have an important role in many cases. The algorithm is the same in all of them, just switching the logical operator. All of the three functions are vectorized with AVX 2.0. The function return a number with the size of the smallest argument, where it always computes the least significant *limbs*. The used intrinsics are:

- *_mm256_and_si256*;
- *_mm256_or_si256*;
- *_mm256_xor_si256*;
- *_mm256_load_si256*;
- *_mm256_store_si256*.

3.4.3 *Pseudo-Random Number Generator*

This function generates a pseudo-random number with *n limbs*, where *n* is a function's argument. This function is implemented in two versions: (i) a simple loop with a rand and (ii) a vectorized loop with AVX 2.0 instructions. To compile the second implementation it is necessary the Short Vector Random Number Generator Library (SVRNGL). Unfortunately, it just compile on ICC 16.0 or newer. The used intrinsics are:

- *svrng_new_mt19937_engine*;
- *svrng_generate4_ulong*;
- *_mm256_store_si256*.

3.4.4 *Compare*

The MpIM implements two compare functions. The first is a standard comparison of two numbers that returns 1 or -1 if the first operand is greater or smaller, respectively. The second compares the absolute value of the number. Both function return 0 if the operands are equal.

It were not yet vectorized due to priority judgement.

3.5 EVALUATION RESULTS

Addition

Figure 18 compares the five implemented additions, where the second assembly version get the best results. It succeeds because this version does two additions sequentially avoiding one overflow test per cycle.

Figure 19 compares the MpIM with other libraries, and the *assembly-v2* version obtains the best results for the smallest input-sizes and in the last two. The CLN get the bets results for medium input-sizes. The tested libraries have similar execution times for all different input-sizes, except the NTL that obtain the worst execution times for greater input-sizes.

Despite the efficient assembly implementation of the MpIM, our addition is not the best in all scenarios. It happens because libraries like GMP have several implementations in assembly language optimized for several microarchitectures. Contrary to GMP, our addition just implements the main for-loop in assembly language.

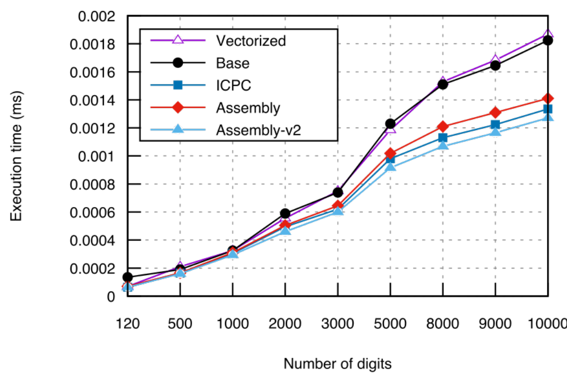


Figure 18: Comparison between the 5 addition implementations of the MpIM.

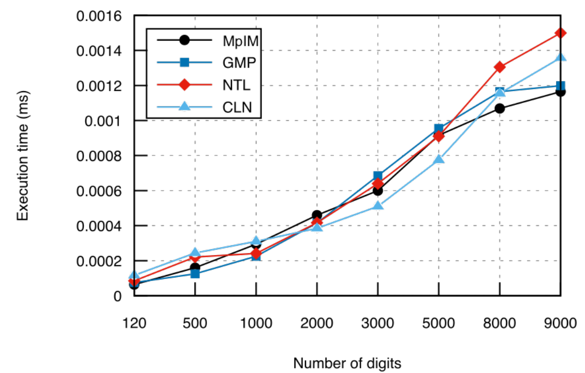


Figure 19: Comparison of MpIM's addition to other libraries.

Subtraction

Figure 20 compares the second implementation of the MpIM to other libraries. The subtraction is almost at the level of NTL and GMP, but it becomes worst on greater numbers (greater than 3000 digits). Surprisingly, CLN get the best results for medium and greater input-sizes.

Due to time constraints, we only implement two versions of the subtraction algorithm. However, it is possible to obtain better execution times with the same strategy used in the addition algorithm (*assembly-v2*), by avoiding one borrow test per cycle.

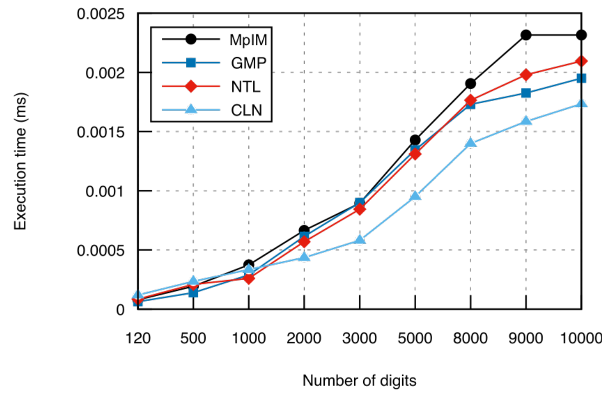


Figure 20: Comparison of MpIM's subtraction to other libraries.

Multiplication

Figure 21 compares the execution times of the long multiplication and the Karatsuba algorithm of the MpIM. As was expected the long multiplication obtains the best result for the smallest test (120 digits).

Figure 22 compares the MpIM's multiplication with other libraries. The presented execution times of the MpIM combine the execution time of both long multiplication and the Karatsuba algorithm. Our multiplication obtains better results than NTL until medium input-sizes (lower than 2000 digits). The bad results of the MpIM from 3000 input-sizes is explained by the use of more efficient algorithms for bigger input-sizes, i.e, Toom-Cook n-way. The figure shows that CLN and GMP have the best implementations of the multiplication algorithm.

The test with the input-size 120 shows that it is imperative to improve the MpIM's long multiplication to obtain an overall speedup for all input-sizes.

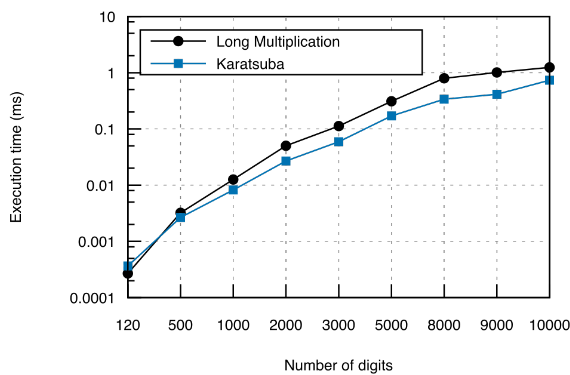


Figure 21: Comparison between the long multiplication and the Karatsuba implementations of the MpIM.

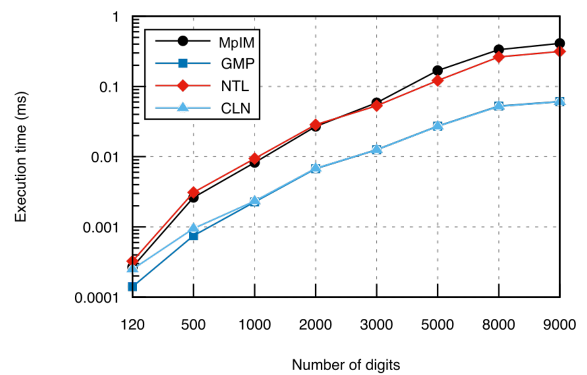


Figure 22: Comparison of MpIM's multiplication to other libraries.

Division

The MpIM's division is not compared to CLN library because it does not implement it.

Due to time constraints, it was not invested much time in the division algorithm and because of that the best results were not obtained. The MpIM's division have to be revised to obtain similar execution times such as GMP and NTL libraries.

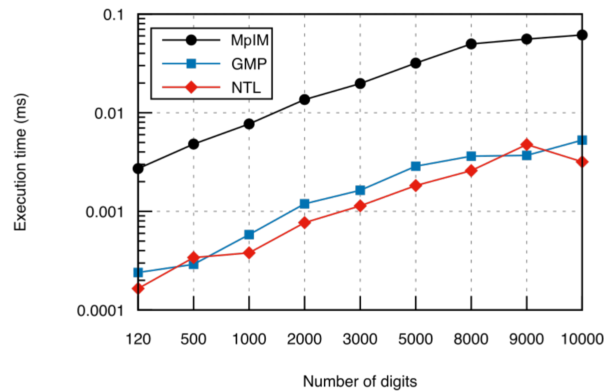


Figure 23: Comparison of MpIM's division to other libraries.

Shifts

The right and left shift functions are not compared to GMP library because it does not implement them.

Figures 24 compares the MpIM's right shift with other libraries. Our vectorized implementation get the best execution times with a reasonable margin for every input-sizes.

Figures 25 compares the MpIM's right shift with other libraries. The left shift stats the best results for smaller input-sizes ($\leq$ 500). Then, it stats between CNL and NTL for the remaining input-sizes. It happens because the MpIm's left shift starts by accessing higher memory locations and then accesses lower memory positions. Since larger input-sizes do not fit entirely in the L1 cache, it generates cache misses every time that the algorithm tries to access a particular memory position.

The right shift approach is the same that the left shift. However, they do not have similar execution times. It happens because the compiler is optimized, by default, to pre-fetch higher memory positions. Thus, cache misses will be generated in the left shift. Perchance, pre-fetch instructions could solve this problem.

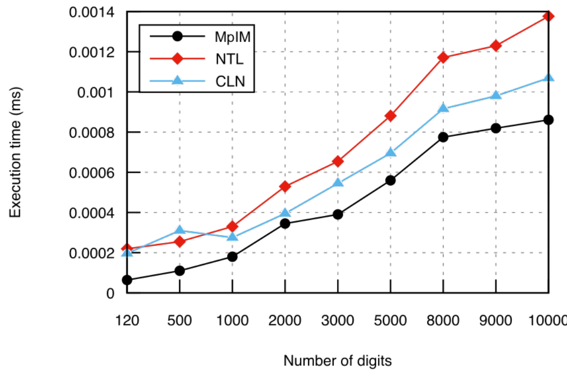


Figure 24: Comparison of MpIM's right shift to other libraries.

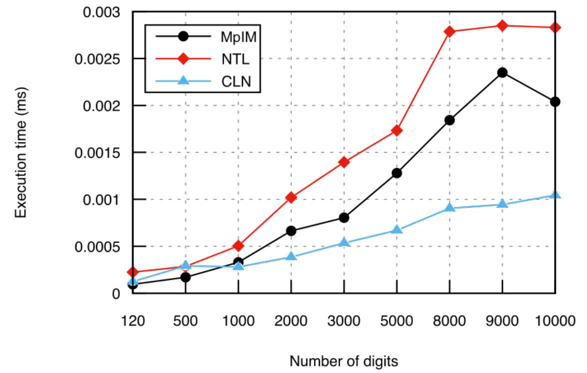


Figure 25: Comparison of MpIM's left shift to other libraries.

Others

All three logical functions are not compared to the NTL library because it does not implement them.

Figures 26, 27 and 28 compare the logical functions *'or'*, *'and'*, and *'xor'* of the MpIM to other libraries, respectively. The three MpIM's functions achieve the best execution times for almost every input-sizes. For lower input-sizes they are at the same level than GMP, however they increase their advantage on larger input-sizes. Figures 28 obtains almost twice better execution time for the biggest input-size, when compared to CLN library.

