DIMACS Technical Report 2008-12
November 2008

A Parallel LLL using POSIX Threads

by

Werner Backes
Dept. of Computer Science
Stevens Institute of Technology
Hoboken, New Jersey 07030

Susanne Wetzel[1]
Dept. of Computer Science
Stevens Institute of Technology
Hoboken, New Jersey 07030

[1]Permanent Member

DIMACS is a collaborative project of Rutgers University, Princeton University, AT&T Labs–Research, Bell Labs, NEC Laboratories America and Telcordia Technologies, as well as affiliate members Avaya Labs, HP Labs, IBM Research, Microsoft Research, Stevens Institute of Technology, Georgia Institute of Technology and Rensselaer Polytechnic Institute. DIMACS was founded as an NSF Science and Technology Center.

ABSTRACT

In this paper we introduce a new parallel variant of the LLL lattice basis reduction algorithm. Lattice theory and in particular lattice basis reduction continues to play an integral role in cryptography. Not only does it provide effective cryptanalysis tools but it is also believed to bring about new cryptographic primitives that exhibit strong security even in the presence of quantum computers. In theory, many aspects of lattices are already well-understood. Yet, many practical aspects, like the performance of lattice basis reduction algorithms, are still under investigation.

In this paper, we introduce a new parallel lattice basis reduction algorithm that overcomes shortcomings of previously introduced algorithms. First and foremost, our new algorithm is based on the Schnorr-Euchner algorithm and as such is the first—to the best of our knowledge—to provide a parallel implementation for the Schnorr-Euchner algorithm. Second, using POSIX threads allows us to make effective use of today's multi-processor, multi-core computer architecture. Developing in a shared memory setting allows us to replace time consuming inter-process communication with synchronization points (barriers) and locks (mutexes). Our implementation of the parallel LLL is optimized for reducing high dimensional lattice bases with big entries that would require a multi-precision floating-point arithmetic to approximate the lattice basis if the original Schnorr-Euchner algorithm was used for the reduction. The reduction of these lattice bases is of great interest, e.g., for cryptanalyzing RSA. In experiments with sparse and dense lattice bases, experiments with our new parallel LLL show (compared to the non-parallel algorithm) a speed-up factor of about 1.75 for the 2-thread and close to factor 3 for the 4-thread version. The overhead of the parallel LLL decreases with increasing dimension of the lattice basis to less than 10% for the 2-thread and less than 15% for the 4-thread version.

1 Introduction

Lattice theory and in particular lattice basis reduction is of great importance in cryptography. Not only does it provide effective cryptanalysis tools but it is also believed to bring about new cryptographic primitives that exhibit strong security even in the presence of quantum computers [24, 31]. In theory, many aspects of lattices are already well-understood. On the other hand many practical aspects, like the performance of lattice basis reduction algorithms, are still under investigation.

Lattice basis reduction algorithms try to find a *good* basis, i.e., a basis representing the lattice where the base vectors are not only as orthogonal as possible to each other, but also as short as possible. The LLL algorithm introduced by Lenstra, Lenstra and Lovász in [20] was the first algorithm to allow an efficient computation of a fairly well-reduced lattice basis in theory but suffered from stability and performance issues in practice. In [32], Schnorr and Euchner introduced an efficient variant of the LLL algorithm, which could efficiently be used in practice, e.g., in cryptanalysis [32, 28, 29]. Since then, the main focus of research has been on improving the performance and stability of the algorithms (e.g., [11, 17, 18, 25, 26]). However, little to no progress has been made in the last few years with respect to parallel lattice basis reduction and making effective use of parallelization in practice. There are two main lines of previous work on parallel lattice basis reduction. The first was mainly based on the original LLL algorithm [35, 21, 15, 16, 36, 37]. Consequently, these solutions suffer from similar shortcomings in terms of stability and performance in practice as the original LLL algorithm. The second line of work is focused on vector computers [14, 13, 12, 38], an architecture which is quite different from mainstream compute servers.

In this paper, we present a new parallel LLL algorithm that overcomes both of these shortcomings. First and foremost, our new algorithm is based on the Schnorr-Euchner algorithm and as such is the first—to the best of our knowledge—to provide a parallel implementation for the Schnorr-Euchner algorithm. Second, using POSIX threads [6, 34] allows us to make effective use of today's multi-processor, multi-core computer architecture. Developing in a shared memory setting allows us to replace time consuming inter-process communication with synchronization points (barriers) and locks (mutexes). Our implementation of the parallel LLL is optimized for reducing high dimensional lattice bases with big entries that would require a multi-precision floating-point arithmetic to approximate the lattice basis if the original Schnorr-Euchner algorithm was used for the reduction. The reduction of these lattice bases is of great interest, e.g., for cryptanalyzing RSA [22, 23, 5, 4]. In experiments with sparse and dense lattice bases, experiments with our new parallel LLL show (compared to the non-parallel algorithm) a speed-up factor of about 1.75 for the 2-thread and close to factor 3 for the 4-thread version. The overhead of the parallel LLL decreases with increasing dimension of the lattice basis to less than 10% for the 2-thread and less than 15% for the 4-thread version.

Outline: Section 2 provides the definitions and notations used in the remainder of the paper and introduces the Schnorr-Euchner algorithm. The main contributions of this paper are in Section 3. It describes our new algorithm, i.e., it shows in detail how the Schnorr-Euchner algorithm can be parallelized by using threads with locks and barriers. Section 4 introduces knapsack and unimodu-

lar lattice bases, discusses the parameters that control the parallel LLL reduction, and provides an analysis of our experiments. The paper closes with a discussion on directions for future work.

