Parallel Gauss Sieve Algorithm: Solving the SVP in the Ideal Lattice of 128 dimensions

Tsukasa Ishiguro[1], Shinsaku Kiyomoto[1], Yutaka Miyake[1], and Tsuyohsi Takagi[2]

[1] KDDI R&D Laboratories Inc., 2-1-15 Ohara, Fujimino, Saitama 356-8502, Japan
`{tsukasa,kiyomoto,miyake}@kddilabs.jp`
[2] Institute of Mathematics for Industry, Kyushu University, 744, Motooka, Nishi-ku, Fukuoka 819-0395, Japan
`takagi@imi.kyushu-u.ac.jp`

Abstract. In this paper, we report that we have solved the shortest vector problem (SVP) over a 128-dimensional lattice, which is currently the highest dimension of the SVP that has ever been solved. The security of lattice-based cryptography is based on the hardness of solving the SVP in lattices. In 2010 Micciancio *et al.* proposed a Gauss Sieve algorithm for heuristically solving the SVP using list L of Gauss-reduced vectors. Milde *et al.* proposed a parallel implementation method for the Gauss Sieve algorithm. However, the efficiency of more than 10 threads in their implementation decreases due to a large number of non-Gauss-reduced vectors appearing in the distributed list of each thread. In this paper, we propose a more practical parallelized Gauss Sieve algorithm. Our algorithm deploys an additional Gauss-reduced list V of sample vectors assigned to each thread, and all vectors in list L remain Gauss-reduced by mutually reducing them using all sample vectors in V. Therefore, our algorithm enables the Gauss Sieve algorithm to run without excessive overhead even in a large-scale parallel computation of more than 1,000 threads. Moreover, for speed-up, we use the bi-directional rotation structure of an ideal lattice that makes the generation of additional vectors in the list with almost no additional overhead. Finally, we have succeeded in solving the SVP over a 128-dimensional ideal lattice generated by cyclotomic polynomial $x^{128}+1$ using about 30,000 CPU hours.

Keywords: shortest vector problem, lattice-based cryptography, ideal lattice, Gauss Sieve algorithm, parallel algorithm

1 Introduction

Lattice-based cryptography has been considered a powerful primitive for constructing useful cryptographic protocols. The security of lattice-based cryptography is based on the hardness of solving the shortest vector problem (SVP), which involves searching for the shortest non-zero vectors in lattices. Ajtai has proved that the worst case complexity of solving the SVP is NP-hard under randomized reductions [1]. The α-SVP [15] is an approximation problem of the SVP, which searches for elements with the size of the shortest vector multiplied by a small approximation factor α. Many cryptographic primitives have been built on lattices due to their security against quantum computers and their novel functionalities: Ajtai-Dwork scheme [2], NTRU [12], fully-homomorphic cryptosystems [10], and multi-linear maps [9].

There are several approaches for solving the SVP and the α-SVP. The fastest deterministic algorithm is the Voronoi cell algorithm [16], which runs in exponential time $2^{O(n)}$ and space $2^{O(n)}$ for n-dimensional lattices. The sieving algorithms, which are explained in the next subsection, are probabilistic algorithms that require exponential time $2^{O(n)}$ and space $2^{O(n)}$ [3, 19, 6, 5]. The enumeration algorithms are exhaustive search algorithms that need time $2^{O(n^2)}$ of exponent n^2, but only the polynomial size of space [27, 28, 8], and thus they are suitable for parallelization using multicore CPUs and GPUs. Moreover, the lattice basis reduction such as LLL or BKZ is a polynomial-time approximation algorithm [14, 26]. The family of reduction algorithms is used not only for solving the α-SVP but also for pre-computing before running the other algorithms for solving the SVP.

1.1 Sieving Algorithms and Ideal Lattices

In 2001 Ajtai *et al.* proposed the first sieve algorithm for solving the SVP [3]. The sieving algorithm consists of a list L of vectors in the lattice and a reduction algorithm that outputs a shorter vector from two input vectors. List L manages the vectors reduced by the reduction algorithm. The number of vectors in L increases but the norm of several vectors L is shrunk by the reduction algorithm, and eventually the shortest non-zero vector can be found in list L.