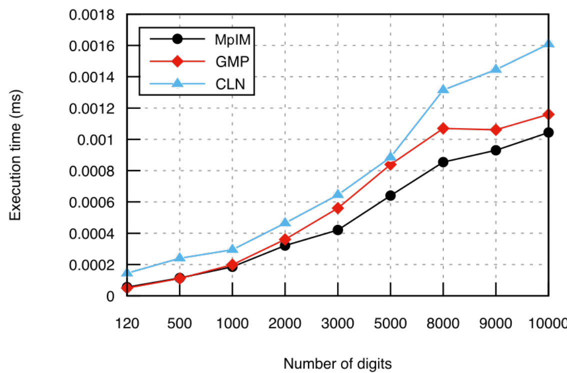


Figure 26: Comparison of MpIM's *'or'* function to other libraries.

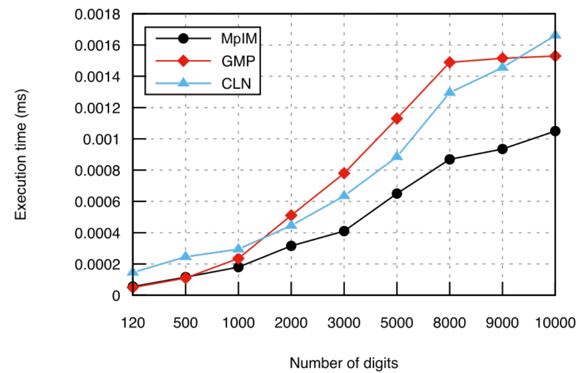


Figure 27: Comparison of MpIM's *'and'* function to other libraries.

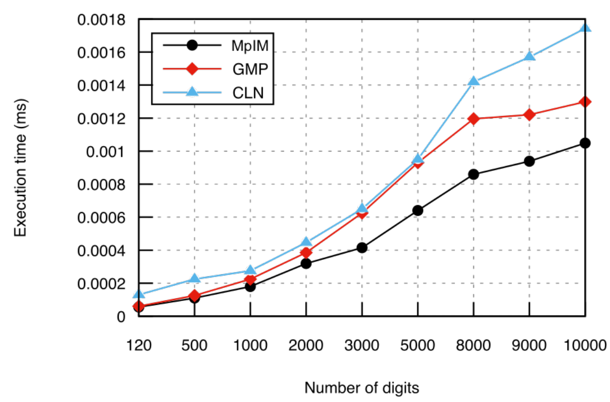


Figure 28: Comparison of MpIM's *'xor'* function to other libraries.

4

THE QIAO'S JACOBI METHOD

In this chapter we identify several bottlenecks of the Qiao's Jacobi method and propose efficient solutions to minimize them. Our final implementation is described in the Algorithm 4.1. While we identify a particular bottleneck we indicate the correspondent lines in the proposed version. All the proposed implementations of this chapter are included in the LattBRed module.

We start to inline the function *Lagrange* in the main algorithm. It allows to make changes directly in the matrix $\mathbf{Z}$ of the main function. To inline the *Lagrange* function we assume that $\mathbf{Z}'$ of the *Lagrange* function and $\mathbf{Z}$ of the main function are the same. With it, the matrix initialization in the line 13 is removed and the matrix multiplication in the line 7 is avoided, since the same result is produced in the line 23.

The Qiao's Jacobi method is highly parallel due to the majority of matrix computations, which makes it an opportunity to exploit it in parallel systems. We will see in this chapter that this statement may be risky. With a closer look to the Algorithm 2.10 we identified opportunities to avoid unnecessary computation on integral matrix multiplications. This opportunity appears in the *Lagrange* function when we analysis the output of a multiplication of any square matrix $\mathbf{M}$ by $\mathbf{Z}$ or $\mathbf{Z}^T$. We noticed that the multiplication $\mathbf{MZ}$ outcomes with almost the same result as matrix $\mathbf{M}$, where it subtracts the column i by the multiplication of q by the column j. The same happens in the matrix multiplication $\mathbf{Z}^T\mathbf{M}$, which computes rows instead of columns. Thus, it is possible to substitute heavy integral matrix multiplication by a row and column computations. Therefore, the matrix $\mathbf{Z}$ of the *Lagrange* function is not needed because the computations are done directly in matrices $\mathbf{G}$ and $\mathbf{Z}'$. To implement this improvement the line 11 is substituted by $\mathbf{G}(i{:}) = \mathbf{G}(i{:}) - q \times \mathbf{G}(j{:})$ and $\mathbf{G}({:}i) = \mathbf{G}({:}i) - q \times \mathbf{G}({:}j)$ plus the column and row swaps, and the line 12 is substituted by $\mathbf{Z}(i{:}) = \mathbf{Z}(i{:}) - q \times \mathbf{Z}(j{:})$ plus a row swap. The functions Row_Column_Computations and Row_Computations of the Algorithm 4.1 intent to represent these computations on a matrix. If q is 0, these function have no effect. For comparison purposes, we consider the Qiao's Jacobi method with this improvement as the base version.

After performing the functions Row_Column_Computations and Row_Computations, it needs to perform swap of the row and column i and j in order to maintain the algorithm

properties. These swaps are equivalent to the swaps that are in lines 15 to 17 of the *Lagrange* function. Thus, we avoid these swaps when $G_{ii} < G_{jj}$, because they cancel each other out.

Algorithm 4.1 Proposed Qiao's Jacobi method

Require: $\mathbf{A}_n$, *where n is the lattice dimension*
1: $\mathbf{Z} = \mathbf{I}_n$
2: $\mathbf{G} = \mathbf{A}\mathbf{A}^T$
3: **for** $i = 1$ *to* n **do**
4: $\quad$ $\mathbf{permut}_i = i$
5: **end for**
6: **while** $!isLagrangeReduced(\mathbf{G})$ **do**
7: $\quad$ **for** $iP = 0$ *to* $n - 1$ **do**
8: $\quad\quad$ **for** $jP = iP + 1$ *to* n **do**
9: $\quad\quad\quad$ $i = \mathbf{permut}_{iP}$
10: $\quad\quad\quad$ $j = \mathbf{permut}_{jP}$
11: $\quad\quad\quad$ **if** $G_{ii} < G_{jj}$ **then**
12: $\quad\quad\quad\quad$ $q = \lfloor G_{ij} / G_{ii} \rceil$
13: $\quad\quad\quad\quad$ **if** $q \neq 0$ **then**
14: $\quad\quad\quad\quad\quad$ Row_Column_Computations$(\mathbf{G}, q, j, i)$
15: $\quad\quad\quad\quad\quad$ Row_Computations$(\mathbf{Z}, q, j, i)$
16: $\quad\quad\quad\quad$ **end if**
17: $\quad\quad\quad$ **else**
18: $\quad\quad\quad\quad$ $q = \lfloor G_{ij} / G_{jj} \rceil$
19: $\quad\quad\quad\quad$ **if** $q \neq 0$ **then**
20: $\quad\quad\quad\quad\quad$ Row_Column_Computations$(\mathbf{G}, q, i, j)$
21: $\quad\quad\quad\quad\quad$ Row_Computations$(\mathbf{Z}, q, i, j)$
22: $\quad\quad\quad\quad$ **end if**
23: $\quad\quad\quad\quad$ SWAP$(\mathbf{permut}_{iP}, \mathbf{permut}_{jP})$
24: $\quad\quad\quad$ **end if**
25: $\quad\quad$ **end for**
26: $\quad$ **end for**
27: **end while**
28: $\mathbf{G} = \mathbf{Z}\mathbf{A}$
29: **return** GET_ORDERED_BASIS$(\mathbf{G}, \mathbf{permut})$

The matrix **G** is the gram-matrix of the matrix **A** (see line 2 of the main function). G is a symmetric matrix, which leads the *Lagrange* function to do the same computation twice when it computes the function Row_Column_Computations. Therefore, we assign the two elements at once, where $\mathbf{G}(: i) = \mathbf{G}(i :) = \mathbf{G}(i :) - q \times \mathbf{G}(j :)$. The improvement has one exception regarding the previous state. The exception occurs in the element G_{ii}, which has to be subtracted twice, first by $q \times G_{ij}$ and then by $q \times G_{ji}$.

By profiling our implementation, we notice that a large portion of the execution time was consumed by the swaps, around 70% in some cases. To avoid expensive row and column swaps, we add a vector of permutations **permut** with n positions, where $\mathbf{permut}_i = i$. Each

position indicates the index of a particular vector in the matrices **G** and **Z**. Thus, we just need to swap the values of the vector of permutations to maintain the algorithm properties. To implement it, we add the lines 3 and 4 to initialize the vector, the lines 9 and 10 to get the correspondent vector indexes and the swap of line 23.

The S. Qiao represents the basis vectors in the different columns of the matrix. Due to better data access patterns in C++, we represent the basis vectors in rows instead of columns.

After all these improvements, we were capable to achieve performances way more attractive than the original algorithm. We implemented two versions of the Algorithm 4.1, (i) a MP version to reduce GM bases, and (ii) a version with primitive data types to reduce Ajtai-type bases.

4.0.1 *Vectorization*

The second version uses primitive data types. Thus, we were able to exploit some features of current processor architectures, such as SIMD extensions. The following parts of the algorithm were improved:

- Gram-matrix computation ($\mathbf{AA}^T$);
- Row_Computations function;
- Matrix multiplication ($\mathbf{AZ}$).

The Gram-matrix computation was improved in several ways. Since the Gram-matrix is symmetric it only computes the inner products of the upper-triangular matrix and the lower-triangular is assigned with the previous results. Then, to allow the use of SIMD instructions we add padding in the end of each vector to guarantee the data alignment and the memory accesses are always made in contiguous memory, except in the lower-triangular assignments.

The function Row_Computations computes operations through the vector elements. Since the vector's data was aligned in the previous optimization, we just add Intel intrinsics.

It is impossible to efficiently vectorize the standard matrix multiplication algorithm due to the memory access patterns. There is vector instructions that gather the data, however the Intel warning about huge time penalties. To efficiently vectorize it, the matrix multiplication's cycles were reordered in a cache-friendly access pattern, allowing contiguous memory accesses (stride-1).

$$Speedup = \frac{Original\ Execution\ Time}{Expected\ Execution\ Time} = \frac{100\%}{(100\% - \mathbf{30.34}\%) + (\frac{30.34\%}{8})} \tag{13}$$

Although vectorization may lead to significative speedups, it is crucial to measure the theoretical speedup before initiating the implementation. The identified parts in the code consume 9.5%, 0.84% and 20% of the execution time, respectively, making a total of 30.34%. Taking in account that each SIMD instruction operates on 8 integer elements at once (in this case study), it is expected a theoretical speedup of 1.36. Equation 13 shows how the speedup was computed, where we inserted the bold face numbers on the equation. The used Intel intrinsics are:

- *_mm256_load_si256*;
- *_mm256_store_si256*;
- *_mm256_set1_epi32*;
- *_mm256_mullo_epi32*;
- *_mm256_add_epi32*;
- *_mm256_sub_epi32*;

4.0.2 *Evaluation Results*

This section measures the impact of most important improvements proposed to the Algorithm 2.10. We analysed the two implemented versions and compared them with the LLL algorithm.