2 Preliminaries

A *lattice* $L \subset \mathbb{R}^n$ is an additive discrete subgroup of $\mathbb{R}^n$ such that $L = \left\{\sum_{i=1}^{k} x_i \underline{b}_i | x_i \in \mathbb{Z}, 1 \leq i \leq k\right\}$ with linear independent vectors $\underline{b}_1, \ldots, \underline{b}_k \in \mathbb{R}^n$ ($k \leq n$). $B = (\underline{b}_1, \ldots, \underline{b}_k) \in \mathbb{R}^{n \times k}$ is the *lattice basis* of L with dimension k. A lattice may have an infinite number of bases. Different bases B and B' for the same lattice L can be transformed into each other by means of a *unimodular transformation*, i.e., $B' = BU$ with $U \in \mathbb{Z}^{n \times k}$ and $|\det U| = 1$. Typical unimodular transformations are the exchange of two base vectors—referred to as swap—or the adding of an integral multiple of one base vector to another one—generally referred to as translation. The *determinant* $\det(L) = |\det(B^T B)|^{\frac{1}{2}}$ of a lattice is an invariant. The Hadamard inequality $\det(L) \leq \prod_{i=1}^{k} \|\underline{b}_i\|$ (where $\|.\|$ denotes the Euclidean length of a vector) gives an upper bound for the determinant of the lattice. Equality holds if B is an orthogonal basis. The *orthogonalization* $B^* = (\underline{b}_1^*, \ldots, \underline{b}_k^*)$ of a lattice basis $B = (\underline{b}_1, \ldots, \underline{b}_k) \in \mathbb{R}^{n \times k}$ can be computed by means of the Gram-Schmidt method: $\underline{b}_1^* = \underline{b}_1$, $\underline{b}_i^* = \underline{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \underline{b}_j^*$ for $2 \leq i \leq k$ where $\mu_{i,j} = \frac{\langle \underline{b}_i, \underline{b}_j^* \rangle}{\|\underline{b}_j^*\|}$ for $1 \leq j < i \leq k$ and $\langle ., . \rangle$ defines the scalar product of two vectors. It is important to note that for a lattice $L \subset \mathbb{R}^n$ with basis $B = (\underline{b}_1, \ldots, \underline{b}_k) \in \mathbb{R}^{n \times k}$ a vector $\underline{b}_i^*$ of the orthogonalization $B^* = (\underline{b}_1^*, \ldots, \underline{b}_k^*) \in \mathbb{R}^{n \times k}$ is not necessarily in L. Furthermore, computing the orthogonalization B^* of a lattice basis using the Gram-Schmidt method strongly depends on the order of the basis vector of the lattice basis B. The *defect* of a lattice basis $B = (\underline{b}_1, \ldots, \underline{b}_k) \in \mathbb{R}^{n \times k}$ defined as $\mathrm{dft}(B) = \frac{\prod_{i=1}^{n} \|\underline{b}_i\|}{\det(L)}$ allows one to compare the quality of different bases. Generally, $\mathrm{dft}(B) \geq 1$ and $\mathrm{dft}(B) = 1$ for an orthogonal basis. The goal of lattice basis reduction is to determine a basis with as small a defect as possible. That is, for a lattice $L \subset \mathbb{R}^n$ with bases B and $B' \in \mathbb{R}^{n \times k}$, B' is better reduced than B if $\mathrm{dft}(B') < \mathrm{dft}(B)$. The most well-known and most-widely used lattice basis reduction method is the LLL reduction method [20]:

Definition 1 *For a lattice $L \subseteq \mathbb{Z}^n$ with basis $B = (\underline{b}_1, \ldots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$, corresponding Gram-Schmidt orthogonalization $B^* = (\underline{b}_1^*, \ldots, \underline{b}_k^*) \in \mathbb{Z}^{n \times k}$ and coefficients $\mu_{i,j}$ ($1 \leq j < i \leq k$), the basis B is LLL-reduced if (1) $|\mu_{i,j}| \leq \frac{1}{2}$ with $1 \leq j < i \leq k$ and (2) $\|\underline{b}_i^* + \mu_{i,i-1}\underline{b}_{i-1}^*\|^2 \geq y\|\underline{b}_{i-1}^*\|^2$ for $1 < i \leq k$.*

The reduction parameter y may arbitrarily be chosen in $\left(\frac{1}{4}, 1\right)$. Condition (1) is generally referred to as size-reduction [7, 30]. The Schnorr-Euchner algorithm [32, 2] allows for an efficient computation of an LLL-reduced lattice basis in practice. By and large, Algorithm 1 is the original Schnorr-Euchner algorithm. In order to make LLL reduction practical, the Schnorr-Euchner algorithm uses floating-point approximations of vectors and the basis (APPROX_BASIS and APPROX_VECTOR) in Lines `(1)` and `(23)`. For stability reasons, this in turn requires that the Schnorr-Euchner algorithm includes suitable measures in the form of correction steps (see [32] for details). These corrections include either the computation of exact scalar products (Line `(6)`)

as part of the Gram-Schmidt orthogonalization or a step-back (Line (29)) due to a large μ_{ij} used as part of the size-reduction (Line (18)). In order to prevent the corruption of the lattice, the Schnorr-Euchner algorithm uses an exact data type is used to modify the actual lattice basis (Line (17)). (In Algorithm 1, p denotes the bit precision of the data type used to approximate the lattice basis.)

A modification for computing the Gram-Schmidt coefficients was introduced in [26] which was shown to allow for an increased accuracy of the orthogonalization. As part of our work, we adapted these modifications to the Schnorr-Euchner algorithm (Algorithm 1, Lines (3) - (13)). A second modification (originally introduced in [26]) is the checking of Condition (2) (Definition 1) followed by possibly swapping the respective lattice basis vectors (Algorithm 1, Lines (34) - (46)). We introduced both modifications to increase the overall performance and stability of the original Schnorr-Euchner algorithm.

Algorithm 1:

INPUT: Lattice basis $B = (\underline{b}_1, \ldots, \underline{b}_k) \in \mathbb{Z}^{n \times k}$
OUTPUT: LLL-reduced lattice basis B

(1) APPROX_BASIS(B′,B)
(2) **while** ($i \leq k$) **do**
(3) $\mu_{ii} = 1$, $R_{ii} = \|\underline{b}'_i\|$, $S_1 = R_{ii}$
(4) **for** ($1 \leq j < i$) **do**
(5) **if** ($|\langle \underline{b}'_i, \underline{b}'_j \rangle| < 2^{\frac{p}{2}} \|\underline{b}'_i\| \|\underline{b}'_j\|$) **then**
(6) $R_{ij} = \text{APPROX\_VALUE}(\langle \underline{b}_i, \underline{b}_j \rangle)$
(7) **else**
(8) $R_{ij} = \langle \underline{b}'_i, \underline{b}'_j \rangle$
(9) $R_{ij} = R_{ij} - \sum_{m=1}^{j-1} R_{im}\mu_{jm}$
(10) $\mu_{ij} = \frac{R_{ij}}{R_{jj}}$
(11) $R_{ii} = R_{ii} - R_{ij}\mu_{ij}$
(12) $S_{j+1} = R_{ii}$
(13) **od**
(14) **for** ($i > j \geq 1$) **do**
(15) **if** ($|\mu_{i,j}| > \frac{1}{2}$) **then**
(16) $F_r = true$
(17) $b_i = b_i - \lceil \mu_{ij} \rfloor b_j$
(18) **if** ($|\mu_{ij}| > 2^{\frac{p}{2}}$) **then**
(19) $F_c = true$
(20) **for** ($1 \leq m \leq j$) **do**
(21) $\mu_{im} = \mu_{im} - \lceil \mu_{ij} \rfloor \mu_{jm}$
(22) **fi**
(23) **od**
(24) **if** ($F_r = true$) **then**
(25) APPROX_VECTOR($\underline{b}'_i, \underline{b}_i$)
(26) **if** ($F_c = false \wedge F_r = true$) **then**
(27) RECOMPUTE_R$_{ij}$() [1]
(28) $F_r = false$
(29) **if** ($F_c = true$) **then**
(30) $i = \max(i-1, 2)$
(31) $F_c = false$
(32) **else**
(33) $i' = i$
(34) **while** ($(i > 1) \wedge (yR_{i-1,i-1} > S_{i-1})$) **do**
(35) $b_i \leftrightarrow b_{i-1}$
(36) SWAP($\underline{b}_i, \underline{b}_{i-1}$)
(37) SWAP($\underline{b}'_i, \underline{b}'_{i-1}$)
(38) $i = i - 1$
(39) **od**
(40) **if** ($i \neq i'$) **then**
(41) **if** ($i = 1$) **then**
(42) $R_{11} = \|\underline{b}'_1\|$
(43) $i = 2$
(44) **fi**
(45) **else**
(46) $i = i + 1$
(47) **od**

[1] RECOMPUTE_R$_{ij}$ in Line (27) performs the same operations to recompute R_{ij} as shown in Lines (3) - (13).