There are many variants of the sieving algorithm [19, 6, 5] that try to improve the computational costs of the algorithm. In 2009 Micciancio *et al.* proposed a practical sieving algorithm, called the Gauss Sieve algorithm [17]. The theoretical upper bound of the computation time of the Gauss Sieve algorithm is not yet proved; however, the Gauss Sieve algorithm is faster than any other sieve algorithm in practice, because it deploys a list L of pair-wisely Gauss-reduced vectors that can gradually reduce the norm of vectors in the list. The time complexity of the Gauss sieve is asymptotically estimated to be $2^{0.41n}$ for n-dimensional lattices [17]. In 2011 Milde *et al.* considered a parallelization variant of the Gauss Sieve algorithm. The distributed list $L_i(i = 1, 2, ..., t)$ of $L = \cup_i L_i$ with a queue $Q_i(i = 1, 2, ..., t)$ is assigned to each thread, where L_i is connected to adjacent list L_{i+1} and the Gauss-reduced vectors are transferred from list L_i to L_{i+1} using queue Q_i for $t \in \mathbb{N}$, where we set $L_{t+1} = L_1$. However, several vectors in the whole list $L = \cup_i L_i$ are no longer Gauss-reduced without exact synchronization of queues Q_i for all threads, and thus the efficiency decreases as the number of threads increases. From the experiment in Milde *et al.*, once the number of threads increases to more than ten, the speed-up factor does not exceed around five. Therefore, it is difficult to apply to large-scale parallel computation.

In order to realize efficient construction of lattice-based cryptography, ideal lattices are often used. Using ideal lattices, many cryptographic primitives work faster and require less storage [12, 9]. One of the open problems is whether the computational problems related to the ideal lattices are easier to solve compared with those of random lattices [21]. First, Micciancio *et al.* mentioned the possibility of speeding up the sieving algorithm for ideal lattices [17]. In ideal lattices, several vectors of similar norms have a rotation structure, and thus it is possible to compute the set of vectors in the reduction algorithm derived from the sieve algorithm without a large overhead. Schneider *et al.* proposed the Ideal Gauss Sieve algorithm, which uses the rotation structure of the *Anti-cyclic lattice* generated by polynomial $x^n + 1$ where n is a power of two [24]. Then their proposed algorithm enables the Gauss Sieve algorithm to run about 25 times faster on 60-dimensional ideal lattices.

1.2 Our Contribution

In this paper, we improve the Gauss Sieve algorithm as follows:

- *Efficient parallelization*
  We propose a parallelized Gauss Sieve algorithm using an additional list V generated by the multisampling technique of vectors in the lattice. Our algorithm mutually reduces the vectors in both L and V, so that all vectors in both list V and L remain pair-wisely Gauss-reduced. Using this technique, the reduction algorithm can be easily parallelized. Additionally, even if the number of threads increases, our algorithm keeps the vector set pairwise-reduced and efficiency is maintained. Therefore, our algorithm enables the Gauss Sieve algorithm to run without excessive overhead even in a large-scale parallel computation of more than 1,000 threads.
- *Speed-up technique using structure of ideal lattice*
  We find a new condition of the ideal lattice that is suitable for speed-up of the Gauss Sieve algorithm, namely a "*Trinomial lattice*" generated by the irreducible trinomial $x^n + x^k + 1$ for $1 < k < n$. A rotation operation on *Trinomial lattice* requires no greater computational cost than the *Anti-cyclic lattice.* Additionally, we propose speed-up techniques that use a bi-directional rotation structure of an ideal lattice, called "*Inverse rotation*" and "*Updating vectors*". Using *Inverse rotation*, the reversely rotated short vectors can be generated at the same cost as the rotation of vectors. Moreover, the *Updating vectors* technique can convert the vectors in the list to shorter ones with a non-negligible probability in the *Trinomial lattice.* Using these techniques, Gauss Sieve algorithm can solve a wider range of SVPs on the ideal lattice several times faster than on random lattices.

With the result of our proposed algorithm, we succeeded in solving the SVP on a 128-dimensional ideal lattice generated by the cyclotomic polynomial $x^{128}+1$ using about 30,000 CPU hours. In our experiment, we used 84 instances which have 16 cores and each instance runs the 32 threads, namely the number of total threads is 2,688. At this time, the SVP over 126-dimensional random lattice is the largest dimension that has been solved in the SVP Challenge by Chen *et al.* [25]. Our results will contribute to estimating the key length used in lattice-based cryptography.

2 Definitions and Problems

In this section, we provide a short overview of the definition of the SVP on the lattice. We then explain the concept of Gauss-reduced and pairwise-reduced for a set of vectors on the lattice used for the Gauss Sieve algorithm.