It is not possible to compare the execution time of our implementations and the announced times in Qiao (2012) due to several reasons. First, some of our implementations use integer MP arithmetic, and S. Qiao does not indicate which data type is using in his experiment. However, floating-point arithmetic is referred in [Jeremic and Qiao (2014)]. Secondly, S. Qiao generates random lattice bases but did not indicate which type of lattice basis he performed the experiments. S. Qiao obtained a maximal number of 8 sweeps, however GM bases require more sweeps than the expected. The number of sweeps is illustrated in Figure 33. The reason why it performs so many sweeps is because GM bases are very ill-conditioned.

Figure 29 compares the LLL algorithm with the 'extended doubles' data type from the NTL and three MP implementations of the Qiao's Jacobi method. The *'MP Qiao Base'* is the base version of the Algorithm 2.10, where we removed almost every matrix multiplications. The *'MP Qiao w/ Swaps'* does not implement the proposed improvement that avoids integral matrix swaps, and the *'MP Qiao'* implements all the proposed improvements for MP arithmetic. The experiment shows that after all the improvements, the *'MP Qiao'* implementation is about 3 times faster than the base version, and the last improvement speed up

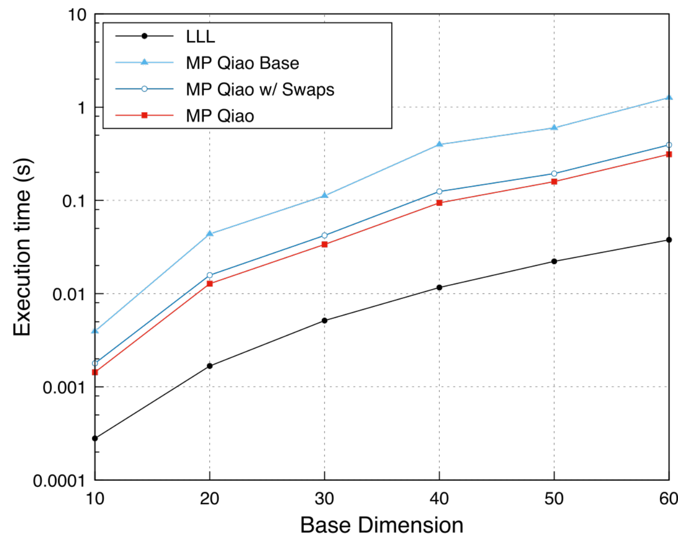


Figure 29: Execution times of sequential LLL_XD and Qiao's Jacobi method in GM bases.

the *'MP Qiao w/ Swaps'* is 1.25 times faster which we consider good because the MP version just swaps the pointers that point to the struct that handles the MP number. Figure 29 also shows that LLL algorithm gets better execution times in all tested dimensions, which makes it the best algorithm to execute with GM bases. This result was not the expected, and the bad results of the Qiao's Jacobi method are explained by the large amount of sweeps taken.

Figure 30 illustrates the execution times of the LLL algorithm and four incremental implementations of the Qiao's Jacobi method in Ajtai-type bases. The *'Qiao Base'* corresponds to the base version of the Qiao's Jacobi method with primitive data types. The other three implementations aims to measure the impact of some proposed improvements in the *'Qiao Base'* implementation, where the *'Qiao-Vec'* and *'Qiao-noVec'* implement almost every proposed improvements, but the first one takes advantage of vector instruction in ISA. Finally, the *'Qiao'* version implements all the proposed improvements.

Unlike Figure 29, the *'Qiao Base'* version gets similar results to the fastest LLL version of NTL (LLL_FP), where it is worse in lower dimensions and gets faster than LLL as the dimension increases. The Qiao's Jacobi method performs better in Ajtai-type bases than GM bases because it takes less sweeps to converge to a solution. Figure 31 shows the number of sweeps taken to converge a L-reduced basis. The Qiao's Jacobi method achieves a similar number of sweeps in Ajtai-type basis (6 sweeps), while testing a 500 dimension basis. Thus, we conclude that the good execution times depend on the input basis.

Figures 30 and 31 illustrate data from the same experiment. Therefore, the execution time is linked to the number of sweeps, i.e., in both graphs it increases for dimensions 325 and 450, and decreases for dimensions 350 and 475.

The *'Qiao-noVec'* version achieves the maximal speedup of 28 times faster than the *'Qiao Base'* for the dimension 325 and the minimal speedup of 3.8 times for the dimension 875.

Thus, we can say that the speedup is linked to the number of sweeps. In proposed improvements, the *'Qiao-noVec'* version already implements better memory access patterns and it is ready to support vector instructions.

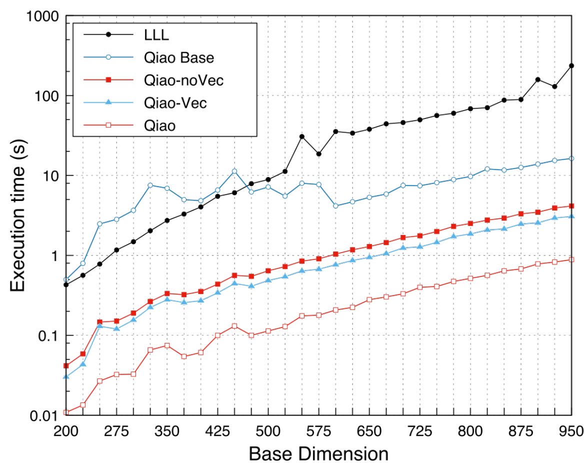


Figure 30: Execution times of sequential LLL_FP and Qiao algorithm in Ajtai-type bases.

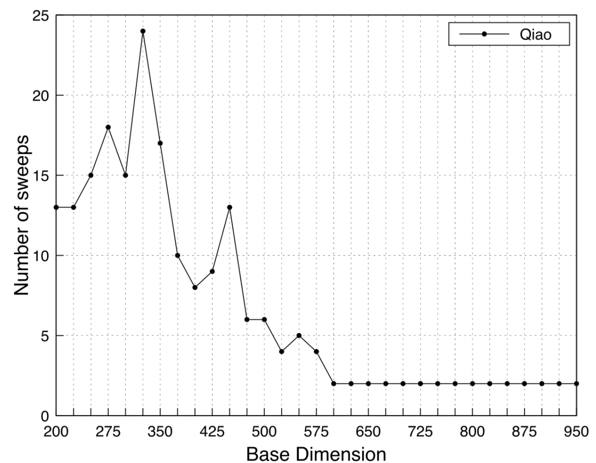


Figure 31: Number of necessary sweeps to converge to a solution in Ajtai-type bases.

Previously we computed a max theoretical speedup of 1.36 times faster by implementing the same code with SIMD instructions. Afterwards, the *'Qiao-Vec'* version got a speedup of 1.30 for the dimension 400, when compared with the *'Qiao-noVec'* version. It means that 5% of the execution time is the overhead.

Currently, the bottleneck of the *'Qiao-Vec'* version is in the column's swap, where, depending of the basis dimension, it consumes about 70% of the execution time. To avoid the swaps we implemented the proposed improvement that avoids column and row swaps, and obtained the execution times of the *'Qiao'* version. As expected the speedup depends of the number of swaps for a particular basis input. Comparing *'Qiao'* and *'Qiao-Vec'* versions, the first one obtained the average speedup of 3.64 times, where the maximal and minimal were 4.84 and 2.78 respectively.

Figure 32 shows the obtained speedups of the implemented Qiao's Jacobi method version and the LLL algorithm. After all the proposed improvements we obtained an average speedup of 42.84 times comparing *'Qiao'* and *'Qiao-Vec'* versions, and it was greater than 100 times for some dimensions. We also notice that the speedup of LLL and *'Qiao'* increases with higher dimensions, where *'Qiao'* is 265 times faster than LLL algorithm for dimension 950.

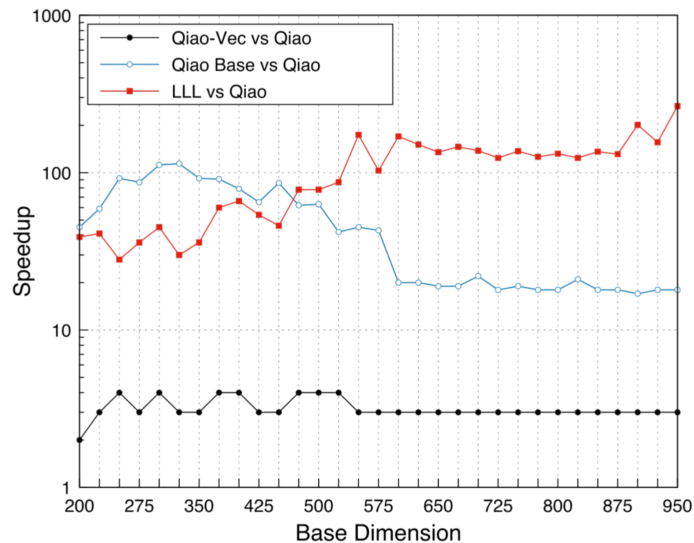


Figure 32: Speedups comparison between sequential LLL and Qiao's Jacobi method implementations in Ajtai-type bases.

4.1 PARALLEL VERSION

After identifying the most important optimizations, the Algorithm 2.10 lost a major part of its parallel properties based on matrix multiplications.

As previous mentioned, it is possible parallelize the Qiao's Jacobi method by reducing several sub-lattices at once with some limitations. In [Jeremic and Qiao (2014)], the order in which the vectors are reduced is changed to allow their parallel approach. However, it changes the output lattice and the flow of the algorithm. Section 4.2 will show how the order in which the vectors are reduced affects the quality of the reduced basis.

To analyze the impact of the *chess tournament ordering*, that was referred in section 2.2.5, we count the required number of sweeps for the algorithm to converge to a solution. Figure 33 illustrates the average number of sweeps out of 50 different GM lattices of the same dimension, where it illustrates the original algorithm as *'Qiao'*, and the algorithm with the *chess tournament ordering* as *'Qiao-Chess'*. We tried to perform the same test in Ajtai-type bases but the *'Qiao-Chess'* version did not terminate in reasonable time.

Figure 33 shows a significant increase of sweeps in the *'Qiao-Chess'* version, which will affect negatively the performance of the algorithm, since the number of sweeps is directly linked to the algorithm's performance.

The *Chess tournament ordering* seams to be the best strategy to maximize the number the vector reductions at a time and to avoid data hazards. Due to these, we used this ordering in the proposed parallel approaches.