3 Parallel LLL Reduction using POSIX Threads

On a single computer system, programs can be parallelized by using multiple processes or threads [6, 34]. Threads can be viewed as a kind of lightweight or stripped-down process. The operating system can easily switch between multiple threads within a process because these threads share the same address space. Processes, on the other hand, do not share memory and therefore have to communicate through shared files or communication channels. Thus, only a limited amount of sharing between processes is practical. These characteristics make threads the optimal choice for the parallelization of the LLL reduction algorithm due to the amount of data that has to be shared in the course of the reduction process. In the following, our work is based on the use of POSIX threads as they offer a standardized application programming interface (API) for many different operating systems [6, 34]. We are using the two techniques *mutex* and *barrier* for the synchronization of critical memory access in the threads. A mutex[2] is a mutual-exclusion interface that allows to ensure that only one thread at a time is accessing critical data by setting (locking) the mutex on entering a critical code section and releasing (unlocking) it upon completion. A thread cannot acquire a mutex that is locked (i.e., owned) by another thread, but has to wait until the owner of the mutex unlocks it. Barriers are used to ensure that a number of threads that cooperate on a parallel computation wait for each other at a specific point in the algorithm before any of them is allowed to continue with further computations. These constructs seem to allow to parallelize arbitrary code as long as enough barriers and mutexes are used. However, the excessive use of barriers and mutexes has a negative effect on the overall running time. This is due to the fact that the higher the number of mutexes and barriers, the higher is the probability of threads running into situations where they have to wait for each other or wait for a mutex to be unlocked. Consequently, it is crucial to minimize the use of these constructs. This directly implies the need to identify as many independent (concurrent) code sequences as possible in order to allow for an efficient parallelization.

Using barriers and mutexes allows us to implement a new parallel lattice basis reduction algorithm which is well-suited for reducing bases of high dimensions and with large entries. This new parallel LLL is based on the Schnorr-Euchner algorithm (see Algorithm 1). The performance and limitations of the Schnorr-Euchner algorithm are dependent on the data type used for the approximation of the lattice basis. In case of machine-type doubles, the reduction algorithm performs fairly well [32, 33, 37, 1, 2]. Previous experiments [2, 3] revealed a strong connection between the number of exact scalar products, reduction steps, and the running time of the algorithm. The reduction time is dominated by the operations using a long integer arithmetic. However, a major drawback in using doubles for the approximation of the lattice basis is the limitation in terms of dimension and bit length of lattice basis entries due to the fixed size of the double data type. One can avoid these restrictions by using a less efficient multi-precision floating point arithmetic. The reduction of lattice bases in high dimensions and entries of high bit length (which requires the use of a multi-precision floating point arithmetic for the approximation) are of interest in various context, e.g., for certain attacks on RSA [22, 23, 5, 4]. Using a multi-precision floating-point arith-

[2]The word mutex is derived from mutually exclusive.

metic to approximate the lattice basis changes the running time behavior of the Schnorr-Euchner reduction algorithm dramatically. In this case, the operations on the approximate data type, e.g., in the Gram-Schmidt orthogonalization, are a major contributor to the overall running time of the Schnorr-Euchner algorithm. Thus, the main challenge is to parallelize these computations despite existing dependencies. Yet, operations on the exact data type (long integer arithmetic) are vector operations that can be parallelize more easily [12, 15, 16, 21, 35].

In striving to parallelize the reduction process, the dependencies within the Schnorr-Euchner algorithm (Algorithm 1) force us to keep the main structure of the algorithm. That is, one iteration of the main loop (Lines `(2) - (47)`) will be performed at the time. Furthermore, the overall structure of the main loop remains intact. It is possible to identify three main parts in the main loop, i.e, the orthogonalization (Lines `(3) - (13)`), the size-reduction (Lines `(14) - (23)`), and the condition-check part (Lines `(34) - (46)`). The order of these parts and the computations in between cannot be modified, i.e., in every iteration of the main loop, the computation of the orthogonalization has to be completed before one can do to the size-reduction and finally the checking of second LLL condition (Definition 1) after the size-reduction is finished. On the other hand, we show that the computations within each part can be modified or rearranged and eventually can be parallelized efficiently. The main contribution in this paper is in developing methods that allow to perform the computations of each part in parallel. A major challenge of this approach is to find means that balance computations well among all threads for all parallel parts in order to minimize the waiting times at the barrier in between these parts. It is crucial to minimize the amount of computations that cannot be parallelized, because they limit the maximum speed-up factor of the parallel algorithm according to Amdahl's law [6].

In our implementation of the parallel LLL, we were able to parallelize the orthogonalization and the size-reduction part. We are computing the scalar products separately from the orthogonalization to achieve a better overall balance. Unfortunately, the condition-check part cannot be parallelized. However, this is not a major drawback due to the small amount of computations that have to be performed in that part. In the following, we are detailing the concepts behind the parallelization of each part including rearrangements and modifications of the non-parallel computations these parts are based on.