Let $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ be a set of n linearly independent vectors in $\mathbb{R}^m$. The lattice generated by B is the set $\mathcal{L}(B) = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_n) = \{\sum_{1\le i\le n} x_i\mathbf{b}_i,\ x_i \in \mathbb{Z}\}$ of all integer linear combinations of the vectors in B. The set B is called *basis* of the lattice $\mathcal{L}(B)$. In the following, we denote by $\mathcal{L}(\mathbf{B})$ the lattice of basis B as the matrix representation $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n) \in \mathbb{R}^{m\times n}$. If $n = m$, the lattice $\mathcal{L}(B)$ is called full-rank. In this paper, for the sake of simplicity, we will consider only full-rank lattices and assume that all the basis vectors $\mathbf{b}_i (i = 1, 2, ..., n)$ have only integer entries.

The Euclidean norm of vector $\mathbf{v} = (v_0, \ldots, v_{n-1}) \in \mathcal{L}(\mathbf{B})$ is denoted by $||\mathbf{v}|| = \sum_{0\le i<n} v_i^2$. The norm of the shortest and second non-zero vectors in $\mathcal{L}(\mathbf{B})$ is denoted by $\lambda_1(\mathcal{L}(\mathbf{B}))$ and $\lambda_2(\mathcal{L}(\mathbf{B}))$, respectively. The inner product of two vectors $\mathbf{a} = (a_0, \ldots, a_{n-1}), \mathbf{b} = (b_0, \ldots, b_{n-1}) \in \mathcal{L}(\mathbf{B})$ is defined by $\langle \mathbf{a}, \mathbf{b}\rangle = \sum_{0\le i<n} a_i b_i$. For $x \in \mathbb{R}$, $\lfloor x\rceil$ denotes the nearest integer to x, namely $\lfloor x + 1/2\rfloor$.

We define the shortest vector problem (SVP) on a lattice as follow.

Definition 1 (Shortest vector problem on a lattice) *Given a lattice $\mathcal{L}(\mathbf{B})$, find the shortest non-zero vector of the length $\lambda_1(\mathcal{L}(\mathbf{B}))$ in $\mathcal{L}(\mathbf{B})$.*

From the Gaussian heuristic, the length of the shortest vector in lattice $\mathcal{L}(\mathbf{B})$ is estimated to be $\lambda_1(\mathcal{L}(\mathbf{B})) = (1/\sqrt{\pi})\Gamma(\frac{n}{2}+1)^{\frac{1}{n}} \cdot \det(\mathcal{L}(\mathbf{B}))^{\frac{1}{n}}$, where $\Gamma(x)$ is the gamma-function [23].

Let $g(x) \in \mathbb{Z}[x]$ be a monic polynomial of degree n, and let I be an ideal of ring $\mathbb{Z}[x]/(g(x))$. All elements of ideal I are represented by polynomials $\mathbf{v}(x) = \sum_{0\le i<n} v_i x^i$ in $\mathbb{Z}[x]/(g(x))$. We identify $\mathbf{v}(x)$ with vectors $\mathbf{v} = (v_0, \ldots, v_{n-1}) \in \mathbb{Z}^n$. The ideal I is an additive subgroup of $\mathbb{Z}[x]/(g(x))$, and the set $\{\mathbf{v} = (v_0, \ldots, v_{n-1}) \in \mathbb{Z}^n | \mathbf{v}(x) = \sum_{0\le i<n} v_i x^i \in I\}$ becomes a lattice. This is called the ideal lattice generated by $g(x)$, and its basis B consists of the rotation vectors $x^i\mathbf{v}(x) \in \mathbb{Z}[x]/(g(x))$ for $i = 0, 1, ..., n-1$. The cyclotomic polynomials such as $g(x) = x^n + 1$ for $n = 2^h$ with some positive integer h are often used for generating the ideal lattice in cryptography.

2.1 Gauss-reduced and Pairwise-reduced

We define Gauss-reduced and pairwise-reduced for a set of vectors on lattice $\mathcal{L}(\mathbf{B})$. We then explain an algorithm for determining and reducing two given vectors of lattice $\mathcal{L}(\mathbf{B})$.

First, the definition of Gauss-reduced is as follows.