Currently, the implementation in [Jeremic and Qiao (2014)] parallelizes $n/2$ vector reductions by creating $n/2$ threads. However, the creation of a fixed number of threads cannot be

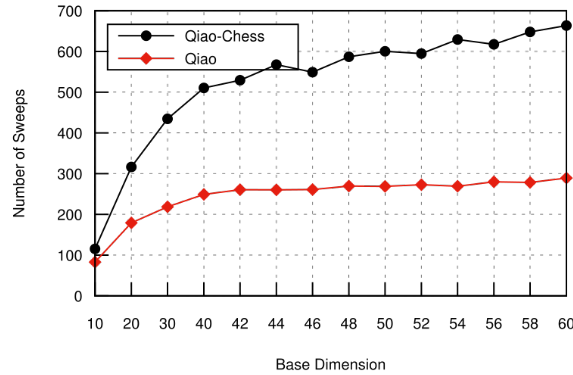


Figure 33: Number of necessary sweeps to converge to a solution in GM bases.

the best strategy to solve problems where the number of threads is substantially larger than the number of available Processor Units (PUs), i.e, a 800 dimension basis in a device with 32 PUs. The parallel strategies that we will describe below target an implementation that adapts the problem for a limited number of available PUs. Currently, the user can choose the number of threads that better adapt to a particular hardware or basis dimension.

To avoid data hazards, we need to create synchronization points and remove one of the previous proposed improvements. These are in the inner zone of the parallel implementation, which make it happen several times. Depending on the number of threads, these synchronization points are susceptible to create overhead. We reduced the number of synchronization points from 4 to 2 by switching between the column- or row-major less times. Therefore, we first compute all row-major computations and just then the column-major computations.

Other issue of the algorithm is the unbalanced workload, since some threads may have to do row and column swaps and others not. Unfortunately, there is no way to predict which threads have to do the swaps. We substantially reduced the unbalance with our last improvement since we avoided the integral matrix swaps. However, the evaluation tests of the subsection 4.1.1 do not include this improvement and perform integral matrix swaps.

Our parallel approach was not designed to have great scalability, since it only scales until $n/2$ threads. It is possible to use more threads but it will probably get worse results since it has to synchronize more threads in each synchronization point. As it was previously mentioned, they aim to execute large lattice basis in environments with a limited number of PUs.

It is possible to adopt a strategy that enhances the scalability by parallelizing each operation of each row or column. However, it creates too fine grain computations where the created overhead will not compensate the parallelization.

We created two parallel approaches that underline the previous discussion. The *chess tournament ordering* is used in both implementations, and they just differ in the order that

all $n/2$ vector reductions are done. The first approach is divided in two parts, where it performs all row-major order computations in first place, and just then it performs all column-major order computations. Each part is within a for-cycle that performs all $n/2$ pair combinations. To not compute the value of the variable q of each pair combination in each for-cycle, we cache its value in the first for-cycle into an array and reuse it in the second one.

The second approach parallelizes x of the $n/2$ pair combinations at once, by performing all row and column computations. To avoid data hazards, we implement synchronization points between the row and column computation and in the end of the column computation.

We used OpenMP to parallelize both implementations. In the first approach the synchronization points were implemented with the inherit properties of the directive *'#pragma omp for'*, which already implements a barrier. In the second approach the synchronization points are implemented with the directive *'#pragma omp barrier'*.

The first approach has the advantage of computing all row- or column-major order computations without synchronization points what could lead to better performances. However, it has time penalties to access the stored data on the second for-cycle. This approach leads to a higher memory consumption.

Unlike the first approach, the second one has a lower memory consumption, since it does not have to memoize data. However it may have to deal with more synchronization points, i.e., if the number of threads is lower than $n/2$, it has to do all the iteration more than once.

4.1.1 *Evaluation Results*

Figures 34 and 35 illustrate how the execution time changed with the increase of the number of threads. It uses GM lattice bases with the dimension between 10 and 60 and shows the average value out of 50 different bases. Depending in the lattice dimension, both figures do not illustrate the value for greater number of threads, i.e., dimension 10 with 16 threads. It happens because the execution times got worse because the number of threads is greater than $n/2$.

In every dimension of both figures, the test with 1 thread gets the best execution time. It means that overhead created to parallelize the algorithm takes a large portion of the final execution time, which means that the parallelization does not compensate for GM bases where it is used MP.

Note that these approaches may have different results in implementations with primitive data types, where it deals with different memory access patterns.

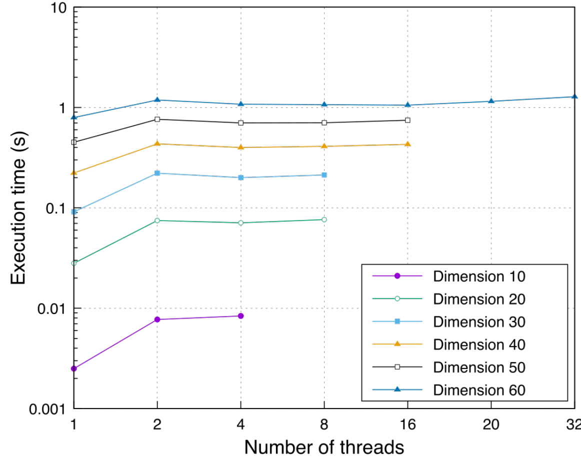


Figure 34: Execution times of first parallel approach in GM bases.

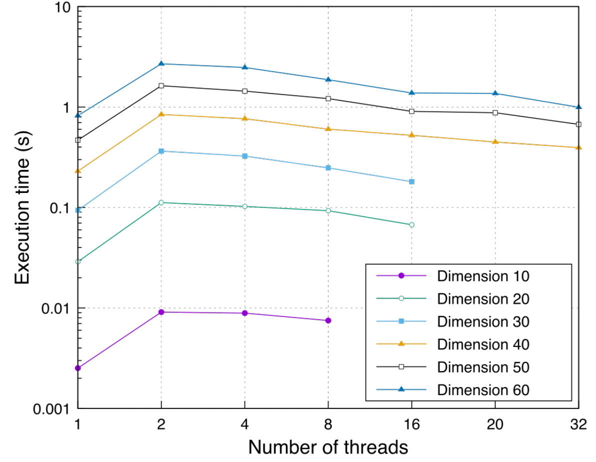


Figure 35: Execution times of second parallel approach in GM bases.

Figure 34 shows that every tests with 4 threads get better execution times than the same test with 2 threads, except for the dimension 10. Depending of the dimension of the basis the result get worse with more threads.

Figure 35 shows that the second approach scales better than the first one. However, we noticed that the overhead created from the execution of 1 to 2 threads is greater in the second approach. In general, the second approach get better results than the first one.

We did a small change in the first approach and create more fine grain tasks in order to get a better scalability, but the results got worse.

4.2 BASIS QUALITY ASSESSMENT

Goldstein and Mayer Matrices

We tested basis dimensions between 10 and 60 for GM bases. We present the average value out of 50 different bases for each dimension. The results are displayed in the Table 2. Depending on the algorithm, it gets very different results, which makes it hard to read in graphs. The enumerated metrics presented in the Table 2 were introduced in the subsection 2.2.6. These are the HF of a basis, the sequence of the GS norms, the norm of the last GS vector, the average of the norms of a basis, the product of the norms of a basis and the orthogonal defect of a basis, respectivately.

In general the LLL algorithm got the best results for every tests, with some exceptions, where the HF, the average and product of the norms and the orthogonal defect were always the best in all dimensions. In the Gram-Schmidt sequence, LLL gets worst in the dimension 10 but it is better in all other dimensions. *'Qiao-Chess'* got better results for the norm of the

Dimension	Algorithm	1.	2.	3.	4.	5.	6.
10	LLL	0,9E+00	9,6E+00	1,1E+03	1,2E+03	4,8E+30	1,3E-26
	Qiao	2,4E+04	-8,6E+03	1,4E+00	9,0E+04	3,7E+58	9,8E-25
	Qiao-Chess	1,6E+04	-8,3E+03	3,6E-04	9,2E+04	4,9E+56	9,8E-25
20	LLL	1,5E+00	-2,8E+01	8,2E+02	1,5E+03	2,6E+63	2,5E-56
	Qiao	5,3E+21	-1,9E+12	7,1E-05	3,2E+13	1,7E+294	5,3E-46
	Qiao-Chess	6,1E+25	-1,2E+14	3,3E-02	2,1E+15	1,0E+335	3,6E-44
30	LLL	2,5E+00	-3,7E+01	6,6E+02	1,8E+03	1,1E+98	3,7E-86
	Qiao	3,7E+45	-1,1E+24	4,9E+08	2,8E+25	4,9E+801	5,7E-64
	Qiao-Chess	2,1E+57	-2,6E+29	2,3E+14	6,9E+30	5,2E+971	1,3E-58
40	LLL	3,9E+00	-4,1E+01	5,3E+02	2,3E+03	3,6E+134	4,9E-116
	Qiao	5,5E+71	-8,8E+36	3,1E+23	3,3E+38	1,6E+1937	7,1E-81
	Qiao-Chess	1,3E+89	-4,7E+45	3,7E+31	1,8E+47	8,3E+3317	3,9E-72
50	LLL	6,0E+00	-4,5E+01	4,4E+02	2,8E+03	1,4E+173	6,0E-146
	Qiao	4,9E+99	-4,5E+50	2,6E+49	3,4E+66	2,2E+2683	4,3E-97
	Qiao-Chess	4,8E+124	-1,2E+63	2,3E+49	6,3E+64	8,3E+3317	1,6E-84
60	LLL	9,4E+00	-4,8+01	3,5E+02	3,5E+03	1,7E+214	6,9E-176
	Qiao	1,9E+128	-5,4E+64	9,6E+34	2,3E+52	1,0E4+087	7,4E-113
	Qiao-Chess	3,8E+161	-2,4E+81	8,9E+67	1,6E+83	8,5E+5086	3,0E-96

Table 2: Basis quality results for basis dimensions between 10 and 60 (GM bases).

last Gram-Schmidt vector, except for the dimension 10 and 20. *'Qiao'* algorithm always got results between LLL and *'Qiao-Chess'* algorithms but closer to *'Qiao-Chess'* algorithm.

The *'Qiao-Chess'* just differ from *'Qiao'* in the order of which vectors are reduced first. However it results in different lattice bases with worse quality. Therefore, the order in which the vectors are reduced is important, and it takes an important role in the quality of the reduced basis.

It is important to notice that the LLL results are very different from the *'Qiao'* and *'Qiao-Chess'* algorithms, which means that LLL algorithm is better for GM bases.

Ajtai-type Matrices

We tested basis dimensions between 200 and 950 for Ajtai-type bases. We just tested one sample of each dimension, because we were not able to get a basis generator of this type of basis. The results are illustrated in the next figures.

Figures 36, 37 and 38 show that Qiao's Jacobi method gets worse bases in lower dimensions. However, the same does not verify in higher dimensions, where Qiao's Jacobi method gets results better than LLL in all tests. In some tests Qiao's Jacobi method gets much better results than LLL, i.e., the HF for dimension 950 is more than 4.3 times better.