3.1 Scalar Product Part

The computation of scalar products and exact scalar products in the course of the orthogonalization (Algorithm 1, Lines `(3) - (13)`) does not depend on prior computations within an iteration of the main loop. We therefore can divide the loop in Lines `(4) - (13)` into two loops, one that first computes the scalar products (Lines `(5) - (8)`) and a second that afterwards computes the remainder of the orthogonalization (Lines `(9) - (12)`). Thus, the computation of scalar products can be parallelized separately from the remainder of the orthogonalization. The major challenge in balancing the computation of scalar products is the need for computing exact scalar products under certain conditions for stability reasons. This means that assigning each thread a distinct, equal-sized segment of the iterations of the loop for the computation of the scalar products does

not guarantee an equally distributed computation. Since there is no simple and efficient way to find a suitable partitioning, we instead divide iterations of the loop for the computation of scalar products and exact scalar products into small, distinct segments of size s_{sp}. Every thread requests a new segment as soon as it finishes the computation of its previous segment. This way, computationally intensive segments do not negatively impact the overall balance. We are using a shared segment position counter *sl* that is protected by a mutex mechanism to assign every thread a distinct segment. For each thread t as part of our parallel LLL algorithm the computation of scalar products looks as follows.[3]

Scalar Product – Thread$_t$

```
(1)  s = start_t, e = end_t
(2)  while (s ≤ i) do
(3)    if (e > i) then
(4)      e = i
(5)    for (s ≤ j < e) do
(6)      if (|⟨b'_i, b'_j⟩| < 2^(p/2) ||b'_i|| ||b'_j||) then
(7)        R_ij = APPROX_VALUE(⟨b_i, b_j⟩)
(8)      else
(9)        R_ij = ⟨b_i', b'_j⟩
(10)   od
(11)   MUTEX_LOCK(l_1)
(12)   s = sl,  sl = sl + s_sp,  e = sl
(13)   MUTEX_UNLOCK(l_1)
(14) od
```

The locking mechanism (**MUTEX_LOCK** and **MUTEX_UNLOCK**) is placed around the updates for the segment position counter, which is always increased by s_{sp} to ensure that no two threads are processing the same segment and therefore are accessing non-overlapping entries in the R-matrix. The initial segment for a thread is given by $start_t$ and end_t, and the starting value for *sl* is determined by the maximum of the end_t.

3.2 Orthogonalization Part

Given the parallel scalar product computation, Algorithm 3 shows the remaining part of the orthogonalization. To increase the visibility of the dependency issues in the computation of R_{ij}, we replaced the sum $\sum_{m=1}^{j-1} R_{im}\mu_{jm}$ of Line (9) in the Schnorr-Euchner algorithm with the appropriate loop that is used in practice. One can clearly see (Algorithm 3, Lines (2) - (3)) that all R_{im} (with $1 \leq m < j$) are needed before one can compute R_{ij}. It is important to note that all μ_{jm} (with $1 \leq m < j$) have already been computed. As is, the orthogonalization in Algorithm 3 is hard

[3] In outlining our multi-threaded programs, we distinguish between variables that are local for every thread and variables that are shared among all threads. Local variables are highlighted by the use of a different font (e.g., `local` vs. *shared*).

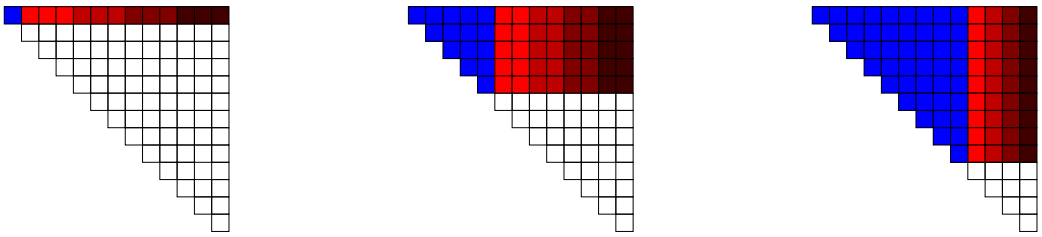

Figure 1: Parallel (each shade of red represents one thread) and non-parallel (blue) computations for main and helper threads

to parallelize. But a parallelized computation of R_{ij} is the key to successfully balance the work among all threads. With Algorithm 4 we have developed a modification of Algorithm 3 that allows to take advantage of the already computed μ_{jm}. Algorithm 4 introduces an additional value r_j to help with the computation of R_{ij}. The final value of r_j is in fact $\sum_{m=1}^{j-1} R_{im}\mu_{jm}$ known from the Schnorr-Euchner algorithm, but the value is computed in a different, more parallel-friendly fashion. Intuitively speaking, instead of computing the sum horizontally like in the Schnorr-Euchner algorithm, Algorithm 4 uses a vertical approach and updates the sum in r_j in phases, which eventually allows for a parallelization of the computation of r_j.

Algorithm 3:

(1) **for** $(1 \leq j < i)$ **do**
(2) **for** $(1 \leq m < j)$ **do**
(3) $R_{ij} = R_{ij} - R_{im}\mu_{jm}$
(4) $\mu_{ij} = \frac{R_{ij}}{Rjj}$
(5) $R_{ii} = R_{ii} - R_{ij}\mu_{ij}$
(6) $S_{j+1} = R_{ii}$

Algorithm 4:

(1) **for** $(1 \leq j < i)$ **do**
(2) **for** $(j \leq l < i)$ **do**
(3) $r_l = r_l + R_{i,j-1}\mu_{l,j-1}$
(4) $R_{ij} = R_{ij} - r_j$
(5) $\mu_{ij} = \frac{R_{ij}}{Rjj}$
(6) $R_{ii} = R_{ii} - R_{ij}\mu_{ij}$
(7) $S_{j+1} = R_{ii}$

In the following, we are now using Algorithm 4 as starting point for describing our parallelization of the orthogonalization. In particular, we detail our new methods on how to compute the r_j (Algorithm 4, Lines (2) - (3)) using multiple threads. A straight-forward approach to parallelize Algorithm 4 would be to put barriers around the computation of r_j and splitting up the iterations of the inner loop (Algorithm 4, Lines (2) - (3)) in between to be computed by all threads. The barriers allow one of the threads to compute R_{ij}, R_{ii}, μ_{ij} and S_{j+1} correctly, while the others wait for the next iteration of the outer loop (Algorithm 4, Lines (1) - (7)) in the orthogonalization. However, the excessive use of barriers is a major drawback which makes this parallelization attempt unusable in practice. Figure 1 shows the intuition for our new approach for the parallelization of the orthogonalization. A main (or control) thread computes the necessary non-parallelizable computations (blue) first, and then distributes the parallel computations (shades of red) among all threads. Obviously, the size of the parallel part depends on the size of the non-parallel part, and it is crucial to find an optimal balance the two. Increasing the non-parallel part decreases the number of barriers. However, at a certain point this may also decrease the workload per thread.