Definition 2 (Gauss-reduced) *If two different vectors $\mathbf{a}, \mathbf{b} \in \mathcal{L}(\mathbf{B})$ satisfy $||\mathbf{a} \pm \mathbf{b}|| \ge \max(||\mathbf{a}||, ||\mathbf{b}||)$, then $\mathbf{a}, \mathbf{b}$ are called Gauss-reduced.*

Micciancio *et al.* showed an algorithm to convert two vectors $\mathbf{a}, \mathbf{b}$ in $\mathcal{L}(\mathbf{B})$ to be Gauss-reduced [17]. The conversion algorithm uses the Reduce algorithm (Alg.1), which outputs vectors $\mathbf{a}'$ for two vectors $\mathbf{a}, \mathbf{b}$ in $\mathcal{L}(\mathbf{B})$. The reduced vector $\mathbf{a}'$ is a linear combination of $\mathbf{a}$ and $\mathbf{b}$ that has a shorter norm than $\max(\mathbf{a}, \mathbf{b})$, or otherwise $\mathbf{a}' = \mathbf{a}$. From this, we can determine whether two vectors $\mathbf{a}, \mathbf{b}$ in $\mathcal{L}(\mathbf{B})$ are Gauss-reduced. Indeed we can easily prove the following lemma.

Lemma 1. *Let $\mathbf{a}, \mathbf{b}$ be two vectors in $\mathcal{L}(\mathbf{B})$. We set $\mathbf{a}'$ =Reduce$(\mathbf{a}, \mathbf{b})$ and $\mathbf{b}'$ =Reduce$(\mathbf{b}, \mathbf{a})$. If both $\mathbf{a} = \mathbf{a}'$ and $\mathbf{b} = \mathbf{b}'$ hold, then $\mathbf{a}, \mathbf{b}$ are Gauss-reduced.*

Algorithm 1 Reduce [17]

INPUT: Vectors $\boldsymbol{p}_1, \boldsymbol{p}_2$ in lattice $\mathcal{L}(\mathbf{B})$
OUTPUT: Vector $\boldsymbol{p}_1$ in lattice $\mathcal{L}(\mathbf{B})$

if $|2 \cdot \langle \boldsymbol{p}_1, \boldsymbol{p}_2 \rangle| > \langle \boldsymbol{p}_2, \boldsymbol{p}_2 \rangle$ **then**
  $\boldsymbol{p}_1 \leftarrow \boldsymbol{p}_1 - \left\lfloor \frac{\langle \boldsymbol{p}_1, \boldsymbol{p}_2 \rangle}{\langle \boldsymbol{p}_2, \boldsymbol{p}_2 \rangle} \right\rceil \cdot \boldsymbol{p}_2$
return $\boldsymbol{p}_1$

Algorithm 2 Gauss Sieve (GS) [17]

INPUT: Lattice basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}, \alpha, \beta > 0 \in \mathbb{R}$
OUTPUT: A shortest vector in $\mathcal{L}(\mathbf{B})$

$L \leftarrow \{\}, S \leftarrow \{\}, K \leftarrow 0$
while $K < \alpha|L| + \beta$ **do**
  if $S \neq \{\}$ **then**
    Pop from Stack S to $\mathbf{v}$
  else
    Generate a new vector $\mathbf{v}$ using Klein's randomized rounding algorithm [13]
  $(\mathbf{v}', L, S) \leftarrow$ Gauss_Reduce$(\mathbf{v}, L, S)$ /* Alg.3 */
  if $||\mathbf{v}|| = 0$ **then**
    $K \leftarrow K + 1$
  else
    $L \leftarrow L \cup \{\mathbf{v}'\}$
return a shortest vector in $\mathcal{L}(\mathbf{B})$

If two vectors $\mathbf{a}, \mathbf{b}$ are not Gauss-reduced, then $\mathbf{a} \neq \mathbf{a}'$ or $\mathbf{b} \neq \mathbf{b}'$ holds by Lemma 1. Recall that the reduced vector $\mathbf{a}' \leftarrow$Reduce$(\mathbf{a}, \mathbf{b})$ has the property $||\mathbf{a}'|| \leq ||\mathbf{a}||$. After performing both Reduce$(\mathbf{a}, \mathbf{b})$ and Reduce$(\mathbf{b}, \mathbf{a})$ we know that the resulting vectors $(\mathbf{a}', \mathbf{b}')$ are either Gauss-reduced, or $\mathbf{a}'$ (or $\mathbf{b}'$) is strictly shorter than $\mathbf{a}$ (or $\mathbf{b}$), respectively. If we repeatedly run the Reduce algorithm for $\mathbf{a} = \mathbf{a}'$ and $\mathbf{b} = \mathbf{b}'$, then we expect the resulting vectors $(\mathbf{a}', \mathbf{b}')$ to become Gauss-reduced. From our experiments in the 100-dimensional lattices, we can obtain the Gauss-reduced vectors after 10 iterations.