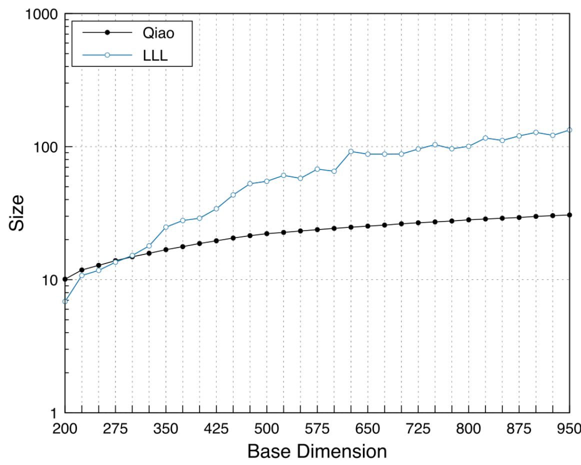


Figure 36: Hermite factor of output basis from LLL algorithm and Qiao's Jacobi method in Ajtai-type bases.

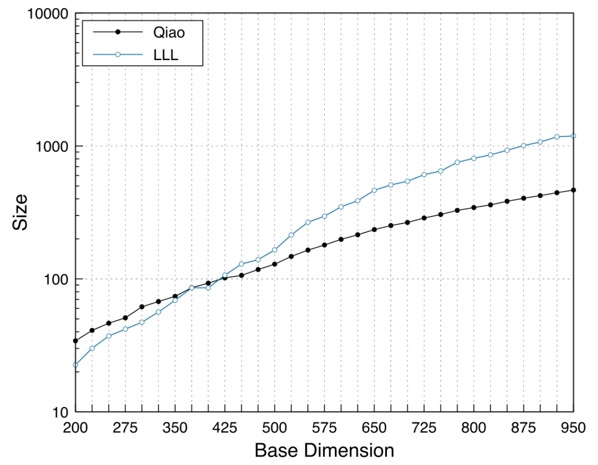


Figure 37: Average of the norms of the output basis from LLL algorithm and Qiao's Jacobi method in Ajtai-type bases.

Figure 39 shows that for most dimensions, the LLL algorithm and the Qiao's Jacobi method get the same result, but for some dimensions the Qiao's Jacobi method displays disappointing results.

The figures of the product of the norms and the orthogonal defect are not shown due to its high scale ($> 10^{300}$). However, both metrics have the same behaviour of the other graphs, where LLL results start better but end up worse.

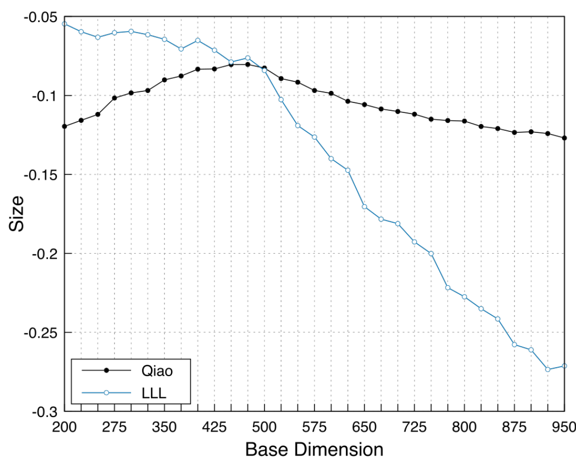


Figure 38: Sequence of the GS norms from LLL algorithm and Qiao's Jacobi method in Ajtai-type bases.

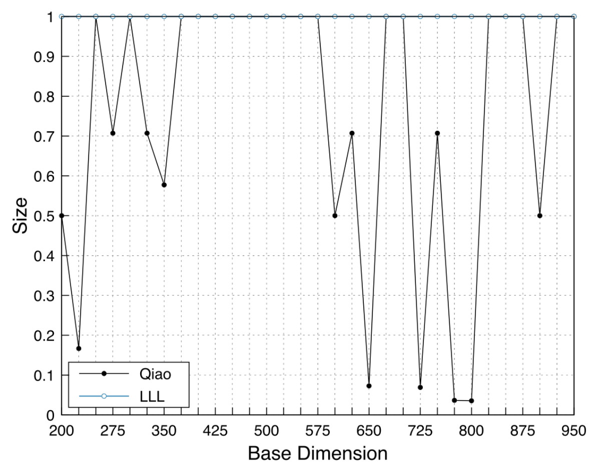


Figure 39: Last GS norms from LLL algorithm and Qiao's Jacobi method in Ajtai-type bases.

5

BKZ, LLL AND QIAO'S JACOBI METHOD

This chapter explores possible parallel approaches of well known lattice basis reduction algorithms such as LLL and BKZ. In addition, a variant of the BKZ algorithm in which the LLL algorithm was replaced by the Qiao's Jacobi method was implemented, and it was assessed the basis quality output of combining the Qiao's Jacobi method with LLL and BKZ algorithms. All the proposed implementations of this chapter are included in the LattBRed module.

5.1 TOWARDS PARALLEL APPROACHES

5.1.1 *Parallel LLL algorithm*

We implemented a floating-point version of the Algorithm 2.8 to do experiments in GM bases. To ensure the correctness of the implementation we compared its basis output and the output of the *'LLL_XD'* function of the NTL. However it did not work as expected, and to obtain the same output we had to compute the GS coefficients and norm after the size-reduction. The computation of the GS coefficients with the size-reduced vector minimize precision errors.

The first LLL implementation uses the NTL to handle with MP arithmetic. However, it took more time than expected in the conversion of the MP number to the *double* datatype, where the GMP's conversion is more efficient. This led to a second implementation where we used the GMP to handle with MP arithmetic.

By profiling our implementation, we noticed that 35% of the execution time is consumed by the computation of the GS coefficients and norms and 62% by the size-reduction. Therefore, it is imperative to parallelize these parts of the algorithm in order to get more performance.

After look into the GS algorithm we found an opportunity to parallelize this procedure by computing the inner products in parallel. This procedure is called several times during the algorithm execution, thus it is not a good approach to create and destroy a new parallel zone every time that the procedure is called, since it is one of the heaviest calls in paral-

lel programming. Therefore, we created a pool of POSIX threads, with *pthreads*, that are waiting for work, where the communication is handled with signals and mutexes.

Algorithm 5.1 Gram-Schmidt process for k

```
1:  function Proposed_GS_Bk(B, M, c, k)
2:      s = 0
3:      for j = 0 to k do
4:          M_{k,j} = InnerProduct(B_k, B_j)
5:      end for
6:      for j = 0 to k − 1 do
7:          tmp = M_{k,j}
8:          for i = 0 to j − 1 do
9:              tmp = tmp − M_{j,i} × buf_i
10:         end for
11:         buf_j = tmp
12:         M_{k,j} = tmp/c_j
13:         s = s + M_{k,j} × tmp
14:     end for
15:     c_k = M_{k,k} − s
16: end function
17: function GS_Bk(B, M, c, k)
18:     s = 0
19:     for j = 0 to k − 1 do
20:         tmp = InnerProduct(B_k, B_j)
21:         for i = 0 to j − 1 do
22:             tmp = tmp − M_{j,i} × buf_i
23:         end for
24:         buf_j = tmp
25:         M_{k,j} = tmp/c_j
26:         s = s + M_{k,j} × tmp
27:     end for
28:     c_k = InnerProduct(B_k, B_k) − s
29: end function
30:
31:
```

To minimize the number of synchronization points we performed the computation of all inner products at once. The Algorithm 5.1 shows both original algorithm and the proposed one, where **B** is the lattice basis, **M** is the matrix that handles the GS coefficients, **c** is the vector that stores the GS norms, k indicates the index of the vector that will be computed, and **buf** is a vector to store the previous results of tmp.

Table 3 shows the execution times of the sequential and parallel versions.The execution times of the parallel version show that the parallelization does not have a good scalability. The execution times increase a lot as the number of threads also increases because of huge amount of calls of the function *'Proposed_GS_Bk'*, where the synchronization points are heavier than the parallelized work. In addition, the new GS process reduces the execution time of the parallel version with just 1 thread, due to better memory accesses.

Number of threads	1	2	4	8
Sequential version	2.16			
Parallel version	1.86	4.50	6.89	10.62

Table 3: Execution times in seconds of the sequential and the parallel version up to 8 threads in a GM basis of dimension 50.

As the GS process, the size-redution is also executed several times. Thus, we followed the same strategy adopted for the GS parallelization, where it uses a pool of threads and resorts to signals and mutexes to communicate. However, we had an undesirable problem

during the parallelization, where the GMP reveals to have bug and it is not fully thread-safe[1]. This bug led our implementation to launch an error and consequently a segmentation fault during the GMP's computations.

5.1.2 *Parallel BKZ algorithm*

In order to better understand the BKZ algorithm, we started by implementing a sequential version of the Algorithm 2.9 where the *ENUM* function is based in the version presented by [Gama et al. (2010)], without the pruning component. The BKZ was implemented without MP arithmetic, and only the first LLL call handles with MP arithmetic.

Most parallel implementations of the BKZ algorithm, parallelize the *ENUM* function or the LLL algorithm. However, we propose a different approach, that consists in executing more than one *ENUM* at a time, where each one is searching in different projected basis.

The implementation starts by calling the LLL algorithm that handles with MP arithmetic, then it creates a set of threads defined by the user, where each thread executes a *worker*. Each *worker* allocates memory to handle with the standalone execution of the *ENUM* function, in which they just share the memory positions that store the main lattice basis and its GS coefficients and norms.

The shared memory can be accessed just by one thread, in order to avoid data hazards. Thus, we use mutexes to guarantee that only one thread accesses the shared memory at a time.

After the *ENUM* function returns a vector, the thread gets the lock and verifies if it can insert it in the basis. If the vector is inserted, it creates a linear dependency that is removed by performing the LLL algorithm. Afterwards, it copies the lattice basis and the GS coefficients and norms, release the lock and repeats the cycle.

This strategy allows two levels of parallelization, and allows parallelization in distributed systems. The first is achieved with our parallel approach where we can use several devices, resorting of MPI for example. Secondly, we have the usual parallel approaches in shared memory, e.g., *ENUM* function and LLL algorithm.

Unfortunately, we were not capable to perform the experiments of this approach, because the LLL algorithm went in infinite loop. We found the problem lately in our LLL implementation, which does not relax the GS coefficient condition. Usually this functionality is not implemented and is not referred in the found papers.

1 GMP Reentrancy - `https://gmplib.org/manual/Reentrancy.html`

5.2 BKZ W/ QIAO'S JACOBI METHOD

After the good results of the Qiao's Jacobi method in comparison to the LLL algorithm for Ajtai-type bases in the Chapter 4, the idea of replacing the LLL algorithm by the Qiao's Jacobi method emerged.

We started by replacing the first LLL call by the Qiao's Jacobi method. The replacement occurred without any problem, however we found out some barriers that are mainly related with the GS coefficients and norms. As it can be noticed, the Qiao's Jacobi method order the basis vectores by its euclidean norm but both the BKZ and the LLL algorithm order the basis vectores by the GS norm. Therefore, they create overhead to order the basis by the GS norms after the execution of the Qiao's Jacoby method.