Orthogonalization – Thread$_1$ (main)

```
(1)  j = 0
(2)  while (j < i) do
(3)    s = j, m = 0
(4)    while (m < s_o ∧ j < i) do
(5)      for (s ≤ l < j) do
(6)        r_j = r_j − R_il μ_jl
(7)      R_ij = R_ij − r_j
(8)      μ_ij = R_ij / R_jj
(9)      R_ii = R_ii − R_ij μ_ij
(10)     S_{j+1} = R_ii
(11)     m = m + 1, j = j + 1
(12)   od
(13)   COMPUTE_SPLIT_VALUES_1(split)
(14)   BARRIER_WAIT(b_1)
(15)   for (j ≤ l < split_1) do
(16)     for (s ≤ m < j) do
(17)       r_l = r_l − R_im μ_lm
(18)   od
```

Orthogonalization – Thread$_t$

```
(1)  e = 0
(2)  while (e < i) do
(3)    s = e, e = e + s_o
(4)    if (e > i) then
(5)      e = i
(6)    BARRIER_WAIT(b_1)
(7)    for (split_t ≤ l < split_{t+1}) do
(8)      for (s ≤ m < e) do
(9)        r_l = r_l − R_im μ_lm
(10) od
```

In every iteration of the loop in Lines (4) - (12), the main thread computes only a small segment of size s_o of the orthogonalization. The size of this segment determines the number of barriers (function **BARRIER_WAIT**) and the amount of work that can be computed in parallel (Thread$_1$, Lines (15) - (17) and Thread$_t$, Lines (7) - (9)) in each iteration. Further details on the parameter choices and their effect on the computational balance are discussed in Section 4.2. Compared to the straight-forward parallelization idea, our new approach now uses one instead of two barriers for each iteration of the outer loops (Thread$_1$, Lines (2) - (18) and Thread$_t$, Lines (2) - (10)). Computing small segments of the orthogonalization significantly reduces the number of iterations of the outer loops and number of barriers needed for the orthogonalization. It is important to note that although not explicitly mentioned here in this paper (to improve readability), we have implemented measures to ensure that variables are not unintentionally overwritten by the subsequent loop iterations. The function COMPUTE_SPLIT_VALUES$_1$ computes the $split_t$ values that are responsible for balancing the parallel computation of r_l among all threads. In addition, the splitting has to ensure that the main thread computes the necessary r_l for the next iteration of its outer loop. The splitting among all threads is not necessarily even.

3.3 Size-Reduction Part

In the non-parallel version of the Schnorr-Euchner algorithm, the size-reduction (Algorithm 5) also contains a part that updates the μ-coefficients. Due to the fact that the actual operations on the exact and approximate lattice basis are simple to parallelize, our parallelization computes the μ-update separately. Consequently, Algorithm 6 has two parts, the μ-update (Lines (1) - (8)) and lattice basis update (Lines (9) - (13)). Our solution furthermore introduces an additional array f to store the μ_{ij} values that are necessary for the update of the lattice basis. For

the first part of Algorithm 6, we adopt a similar strategy to the one used for the orthogonalization (Section 3.2). In contrast to the orthogonalization, we do not need to modify the loop (Algorithm 6, Lines (7) - (8)) that updates the μ-coefficients in order to parallelize it. The dedicated main thread, like in the orthogonalization part, updates only small segments of size s_μ of the μ_{im}. In addition, it performs the computations that cannot be done in parallel (Algorithm 6, Lines (2) - (6)) and distributes the computation among all threads for the parallel parts (Thread$_1$, Lines (21) - (25) and Thread$_t$, Lines (4) - (8)).

Algorithm 5:

```
(1)  for (i > j ≥ 1) do
(2)    if (|μ_{i,j}| > 1/2) then
(3)      F_r = true
(4)      b_i = b_i − ⌈μ_ij⌋ b_j
(5)      if (|μ_ij| > 2^{p/2}) then
(6)        F_c = true
(7)      for (1 ≤ m ≤ j) do
(8)        μ_im = μ_im − ⌈μ_ij⌋ μ_jm
(9)  if (F_r = true) then
(10)   APPROX_VECTOR(b'_i, b_i)
```

Algorithm 6:

```
(1)  for (i > j ≥ 1) do
(2)    f_j = μ_ij
(3)    if (|μ_ij| > 1/2) then
(4)      F_r = true
(5)      if (|μ_ij| > 2^{p/2}) then
(6)        F_c = true
(7)      for (1 ≤ m ≤ j) do
(8)        μ_im = μ_im − ⌈μ_ij⌋ μ_jm
(9)  for (i > j ≥ 1) do
(10)   if (|f_j| > 1/2) then
(11)     b_i = b_i − ⌈f_j⌋ b_j
(12) if (F_r = true) then
(13)   APPROX_VECTOR(b'_i, b_i)
```

μ-Update – Thread$_1$ (main)

```
(1)  j = i − 1, j_e = 0
(2)  while (j ≥ 1) do
(3)    j_s = j, m = 0
(4)    while (m < s_μ ∧ j ≥ j_e) do
(5)      for (j_s ≥ l > j) do
(6)        μ_ij = μ_ij − ⌈f_l⌋ μ_lj
(7)      if (|μ_ij| > 1/2) then
(8)        f_j = μ_ij
(9)        F_r = true
(10)       if (|μ_ij| > 2^{p/2}) then
(11)         F_c = true
(12)       μ_ij = μ_ij − ⌈μ_ij⌋
(13)       m = m + 1
(14)     else
(15)       f_j = 0
(16)     j = j − 1
(17)   od
(18)   j_c = j
(19)   COMPUTE_SPLIT_VALUES_2(split)
(20)   BARRIER_WAIT(b_2)
(21)   j_e = split_T
(22)   for (j_s ≥ m > j) do
(23)     if (f_m ≠ 0) then
(24)       for (split_T ≤ l < split_{T+1}) do
(25)         μ_il = μ_il − ⌈f_m⌋ μ_ml
(26) od
```

μ-Update – Thread$_t$

```
(1)  j = i − 1
(2)  while (j ≥ 1) do
(3)    BARRIER_WAIT(b_2)
(4)    j = j_c
(5)    for (j_s ≥ m > j_c) do
(6)      if (f_m ≠ 0) then
(7)        for (split_t ≤ l < split_{t+1}) do
(8)          μ_il = μ_il − ⌈f_m⌋ μ_ml
(9)  od
```

The difference in computations for the main and helper thread compared to the orthogonalization might require a different splitting. We therefore use the function COMPUTE_SPLIT_VALUES$_2$ to compute the split values. The check for $f_m \neq 0$ (Thread$_1$, Line (23) and Thread$_t$, Line (6)) ensures, that the μ coefficients are only updated if the corresponding $|\mu_{ij}| > \frac{1}{2}$. The second part of Algorithm 6 (Lines (9) - (13)), namely the actual size-reduction step or the update of the exact and approximate lattice basis, can easily be computed in parallel. We only need to split up every vector operation into equal sized parts. This way we can avoid dependencies that would require the use of barriers or mutexes. In contrast to the scalar product part (Section 3.1) we do not have to deal with a phenomenon like the unpredictable behavior of exact scalar products that would suggest splitting up the vector operations into smaller segments. For T being the number of threads and n the dimension of a lattice basis vector, the implementation for Thread$_t$ of the parallel size-reduction for our parallel LLL looks as follows:

Size Reduction – Thread$_t$

(1) **if** ($F_r = true$) **then**
(2) $\quad$ s $= \lceil \frac{(t-1)n}{T} \rfloor$, $\quad$ e $= \lceil \frac{tn}{T} \rfloor$
(3) $\quad$ **for** ($i-1 \geq j \geq 1$) **do**
(4) $\quad\quad$ **if** ($|f_j| > \frac{1}{2}$) **then**
(5) $\quad\quad\quad$ **for** (s $\leq$ l $\leq$ e) **do**
(6) $\quad\quad\quad\quad$ $b_{il} = b_{il} - \lceil f_j \rfloor b_{jl}$
(7) $\quad$ **for** (s $\leq$ l $\leq$ e) **do**
(8) $\quad\quad$ $b'_{il} =$ APPROX_VALUE(b_{il})
(9) $\quad$ $F_r = false$

4 Experiments

4.1 Lattice Bases

The experiments in this paper were performed using sparse and dense lattice bases in order to show that our parallel (i.e., multi-threaded) LLL reduction algorithm performs and scales well on different types of lattice bases. We have chosen a type of knapsack lattice [19, 9, 8, 27, 10] as an example for sparse bases and unimodular lattices as representatives for dense bases. Previous experiments [37, 2] showed that unimodular lattice bases are more difficult to reduce than knapsack lattices given the same dimension and maximum length of the basis entries.

We are using lower and upper triangular matrices with determinant of 1 to generate unimodular lattice bases. The entries in the diagonal are set to 1 while the lower (respectively upper part) of the matrix is selected uniformly at random. Given a lower triangular matrix U and an upper triangular matrix V, we generate the unimodular lattice bases for our tests as $B_u = V \cdot U$. This construction method has been chosen based on earlier experiments [2] to allow for testing of unimodular lattice

bases in higher dimensions in a reasonable time frame. We generated 100 unimodular lattice bases for each dimension $n \in [50, 750]$ in steps of 50 and maximum bit length $b = 1000$. The entries within the lower and upper triangular matrix have been chosen accordingly.

Knapsack lattices have been developed in the context of solving subset sum problems (see Definition 2 in the Appendix) [19, 9, 8, 27]. A typical knapsack lattice basis looks as follows:

$$B_k = \begin{pmatrix} 2 & 0 & \cdots & 0 & 0 & 1 \\ 0 & 2 & \cdots & 0 & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 2 & 0 & 1 \\ 0 & 0 & \cdots & 0 & 2 & 1 \\ a_1W & a_2W & \cdots & a_{n-1}W & a_nW & SW \\ 0 & 0 & \cdots & 0 & 0 & -1 \\ W & W & \cdots & W & W & \frac{n}{2}W \end{pmatrix}$$

For our experiments we have defined $W = \lceil \sqrt{n} \rceil + 1$. The weights $a_1, \ldots, a_n$ have been chosen, uniformly at random[4] from the interval $(1, 2^b]$. We refer to b as the maximum bit length of the lattice basis entries. The sum S has been computed by adding up half of the weights that we have selected at random. We generated and tested 100 knapsack lattices for each dimension $n \in [50, 1000]$ in steps of 50 and maximum bit length $b \in \{1000, 2500\}$.

4.2 Parameters

In this section, we discuss the parameters introduced in Section 3 that are responsible for balancing computations among all threads. As discussed before, every part of the multi-threaded LLL (Section 3) has to be balanced in order to minimize the waiting time at the barriers in between two parallel parts. The number of iterations of the main loop in the multi-threaded LLL algorithm would multiply these waiting times and therefore every major imbalance would decrease the maximum speed-up factor.

The scalar product part (Section 3.1) introduces the parameters $start_t$ and end_t to minimize the amount of mutexes by assigning each thread a fixed first segment of scalar products to be computed. The most important parameter is the size s_{sp} of the subsequent segment. Bigger segments would decrease the number and therefore the overhead caused by locks. Smaller segments, on the other hand, minimize the time needed for a single segment and therefore the waiting time which leads to a more balanced computation. In the orthogonalization part we use s_o for the size of a segment in the computation of R_{ij}. For smaller values of s_o the increased number of barriers (i.e., synchronization points) would impact the overall running time significantly. In experiments we also observed an increase in the overall system time. Larger values, on the other hand, result in a bigger part of the computation being computed by the main thread only which has a negative impact on the scaling or speed-up factor. The split values computed by COMPUTE_SPLIT_VALUES$_1$ play an important

[4]The weights have been chosen independent of the density of the corresponding subset sum problem

role in the overall balancing of the computation. A slight imbalance in the partitioning could give the main thread the extra time to already perform the additional computations for the next iteration of the main loop. We take measures to ensure that the split values of the current iteration are not overwritten in next iteration of the main thread. For the μ-update within the size-reduction part the situation is similar to the orthogonalization part. The size s_μ of a segment of μ_{ij} to be updated has the same effect as the size s_o has on the orthogonalization. Nevertheless, the optimal value for s_μ might be different from the one for s_o due to the different amount of computations within the main thread. The same argument applies to the split values computed by COMPUTE_SPLIT_VALUES$_2$.

We have determined suitable parameter choices by performing small-scale tests with the 2-thread and the 4-thread version of our new parallel LLL algorithm. In our experiments, we use $s_{sp} = s_o = s_\mu = 16$. For the split values (COMPUTE_SPLIT_VALUES$_1$ and COMPUTE_SPLIT_VALUES$_2$) we confirmed that a slight imbalance is indeed beneficial. For the 2-thread case, the split-up is 47.5% for the main thread and 52.5% for the other thread. In the 4-thread version, we modified the split-up to 22% for the main thread and 26% for each of the other threads.

4.3 Results

The experiments in this paper were performed on Sun X2200 servers with two dual core AMD Opteron processors (2.2 GHz) and 4 GB of main memory using the Sun Solaris 10 OS. We compiled all programs with GCC 4.1.2 [39] using the same optimization flags. For our implementation of the parallel and non-parallel LLL algorithm we used GMP 4.2.2 [40] with the AMD64 patch [41] as long integer arithmetic and MPRF 2.3.1 [42] as multi-precision floating-point arithmetic for the approximation of the lattice basis. The experiments were performed using MPFR with 128 bit of precision. Based on our setup we conducted experiments with the non-parallel Schnorr-Euchner algorithm (Algorithm 1) and the 2-thread and 4-thread version (Section 3) of our newly developed implementation of the parallel LLL algorithm.

We used `gettimeofday` to measure the actual time (real time) that the LLL algorithms needed for the reduction. The function `getrusage` is used to determine how much processor time (user time) and system time was necessary. For non-parallel (or single-threaded) applications, real time is, depending on the system load, roughly the sum of user and system time. But in case of multi-threaded applications, `getrusage` returns the sum of user and system time for all threads. We therefore use real time for the single and multi-threaded LLL algorithm to determine the scaling or speed-up factor for our new multi-threaded implementation. Comparing the sum of user and system time for the single and multi-threaded LLL allows us to compute the necessary overhead for the multi-threaded version.