If $\mathbf{a}, \mathbf{b}$ are linearly dependent, the output of Reduce$(\mathbf{a}, \mathbf{b})$ is always the zero vector, *i.e.*, $||\mathbf{a}'|| = 0$, which is called a "collision". The collision is used as the condition for terminating the Gauss Sieve algorithm.

Definition 3 (Pairwise-reduced) *Let A be a set of d vectors in $\mathcal{L}(\mathbf{B})$. If any pair of two vectors $(\mathbf{a}_i, \mathbf{a}_j)$ in A for $i, j = 1, \ldots, d, i \neq j$ is Gauss-reduced, then the A is called pairwise-reduced.*

In general, if we append a vector $\mathbf{b} \in \mathcal{L}(\mathbf{B})$ to a pairwise-reduced set A, then $A \cup \{\mathbf{b}\}$ is not always pairwise-reduced. If any pair of two vectors $(\mathbf{a}_i, \mathbf{b})$ for $\mathbf{a}_1, ..., \mathbf{a}_d \in A$ is Gauss-reduced, then the union $A \cup \{\mathbf{b}\}$ becomes pairwise-reduced from the definition. Obviously we can prove the following lemma that shows that the union of two pairwise-reduced sets of vectors becomes pairwise-reduced by checking whether the pair of two vectors from A and B are Gauss-reduced.

Lemma 2 (Combining Lemma). *Let $A = \{\mathbf{a}_1, \ldots, \mathbf{a}_r\}$ and $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_m\}$ be sets of vectors in $\mathcal{L}(\mathbf{B})$. Assume that both A and B are pairwise-reduced. If any pair of two vectors $(\mathbf{a}_i, \mathbf{b}_j)$ in A, B for $1 \leq i \leq r, 1 \leq j \leq m$ is Gauss-reduced, then the union $A \cup B$ is pairwise-reduced.*

This lemma is used for constructing our proposed parallel algorithm for the Gauss Sieve algorithm.

3 Gauss Sieve Algorithm

In this section, we briefly explain the Gauss Sieve algorithm [17], its parallel implementation [18], and the Ideal Gauss Sieve algorithm [24].

Algorithm 3 Gauss_Reduce [17]

INPUT: Vector $\mathbf{v}$ on lattice $\mathcal{L}(\mathbf{B})$, list L, stack S
OUTPUT: Updated vector $\mathbf{v}$, updated list L, updated stack S

1: $reduce\_flag \leftarrow true$
2: **while** $reduce\_flag = true$ **do**
3: $\quad reduce\_flag \leftarrow false$
4: $\quad$ **for** $\boldsymbol{\ell} \in L$ **do**
5: $\quad\quad \mathbf{v}' \leftarrow \text{Reduce}(\mathbf{v}, \boldsymbol{\ell})$ /* Alg.1 */
6: $\quad\quad$ **if** $\mathbf{v}' \neq \mathbf{v}$ **then**
7: $\quad\quad\quad reduce\_flag \leftarrow true$
8: $\quad\quad\quad \mathbf{v} \leftarrow \mathbf{v}'$
9: **while** $\boldsymbol{\ell} \in L$ **do**
10: $\quad \boldsymbol{\ell}' \leftarrow \text{Reduce}(\boldsymbol{\ell}, \mathbf{v})$ /* Alg.1 */
11: $\quad$ **if** $\boldsymbol{\ell}' \neq \boldsymbol{\ell}$ **then**
12: $\quad\quad S \leftarrow S \cup \{\boldsymbol{\ell}'\}$
13: $\quad\quad L \leftarrow L \backslash \{\boldsymbol{\ell}\}$
14: **return** $(\mathbf{v}, L, S)$

3.1 Gauss Sieve [17]

The Gauss Sieve (GS) algorithm was proposed by Micciancio *et al.* in 2009 [17] and it was implemented as **gsieve** library by Voulgaris [30]. This algorithm is shown in Alg.2 and Alg.3. We prepare two auxiliary lists L and S, where L and S are defined by a set of vectors and a stack of vectors, respectively. L and S are initially assigned as empty. In the beginning of the GS algorithm, a new vector $\mathbf{v}$ is randomly sampled using Klein's randomized rounding algorithm [13].