The *ENUM* also needs the GS norms and coefficients to find the vector with the smallest GS norm. It leads the *ENUM* function to insert in the basis vectors with smaller GS norm but with greater euclidean norm, which leads the algorithm to search in two different directions. To minimize the fact of these two algorithms search in two different ways, we kept the LLL execution after the Qiao's Jacobi method. However, both implementations did not terminate in useful time and we finished their execution.

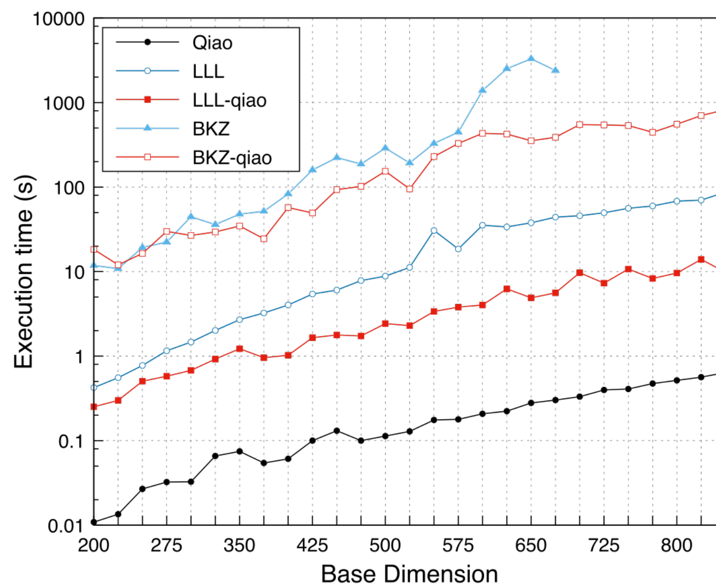


Figure 40: Execution times of the Qiao's Jacobi Method, the LLL and BKZ algorithms.

5.3 REDUCING L-REDUCED BASES

During some unofficial experiments in the section 5.2 we noticed that the LLL algorithm obtained shorter vectors by reducing a L-reduced basis instead of the original one. Thus, we performd experiments in Ajtai-type bases to compare the basis quality output of reducing

a L-reduced basis and its original with the LLL and BKZ algorithms. The BKZ algorithms uses the block size 20 in every experiments.

Figure 40 illustrates the execution time of 3 lattice basis reduction algorithms, where the *'LLL-qiao'*, *'BKZ-qiao'* reduce a L-reduce basis and the others reduce the correspondent original basis.

We introduced a set of metrics to measure the quality of the basis in the subsection 2.2.6. However, we just present some of them in the following figures, since the other metrics reflected similar results. The metric that measures the execution time of the SVP was not tested since the experiments were performed in large basis dimensions, and the current SVP-solvers do not terminate in useful time.

Every below experiments were performed in Ajtai-type bases, in which the *'BKZ-qiao'* and *'LLL-qiao'* reduce a L-reduced basis that was obtained by the Qiao's Jacobi method.

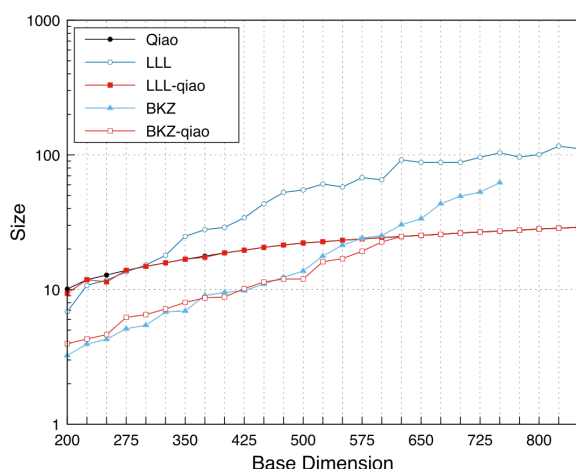


Figure 41: Hermite factor of output basis from LLL and BKZ algorithms and Qiao's Jacobi method.

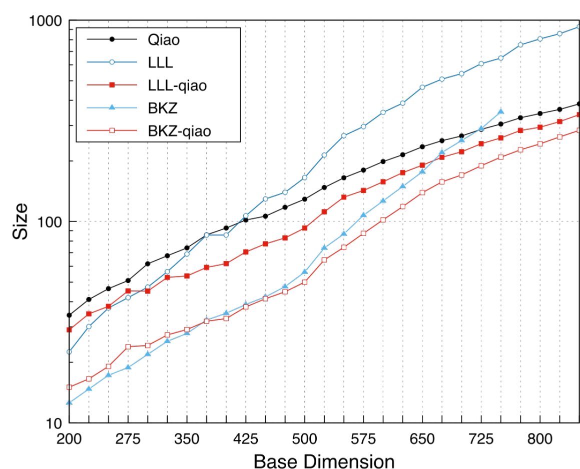


Figure 42: Average of the norms of output basis from LLL and BKZ algorithms and Qiao's Jacobi method.

Figure 41 shows that the LLL algorithm gets the worst HF for bases larger than dimension 300, however the BKZ algorithm obtains the best HF until the dimension 350, and get similar results to *'BKZ-qiao'* until the basis dimension 600. Above that dimension, *'BKZ-qiao'*, *'LLL-qiao'* and *'Qiao'* are overlapped. It also shows that *'LLL-qiao'* does not get better HF for any basis dimension, where both *'LLL-qiao'* and *'Qiao'* are overlapped.

As expected the Figure 42 shows that the BKZ algorithm get the best average norms, and the *'BKZ-qiao'* version obtain the best results from the dimension 375. However, the *'BKZ'* version starts to degrade the average of the norms from the dimension 475, and gets worse than the *'LLL-qiao'* and *'Qiao'* versions from the dimensions 675 and 725 respectively. Unlike the Figure 41, the *'LLL-qiao'* obtain better results than the *'Qiao'*. Once more, the worst results for large bases dimensions were obtained when using the LLL version.

Figure 43 shows similar results to the previous figures, where the BKZ algorithm gets the best results and the LLL gets the worst results for large basis dimensions. The *'LLL-qiao'* and *'LLL'* versions have disparate results by performing the same algorithm, where *'LLL-qiao'* obtains better results.

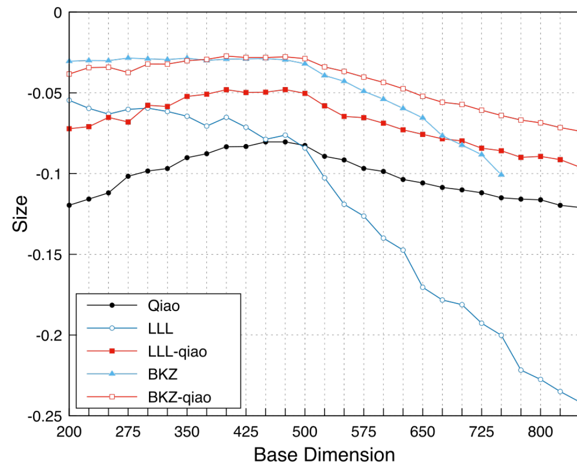


Figure 43: Sequence of the GS norms of the output bases from the LLL and BKZ algorithms and the Qiao's Jacobi method.

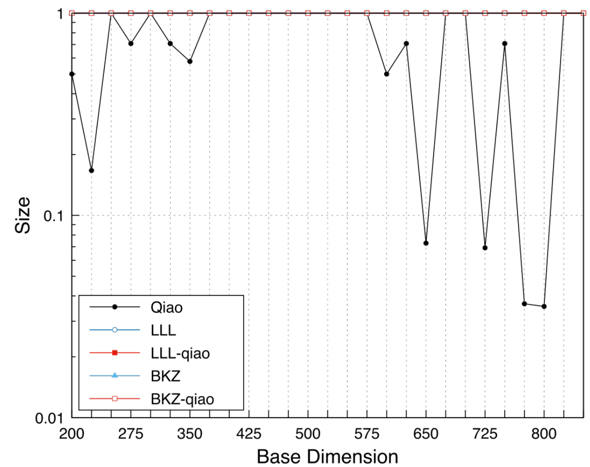


Figure 44: Last GS norms of output basis from LLL and BKZ algorithms and Qiao's Jacobi method.

Figure 44 is not conclusive, since most versions are overlapped, except the *'Qiao'* version for some basis dimensions.

Taking in account the execution times of the Figure 40 it is possible to conclude that the best output basis was obtained, in less time, by combining the Qiao's Jacobi method and the LLL or BKZ algorithms for large basis dimensions. However this was not always observed since the BKZ algorithm gets the best results for smaller dimensions.

Figure 40 shows that *'LLL-qiao'* always obtains better execution times than the *'LLL'*. Thus, it can be concluded that the combination of the Qiao's Jacobi method and the LLL algorithm outcomes in better basis outputs in less time, except for small dimensions.

6

CONCLUSIONS & FUTURE WORK

The dissertation is focused on lattice basis reduction algorithms and MP arithmetic, that is in some case studies one of its key requirements. Therefore, it is expected that the developed material creates value to the scientific community by proposing more efficient implementations or algorithms, and scrutinizing related issues.

An efficient integer MP module was developed that took advantage of the late microarchitectures for logical and shifts operations by using vector instructions. These operations obtained better execution times than the correspondent operation of the most known MP libraries.

Although the addition is one of the simplest operations, it plays a key role and deserved an investment to study several approaches. A vectorized version of this function in AVX-2 was implemented, without obtaining good performances. The poor execution time follows from the dependencies in the MP addition. To obtain better performances we implemented an assembly version that does two additions sequentially avoiding one overflow test per cycle. This last version obtained similar execution times compared to the other libraries, and in some cases it got the best execution time.

Less time was invested in the subtraction algorithm, which resulted in execution times not so good as expected. However, it is possible to implement the same assembly approach taken for the addition algorithm, which will result in more appealing execution times.

The MpIM's multiplication obtained better execution times than the NTL library until a certain number of digits. It happened because NTL library implements algorithms recommended for very large MP numbers, and we just implemented the long multiplication and the Karatsuba algorithms. However, the execution times of our implementations were worse than the execution times of the GMP and CLN libraries. Therefore, further research can be done to improve the MpIM's multiplication. The long multiplication can be improved by loop-unrolling several primitive multiplications in its intermediate steps, and by keeping good memory access patterns. By improving the long multiplication, it will automatically reflect in better execution times of the Karatsuba algorithm.

Due to time constraints, the MpIM's division had execution times that were not the expected, and it needs to be improved in follow researches.

Note that is hard to get better execution times than libraries such as GMP, where functions like addition, subtraction and the multiplication basecase have assembly versions for certain CPU that give significant speedups, mainly through avoiding function call overheads[1].