Figure 2 shows the average running time (real time) per dimension for the non-parallel, the 2-thread and 4-thread version of the LLL algorithm. We limited our experiments for the unimodular lattice bases to bases up to dimension 750 due to time constraints. It can be seen, that the threaded versions of the algorithm lead to a significant decrease in terms of reduction time. The 4-thread version is significantly faster than the 2-thread version. Figure 3 shows the average speed-up factor

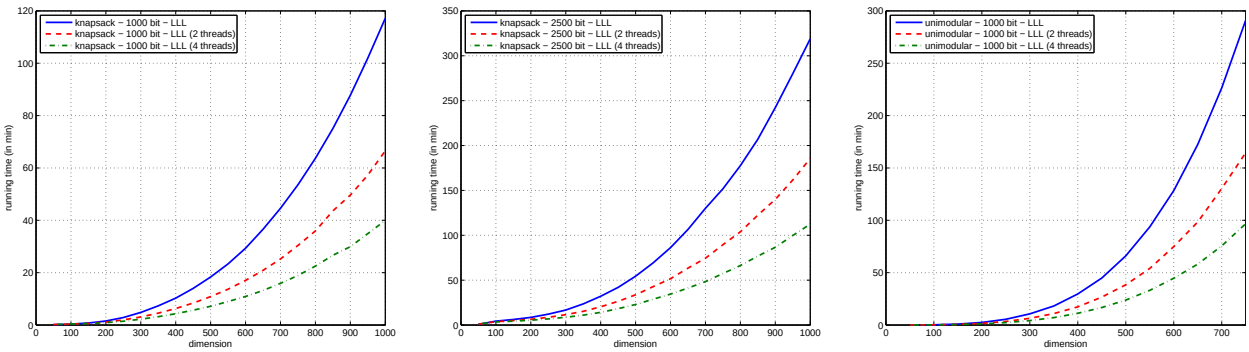

Figure 2: Reduction time (in minutes) for knapsack and unimodular lattice bases

per dimension for the new multi-threaded LLL compared to the non-parallel variant. The speed-up (i.e., scaling factor) is the quotient of the real time of the non-parallel version and the real time of the multi-threaded version. The speed-up factor increases with the dimension up to around 1.75 for the 2-thread and close to factor 3 for the 4-thread version of our new parallel LLL algorithm. It is interesting to note that for the selected parameters the speed-up factor for unimodular lattice bases increase faster than for knapsack lattices, and eventually they reach a similar maximum, as one can clearly observe in the 2-thread case. The 2-thread version, as expected, reaches its maximum earlier than the 4-thread version. Figure 4 shows the average overhead per dimension caused

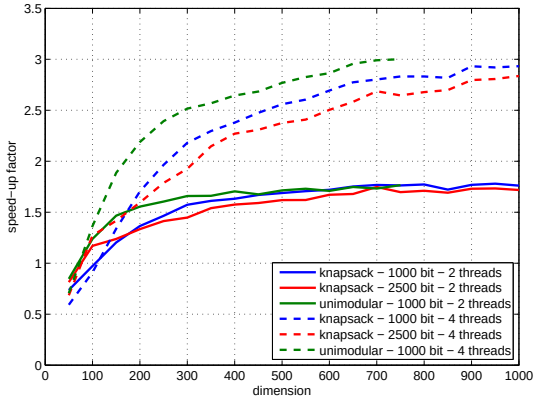

Figure 3: Speed-up for multi-threaded LLL

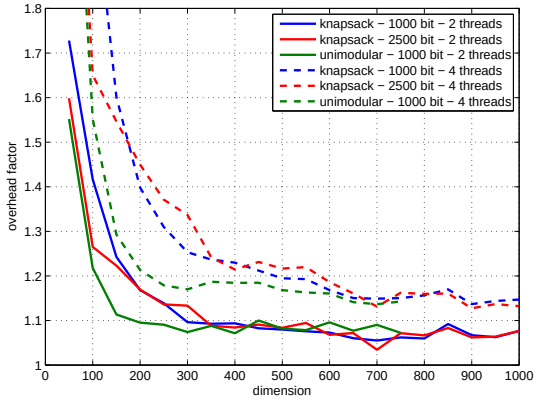

Figure 4: Overhead for multi-threaded LLL

by the new parallel LLL algorithm compared to the non-parallel Schnorr-Euchner algorithm. We define the overhead as the quotient of the sum of user and system time of all threads to the sum of user and system time for the non-parallel version. The overhead of the new parallel LLL decreases with increasing dimension of the lattice basis to less than 10% for the 2-thread and less than 15% for the 4-thread version.

5 Conclusion and Future Work

In this paper we introduced a new parallel LLL reduction algorithm based on the Schnorr-Euchner algorithm. We used POSIX threads to parallelize the reduction in a shared memory setting. In our experiments we show that our new variant scales well and achieves a considerable decrease in the running time of the LLL reduction by taking advantage of today's computers multi-core, multi-processor capabilities.

Future work includes finding heuristics for the selection of the parameters that control the balancing among all threads. In addition, we plan to explore the possibility to parallelize the LLL Gram using buffered transformations [3]. The structure and the required updates of the Gram matrix pose the main challenge in parallelizing this algorithm. We are also looking into the possibility of developing a parallel LLL for the GPU of graphics cards.

6 Acknowledgments

This work was partially supported by the Sun Microsystems Academic Excellence Grant Program.

References

[1] W. Backes and S. Wetzel. New Results on Lattice Basis Reduction in Practice. In *Algorithmic Number Theory (ANTS-00)*, volume 1838 of *LNCS*, pages 135–152. Springer, 2000.

[2] W. Backes and S. Wetzel. Heuristics on Lattice Basis Reduction in Practice. *ACM Journal on Experimental Algorithms*, 7, 2002.

[3] W. Backes and S. Wetzel. An Efficient LLL Gram Using Buffered Transformations. In *Proceedings of CASC 2007*, volume 4770 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 2007.

[4] D. Bleichenbacher and A. May. New Attacks on RSA with Small Secret CRT-Exponents. In *Public Key Cryptography (PKC) 2006*, volume 3958 of *LNCS*, pages 1–13. Springer, 2006.

[5] J. Blömer and A. May. A Tool Kit for Finding Small Roots of Bivariate Polynomials over the Integers. In *In Advances in Cryptology (Eurocrypt 2005)*, volume 3494 of *LNCS*, pages 251–267. Springer, 2005.

[6] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[7] H. Cohen. *A Course in Computational Algebraic Number Theory*. Undergraduate Texts in Mathematics. Springer, 1993.

[8] M. Coster, A. Joux, B. LaMacchia, A. Odlyzko, C. Schnorr, and J. Stern. Improved Low-Density Subset Sum Algorithm. *Journal of Computational Complexity*, 2:111–128, 1992.