S. Qiao claimed that its Jacobi method has inherit parallel properties due to the required matrix computations. However, a good implementation of its algorithm must rely its computations in the Gram-Matrix. It removes most matrix multiplications of the algorithm, and its inherit parallel properties fade out. The dissertation presents a novel sequential implementation of the Qiao's Jacobi method that gets better results than the original implementation presented in [Qiao (2012)]. To accomplish good performances we avoided several computations and memory swaps and we developed two versions of this implementation: a MP version to handle GM bases and a version with primitive data types to handle Ajtai-type bases. Both versions have good have good speedups, but the version with primitive data types stands out with maximal speedups of 114 times and it reached speedups of 20 times with large basis dimensions.

The Qiao's Jacobi method does not performs well in GM bases, since these bases are very ill-conditioned, which leads the algorithm to make many sweeps, degrading its overall performance. Comparing its performance with the LLL algorithm, it takes longer to execute and its basis output is worse than the LLL algorithm for GM bases.

Unlike the previous version, the version with primitive data types performs well in Ajtai-type bases, and takes less time to execute than the LLL algorithm and the basis outputs are better than the LLL algorithm. The good execution times are related to the number of sweeps that the algorithm takes to get a L-reduced basis.

We also proposed two MP parallel approaches of the Qiao's Jacobi method that is best suited to different CPU and basis dimensions, allowing the choice of the number of threads. Despite it had better execution times than the parallel version proposed in [Jeremic and Qiao (2014)], it does not scale well. We implement a parallel MP version, but a version with primitive data type may have different results. It requires further research.

The parallel LLL algorithm does not achieved good execution times. It succeed because the overhead created in the synchronization points is greater than the parallelized work. However, we identified a more efficient GS process. Due to an bug in the GMP library we were not capable to test our parallel approach in a section of the LLL algorithm. It requires further research.

We suggest a parallel approach to a variant of the BKZ algorithm that parallelizes the search in the projected basis in two levels. The first level parallelizes by performing several searches in different projected bases. The second parallelization level is the most known and currently have several works developed. This first level should allows parallelizations in distributed environments, while the second level is recommended to parallelize in shared

1 GMP assembly basics - https://gmplib.org/manual/Assembly-Basics.html

memory systems. Unfortunately, we could not test this approach since our LLL implementation is not prepared to handle with particular special cases that succeed in this parallel approach.

Due to the good results of the Qiao's Jacobi method, we unsuccessfully tried to replace the LLL algorithm by the Qiao's Jacobi method in the BKZ algorithm. Unlike the LLL and BKZ algorithms, the Qiao's Jacobi method does not compute or order the basis by the GS coefficients and norms. Thus, The BKZ tries to get a completely different basis from the Qiao's Jacobi method. Due to this, our implementation did not terminate in useful time and it was terminated.

We studied if the combination of the Qiao's Jacobi method with the LLL and BKZ algorithms results in better output basis. We conclude that the combination of the Qiao's Jacobi method and the LLL algorithm outcomes with better output basis in less time, except for small basis dimensions, where the LLL algorithm get better output basis. In relation to the BKZ algorithm, it is just worth the combination of both algorithm for large basis dimensions.

BIBLIOGRAPHY

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

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. ISBN 0-89791-785-5. doi: 10.1145/237814.237838. URL http://doi.acm.org/10.1145/237814.237838.

Miklos Ajtai. The worst-case behavior of schnorr's algorithm approximating the shortest nonzero vector in a lattice. In *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, STOC '03, pages 396–406, New York, NY, USA, 2003. ACM. ISBN 1-58113-674-9. doi: 10.1145/780542.780602. URL http://doi.acm.org/10.1145/780542.780602.

Fábio Correia Artur Mariano and Christian Bischof. A vectorized, cache efficient lll implementation. *VECPAR - 12th International Meeting on High Performance Computing for Computational Science*, jun 2016. URL http://vecpar.fe.up.pt/2016/docs/VECPAR_2016_paper_18.pdf.

Paul Barrett. *Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor*, pages 311–323. Springer Berlin Heidelberg, Berlin, Heidelberg, 1987. ISBN 978-3-540-47721-1. doi: 10.1007/3-540-47721-7_24. URL http://dx.doi.org/10.1007/3-540-47721-7_24.

Daniel J. Bernstein. *Introduction to post-quantum cryptography*, pages 1–14. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-540-88702-7. doi: 10.1007/978-3-540-88702-7_1. URL http://dx.doi.org/10.1007/978-3-540-88702-7_1.

Marco Bodrato. *Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0*, pages 116–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-73074-3. doi: 10.1007/978-3-540-73074-3_10. URL http://dx.doi.org/10.1007/978-3-540-73074-3_10.

Andrew D Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951. URL http://qjmam.oxfordjournals.org/content/4/2/236.full.pdf.

Richard Brent and Paul Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, New York, NY, USA, 2010. ISBN 0521194695, 9780521194693.

Christoph Burnikel, Joachim Ziegler, Im Stadtwald, and D-Saarbrucken. Fast recursive division, 1998.

John William Scott Cassels. *An introduction to the geometry of numbers*. Springer Science & Business Media, 2012.

Giovanni Cesari and Roman Maeder. Performance analysis of the parallel karatsuba multiplication algorithm for distributed memory architectures. *Journal of Symbolic Computation*, 21(4):467–473, 1996. URL http://dx.doi.org/10.1006/jsco.1996.0026.

Bruce Char, Jeremy Johnson, David Saunders, and Andrew P. Wack. Some experiments with parallel bignum arithmetic. In *In Proc. First Internat. Symp. Parallel Symbolic Comput. PASCO '94*, pages 94–103. World Scientific Publishing, 1994. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.3845.

Yuanmi Chen and Phong Q Nguyen. BKZ 2.0: Better Lattice Security Estimates. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7073 LNCS, pages 1–20. 2011. ISBN 9783642253843. doi: 10.1007/978-3-642-25385-0_1. URL http://link.springer.com/10.1007/978-3-642-25385-0_1.

W.A. Chren. A new residue number system division algorithm. *Computers & Mathematics with Applications*, 19(7):13 – 29, 1990. ISSN 0898-1221. doi: http://dx.doi.org/10.1016/0898-1221(90)90190-U. URL http://www.sciencedirect.com/science/article/pii/089812219090190U.

Svyatoslav Covanov and Emmanuel Thomé. Fast arithmetic for faster integer multiplication. *CoRR*, abs/1502.02800, 2015. URL http://arxiv.org/abs/1502.02800.

Anindya De, Piyush P. Kurur, Chandan Saha, and Ramprasad Saptharishi. Fast integer multiplication using modular arithmetic. *CoRR*, abs/0801.1416, 2008. URL http://arxiv.org/abs/0801.1416.

David Defour and Florent de Dinechin. *Software Carry-Save: A Case Study for Instruction-Level Parallelism*, pages 207–214. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. ISBN 978-3-540-45145-7. doi: 10.1007/978-3-540-45145-7_18. URL http://dx.doi.org/10.1007/978-3-540-45145-7_18.

David Defour, Florent De Dinechin, et al. Software carry-save for fast multiple-precision algorithms. In *35th International Congress of Mathematical Software*, pages 29–40, 2002.

Zilin Du, Maria Eleftheriou, José E. Moreira, and Chee-Keng Yap. Hypergeometric functions in exact geometric computation. *Electr. Notes Theor. Comput. Sci.*, 66(1):53–64, 2002. doi: 10.1016/S1571-0661(04)80378-5. URL http://dx.doi.org/10.1016/S1571-0661(04)80378-5.

Barry S. Fagin. Large integer multiplication on hypercubes. *J. Parallel Distrib. Comput.*, 14(4):426–430, April 1992. ISSN 0743-7315. doi: 10.1016/0743-7315(92)90080-7. URL http://dx.doi.org/10.1016/0743-7315(92)90080-7.

Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2):13, 2007. doi: 10.1145/1236463.1236468. URL http://doi.acm.org/10.1145/1236463.1236468.

Martin Fürer. Faster integer multiplication. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 57–66, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-631-8. doi: 10.1145/1250790.1250800. URL http://doi.acm.org/10.1145/1250790.1250800.

Nicolas Gama and PhongQ. Nguyen. Predicting lattice reduction. In Nigel Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, volume 4965 of *Lecture Notes in Computer Science*, pages 31–51. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-78966-6. doi: 10.1007/978-3-540-78967-3_3. URL http://dx.doi.org/10.1007/978-3-540-78967-3_3.

Nicolas Gama, Phong Q Nguyen, and Oded Regev. Lattice Enumeration Using Extreme Pruning. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6110 LNCS, pages 257–278. 2010. ISBN 3642131891. doi: 10.1007/978-3-642-13190-5_13. URL http://link.springer.com/10.1007/978-3-642-13190-5_13.

Daniel Goldstein and Andrew Mayer. On the equidistribution of Hecke points. *Forum Mathematicum*, 15(2):165–189, jan 2003. ISSN 0933-7741. doi: 10.1515/form.2003.009.

Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996. ISBN 0-8018-5414-8.

Guillaume Hanrot and Damien Stehlé. Worst-Case Hermite-Korkine-Zolotarev Reduced Lattice Bases. jan 2008. URL http://arxiv.org/abs/0801.3331.

Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Analyzing blockwise lattice algorithms using dynamical systems. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 447–464. Springer

Berlin Heidelberg, 2011a. ISBN 978-3-642-22791-2. doi: 10.1007/978-3-642-22792-9_25. URL http://dx.doi.org/10.1007/978-3-642-22792-9_25.

Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Algorithms for the shortest and closest lattice vector problems. In YeowMeng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing, editors, *Coding and Cryptology*, volume 6639 of *Lecture Notes in Computer Science*, pages 159–190. Springer Berlin Heidelberg, 2011b. ISBN 978-3-642-20900-0. doi: 10.1007/978-3-642-20901-7_10. URL http://dx.doi.org/10.1007/978-3-642-20901-7_10.

William Bruce Hart. Efficient divide-and-conquer multiprecision integer division. In *Proceedings of the 2015 IEEE 22Nd Symposium on Computer Arithmetic*, ARITH '15, pages 90–95, Washington, DC, USA, 2015. IEEE Computer Society. ISBN 978-1-4799-8664-4. doi: 10.1109/ARITH.2015.19. URL http://dx.doi.org/10.1109/ARITH.2015.19.

David Harvey, Joris van der Hoeven, and Grégoire Lecerf. Even faster integer multiplication. *Journal of Complexity*, 36:1 – 30, 2016. ISSN 0885-064X. doi: http://dx.doi.org/10.1016/j.jco.2016.03.001. URL http://www.sciencedirect.com/science/article/pii/S0885064X16000182.

Karl Hasselström. *Fast Division of Large Integers: A Comparison of Algorithms*. 2003. URL http://bioinfo.ict.ac.cn/~dbu/AlgorithmCourses/Lectures/Hasselstrom2003.pdf.

Werner Hofschuster and Walter Krämer. *C-XSC 2.0 – A C++ Library for Extended Scientific Computing*, pages 15–35. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-24738-8. doi: 10.1007/978-3-540-24738-8_2. URL http://dx.doi.org/10.1007/978-3-540-24738-8_2.