[9] M. Coster, B. LaMacchia, A. Odlyzko, and C. Schnorr. An Improved Low-Density Subset Sum Algorithm. In *Proceedings EUROCRYPT 1991*, volume 547 of *LNCS*, pages 54–67. Springer, 1991.

[10] I. Damgård. A Design Principle for Hash Functions. In *Proceedings of CRYPTO 1989*, volume 435 of *LNCS*, pages 416–427. Springer, 1989.

[11] B. Filipovic. Implementierung der Gitterbasenreduktion in Segmenten. Master's thesis, University of Frankfurt am Main, 2002.

[12] C. Heckler. *Automatische Parallelisierung und parallele Gitterbasisreduktion*. PhD thesis, Universität des Saarlandes, Saarbrücken, 1995.

[13] C. Heckler and L. Thiele. Parallel complexitiy of lattice basis reduction and a floating-point parallel algorithm. In *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe (PARLE 93)*, pages 744–747, London, UK, 1993. Springer-Verlag.

[14] C. Heckler and L. Thiele. A parallel lattice basis reduction for mesh-connected processor arrays and parallel complexity. In *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing (SPDP 93)*, pages 400–407, Dallas, 1993.

[15] A. Joux. A Fast Parallel Lattice Basis Reduction Algorithm. In *Proceedings of the Second Gauss Symposium*, pages 1–15, 1993.

[16] A. Joux. *La Réduction des Réseaux en Cryptographie*. PhD thesis, Labaratoire d'Informatique de L'Ecole Normale Superieure LIENS, Paris, France, 1993.

[17] H. Koy and C. Schnorr. Segment LLL-Reduction of Lattice Bases. In *Cryptography and lattices*, volume 2146 of *LNCS*, pages 67 – 80. CaLC 2001, Springer, 2001.

[18] H. Koy and C. Schnorr. Segment LLL-Reduction with Floating Point Orthogonalization. In *Cryptography and Lattices*, volume 2146 of *LNCS*, pages 81 – 96. CaLC 2001, Springer, 2001.

[19] C. Lagarias and A. Odlyzko. Solving Low-Density Subset Sum Problems. *JACM*, 32:229–246, 1985.

[20] A. Lenstra, H. Lenstra, and L. Lovász. Factoring Polynomials with Rational Coefficients. *Math. Ann.*, 261:515–534, 1982.

[21] J.-L. Louis Roch and G. Villard. Parallel gcd and lattice basis reduction. In *CONPAR '92/ VAPP V: Proceedings of the Second Joint International Conference on Vector and Parallel Processing*, pages 557–564, London, UK, 1992. Springer-Verlag.

[22] A. May. *New RSA Vulnerabilities Using Lattice Reduction Methods*. PhD thesis, University of Paderborn, 2003.

[23] A. May. Secret Exponent Attacks on RSA-type Schemes with Moduli n=pr̂q. In *Public Key Cryptography, PKC 2004*, volume 2947 of *LNCS*, pages 218–230. Springer, 2004.

[24] D. Micciancio and S. Goldwasser. *Ccomplexity of Lattice Problems—A Cryptographic Perspective*. Kluwer Academic Publishers, 2002.

[25] P. Nguyen and D. Stehlé. Low-Dimensional Lattice Basis Reduction Revisited. In *Proceedings of the 6th Algorithmic Number Theory Symposium (ANTS-04)*, volume 3076 of *LNCS*, pages 338–357. Springer, 2004.

[26] P. Nguyen and D. Stehlé. Floating-Point LLL Revisited. In *Proceedings of Eurocrypt 2005*, volume 3494 of *LNCS*, pages 215–233. Springer, 2005.

[27] P. Nguyen and J. Stern. Adapting Density Attacks to Low-Weight Knapsacks. In *Advances in Cryptology (Asiacrypt 2005)*, LNCS, pages 41–58. Springer, 2005.

[28] P. Q. Nguyen. Cryptanalysis of the Goldreich-Goldwasser-Halevi Cryptosystem from Crypto '97. In *Advances in Cryptology (CRYPTO 1999)*, volume 1666 of *LNCS*, pages 288 – 304. Springer, 1999.

[29] P. Q. Nguyen and J. Stern. Lattice Reduction in Cryptology: An Update. In *Algorithmic Number Theory (ANTS-00)*, volume 1838 of *LNCS*, pages 85 – 112. Algorithmic Number Theory Symposium 4, 2000, Leiden ANTS-00, Springer, 2000.

[30] M. E. Pohst and H. Zassenhaus. *Algorithmic Algebraic Number Theory*. Cambridge University Press, 1989.

[31] O. Regev. Lattice-based Cryptography. In *Advances in Cryptology (CRYPTO 2006)*, volume 4117 of *LNCS*, pages 131–141. Springer, 2006.

[32] C. Schnorr and M. Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. In *Proceedings of Fundamentals of Computation Theory '91*, volume 529 of *LNCS*, pages 68–85. Springer, 1991.

[33] C. Schnorr and M. Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. Technical report, Universität Frankfurt, 1993.

[34] R. W. Stevens and S. A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional, 2005.

[35] G. Villard. Parallel lattice basis reduction. In *ISSAC '92: Papers from the international symposium on Symbolic and algebraic computation*, pages 269–277, New York, NY, USA, 1992. ACM.

[36] S. Wetzel. An Efficient Parallel Block-Reduction Algorithm. In *Algorithmic number theory (ANTS-98)*, volume 1423 of *LNCS*, pages 323 – 337. Springer, 1998.

[37] S. Wetzel. *Lattice Basis Reduction Algorithms and their Applications*. PhD thesis, Universität des Saarlandes, 1998.

[38] K. Wiese. Parallelisierung von lll-algorithmen zur gitterbasisreduktion. eine implementierung auf dem intel ipsc/860 hypercube. Master's thesis, Universität des Saarlandes, Saarbrücken, 1994.

[39] GCC - Homepage, November 2008. `http://gcc.gnu.org`.

[40] GMP - Homepage. `http://gmplib.org/`.

[41] AMD64 patch for GMP 4.2, November 2008. `http://www.loria.fr/~gaudry/mpn_AMD64/index.html`.

[42] MPFR - Homepage, November 2008. `http://www.mpfr.org/`.

Appendix

Definition 2 *(Subset Sum Problem) For given weights $a_1, \ldots, a_n \in \mathbb{N}$ and the sum $S \in \mathbb{N}$ find coefficients $x_1, \ldots x_n \in \{0,1\}$ such that $S = \sum_{i=1}^{n} x_i a_i$.*

Definition 3 *The density of the subset sum problem is defined as $d(a_1, \ldots, a_n) = \frac{n}{log(\max\{a_1,\ldots,a_n\})}$.*

Remark 4. *In [9, 8] an $(n+1)$ dimensional lattice $L(B_k) \subset \mathbb{Z}^{n+3}$ is used to solve subset sum problems with density $d(a_1, \ldots, a_n) < 0.9408$.*