Catalin Hritcu, Iulian Goriac, Raluca Gordan, and Elena Erbiceanu. Designing a multiprecision number theory library. *International Journal of Computing*, 2(3):44–48, 2014. ISSN 2312-5381. URL http://www.computingonline.net/index.php/computing/article/view/228.

Konstantin Isupov and Vladimir Knyazkov. *A Modular-Positional Computation Technique for Multiple-Precision Floating-Point Arithmetic*, pages 47–61. Springer International Publishing, Cham, 2015. ISBN 978-3-319-21909-7. doi: 10.1007/978-3-319-21909-7_5. URL http://dx.doi.org/10.1007/978-3-319-21909-7_5.

Leah H. Jamieson, Philip T. Mueller, and Howard Jay Siegel. Fft algorithms for simd parallel processing systems. *Journal of Parallel and Distributed Computing*, 3(1):48 – 71, 1986. ISSN 0743-7315. doi: http://dx.doi.org/10.1016/0743-7315(86)90027-4. URL http://www.sciencedirect.com/science/article/pii/0743731586900274.

Tudor Jebelean. An algorithm for exact division. *J. Symb. Comput.*, 15(2):169–180, February 1993. ISSN 0747-7171. doi: 10.1006/jsco.1993.1012. URL http://dx.doi.org/10.1006/jsco.1993.1012.

Tudor Jebelean. Exact Division with Karatsuba Complexity. RISC Report Series 96-31, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Schloss Hagenberg, 4232 Hagenberg, Austria, December 1996.

Tudor Jebelean. Practical integer division with karatsuba complexity. In *Proceedings of the 1997 International Symposium on Symbolic and Algebraic Computation*, ISSAC '97, pages 339–341, New York, NY, USA, 1997. ACM. ISBN 0-89791-875-4. doi: 10.1145/258726.258836. URL http://doi.acm.org/10.1145/258726.258836.

Filip Jeremic and Sanzheng Qiao. A parallel jacobi-type lattice basis reduction algorithm. 2014. URL http://www.cas.mcmaster.ca/~qiao/publications/JQ14.pdf.

S.Lennart Johnsson, Ching-Tien Ho, Michel Jacquemin, and Alan Ruttenberg. Computing fast fourier transforms on boolean cubes and related networks, 1988. URL http://dx.doi.org/10.1117/12.942036.

Erich Kaltofen and Markus A. Hitz. Integer division in residue number systems. *IEEE Trans. Comput.*, 44(8):983–989, August 1995. ISSN 0018-9340. doi: 10.1109/12.403714. URL http://dx.doi.org/10.1109/12.403714.

Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 193–206, New York, NY, USA, 1983. ACM. ISBN 0-89791-099-0. doi: 10.1145/800061.808749. URL http://doi.acm.org/10.1145/800061.808749.

A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences*, 145:293–294, 1962.

AnaTolii Alexeevich Karatsuba. The complexity of computations. *Proceedings of the Steklov Institute of Mathematics-Interperiodica Translation*, 211:169–183, 1995. URL http://www.ccas.ru/personal/karatsuba/divcen.pdf.

Miroslav Knezevic, Frederik Vercauteren, and Ingrid Verbauwhede. Faster interleaved modular multiplication based on barrett and montgomery reduction methods. *IEEE Transactions on Computers*, 59(12):1715–1721, 2010. URL http://dx.doi.org/10.1109/TC.2010.93.

Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. ISBN 0-201-89684-2. URL http://dl.acm.org/citation.cfm?id=270146.

Werner Krandick and Tudor Jebelean. Bidirectional exact integer division. *J. Symb. Comput.*, 21(4-6):441–455, June 1996. ISSN 0747-7171. doi: 10.1006/jsco.1996.0024. URL http://dx.doi.org/10.1006/jsco.1996.0024.

Wolfgang Kuechlin, David Lutz, and Nicholas Nevin. *Integer multiplication in PARSAC-2 on stock microprocessors*, pages 206–217. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991. ISBN 978-3-540-38436-6. doi: 10.1007/3-540-54522-0_109. URL http://dx.doi.org/10.1007/3-540-54522-0_109.

A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982a. ISSN 1432-1807. doi: 10.1007/BF01457454. URL http://dx.doi.org/10.1007/BF01457454.

A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982b. ISSN 1432-1807. doi: 10.1007/BF01457454. URL http://dx.doi.org/10.1007/BF01457454.

Artur Mariano. *High performance algorithms for lattice-based cryptanalysis*. PhD thesis, TU-Darmstadt, sep 2016.

Bartlett W Mel. A computational perspective. *Dendrites*, page 421, 2007. URL http://www.springer.com/us/book/9780387252827.

Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996. ISBN 0849385237.

Robert Moenck and Allan Borodin. Fast modular transforms via division. In *Switching and Automata Theory, 1972., IEEE Conference Record of 13th Annual Symposium on*, pages 90–96. IEEE, 1972. URL http://dx.doi.org/10.1109/SWAT.1972.5.

Niels Moller and Torbjorn Granlund. Improved division by invariant integers. *IEEE Trans. Comput.*, 60(2):165–175, February 2011. ISSN 0018-9340. doi: 10.1109/TC.2010.143. URL http://dx.doi.org/10.1109/TC.2010.143.

Phong Q. Nguên and Damien Stehlé. *Floating-Point LLL Revisited*, pages 215–233. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-32055-5. doi: 10.1007/11426639_13. URL http://dx.doi.org/10.1007/11426639_13.

Phong Q. Nguyen. *Hermite's Constant and Lattice Algorithms*, pages 19–69. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-02295-1. doi: 10.1007/978-3-642-02295-1_2. URL http://dx.doi.org/10.1007/978-3-642-02295-1_2.

Phong Q. Nguyen and Damien Stehlé. *LLL on the Average*, pages 238–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-36076-6. doi: 10.1007/11792086_18. URL http://dx.doi.org/10.1007/11792086_18.

Muhamad Nursalman, Arif Sasongko, Yusuf Kurniawan, et al. Improved generalizations of the karatsuba algorithm in gf (2 n). In *Advanced Informatics: Concept, Theory and Application (ICAICTA), 2014 International Conference of*, pages 185–190. IEEE, 2014. URL http://dx.doi.org/10.1109/ICAICTA.2014.7005938.

Amos Omondi and Benjamin Premkumar. *Residue number systems*. 2007. ISBN 9781860948664. doi: 10.1007/978-3-642-54649-5{_}3. URL http://cds.cern.ch/record/1701597.

Sanzheng Qiao. A jacobi method for lattice basis reduction. In *Engineering and Technology (S-CET), 2012 Spring Congress on*, pages 1–4. IEEE, 2012. URL http://dx.doi.org/10.1109/SCET.2012.6342057.

Nathalie Revol and Fabrice Rouillier. Motivations for an arbitrary precision interval arithmetic and the mpfi library. *Reliable Computing*, 11(4):275–290, 2005. ISSN 1573-1340. doi: 10.1007/s11155-005-6891-y. URL http://dx.doi.org/10.1007/s11155-005-6891-y.

Michael C. Ring. Mapm, a portable arbitrary precision math library in c. *C/C++ Users J.*, 19(11):28–38, November 2001. ISSN 1075-2838. URL http://dl.acm.org/citation.cfm?id=586191.586194.

C. P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theor. Comput. Sci.*, 53(2-3):201–224, August 1987. ISSN 0304-3975. doi: 10.1016/0304-3975(87)90064-8. URL http://dx.doi.org/10.1016/0304-3975(87)90064-8.

C. P. Schnorr and M. Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming*, 66(1):181–199, 1994. ISSN 1436-4646. doi: 10.1007/BF01581144. URL http://dx.doi.org/10.1007/BF01581144.

A. Schönhage and V. Strassen. Fast multiplication of large numbers. *Computing*, 7(3):281–292, 1971. ISSN 1436-5057. doi: 10.1007/BF02242355. URL http://dx.doi.org/10.1007/BF02242355.

M. J. Schulte, J. Omar, and E. E. Swartzlander. Optimal initial approximations for the newton-raphson division algorithm. *Computing*, 53(3):233–242, 1994. ISSN 1436-5057. doi: 10.1007/BF02307376. URL http://dx.doi.org/10.1007/BF02307376.

Mike Scott. Missing a trick: Karatsuba variations. 2016. URL http://eprint.iacr.org/2015/1247.pdf.

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. ISSN 0097-5397. doi: 10.1137/S0097539795293172. URL http://dx.doi.org/10.1137/S0097539795293172.

Victor Shoup. Ntl vs flint. We have compiled some benchmarks that compare the relative performance of NTL and FLINT on some fundamental benchmarks., aug 2016.

Carl Ludwig Siegel. *Lectures on the Geometry of Numbers*. Springer Science & Business Media, 2013.

Stål O. Aanderaa Stephen A. Cook. On the minimum computation time of functions. *Transactions of the American Mathematical Society*, 142:291–314, 1969. ISSN 00029947. URL http://www.jstor.org/stable/1995359.

Zhaofei Tian and Sanzheng Qiao. An enhanced jacobi method for lattice-reduction-aided mimo detection. In *Signal and Information Processing (ChinaSIP), 2013 IEEE China Summit & International Conference on*, pages 39–43. IEEE, 2013. URL http://dx.doi.org/10.1109/ChinaSIP.2013.6625293.

FL Tiplea, S Iftene, C Hritcu, I Goriac, RM Gordan, and E Erbiceanu. Mpnt: A multi-precision number theory package. number-theoretic algorithms (i). *Al. I. Cuza" University of Iasi, Faculty of Computer Science, Tech. Rep. TR*, pages 03–02, 2003. URL http://profs.info.uaic.ro/~tr/tr03-02.pdf.

Andrei L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3:714–716, 1963. ISSN 0197–6788.

Joachim von zur Gathen and Jamshid Shokrollahi. *Efficient FPGA-Based Karatsuba Multipliers for Polynomials over F2*, pages 359–369. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-33109-4. doi: 10.1007/11693383_25. URL http://dx.doi.org/10.1007/11693383_25.

Xiaohong Wang and Sanzheng Qiao. A parallel jacobi method for the takagi factorization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '02, June 24 - 27, 2002, Las Vegas, Nevada, USA, Volume 1*, pages 206–214, 2002.

D Wbben, Dominik Seethaler, J Jaldn, and Gerald Matz. Lattice reduction: a survey with applications in wireless communications. *EEE Signal Processing Magazine*, 28(3):70–91, 2011. URL http://dx.doi.org/10.1109/MSP.2010.938758.

André Weimerskirch and Christof Paar. Generalizations of the karatsuba algorithm for efficient implementations. *IACR Cryptology ePrint Archive*, 2006:224, 2006. URL http://eprint.iacr.org/2006/224.

Peiliang Xu. Experimental quality evaluation of lattice basis reduction methods for decorrelating low-dimensional integer least squares problems. *EURASIP Journal on Advances in*

Signal Processing, 2013:137, 2013. ISSN 16876172. doi: 10.1186/1687-6180-2013-137. URL http://asp.eurasipjournals.com/content/pdf/1687-6180-2013-137.pdf.