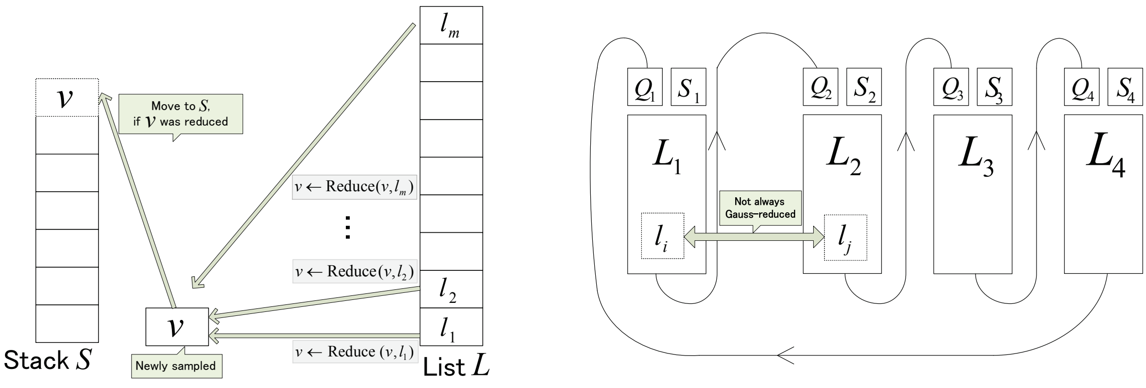
The GS algorithm runs a subroutine, Gauss_Reduce, which updates $\mathbf{v}, L, S$ by the steps in the following two parts. The first part (steps 4 to 8) runs the Reduce algorithm using a list L for updating $\mathbf{v}' = \text{Reduce}(\mathbf{v}, \boldsymbol{\ell}_i)$ for all vectors $\boldsymbol{\ell}_i \in L$. Once the $\mathbf{v}'$ is not equal to $\mathbf{v}$, this vector $\mathbf{v}'$ is moved to stack S. The reason is that if $\mathbf{v}$ is reduced using $\boldsymbol{\ell}_i \in L$, then $\mathbf{v}'$ and $\boldsymbol{\ell}_j, (i > j)$ are not always Gauss-reduced. If the $\mathbf{v}$ is not changed by $\text{Reduce}(\mathbf{v}, \boldsymbol{\ell}_i)$ for all $\boldsymbol{\ell}_i \in L$, the steps in the second part (steps 9 to 13) are performed. The second part runs the Reduce algorithm using a list L that makes the list pairwise-reduced. If $\boldsymbol{\ell}'_i \neq \boldsymbol{\ell}_i$ holds for $\boldsymbol{\ell}'_i = \text{Reduce}(\boldsymbol{\ell}_i, \mathbf{v})$, then the vector $\boldsymbol{\ell}'_i$ is moved to stack S and deleted from L. By the above steps, all pairs $(\mathbf{v}, \boldsymbol{\ell}_i)$ are always Gauss-reduced, where $\boldsymbol{\ell}_i \in L$. Therefore, $L \cup \boldsymbol{v}$ becomes pairwise-reduced by Lemma 2. Then L is updated by $L \cup \mathbf{v}$ and the iteration is continued (step 2 in Alg.2). If the stack is not empty, $\mathbf{v}$ is popped from the stack S, otherwise, $\mathbf{v}$ is newly sampled at step 4 in Alg.2.

The termination condition of the GS algorithm is determined by the number of collisions of the zero vector ($||\mathbf{a}'|| = 0$) that appears in L. The variable K in Alg.2 is the total number of collisions. When the value of K exceeds the threshold condition $\alpha|L| + \beta$, then the GS algorithm is terminated. In the **gsieve** library [30], $\alpha = 1/10$, and $\beta = 200$ are chosen as the threshold values. The theoretical upper bound of the complexity of the GS algorithm is not yet proved; however, in practice, the GS algorithm is faster than any other sieving algorithms. According to Micciancio *et al.* [17], the complexity of the GS algorithm is asymptotically estimated as time $2^{0.52n}$ and space $2^{0.21n}$.

The GS algorithm cannot be easily parallelized for the following reason. If the list is $L = \{\boldsymbol{\ell}_1, \boldsymbol{\ell}_2, ...\}$ in the first part (steps 4 to 8) in the Gauss_Reduce algorithm (Alg.3), then the algorithm runs $\mathbf{v}' \leftarrow \text{Reduce}(\mathbf{v}, \boldsymbol{\ell}_1)$, $\mathbf{v}'' \leftarrow \text{Reduce}(\mathbf{v}', \boldsymbol{\ell}_2)$,... in this iteration. Hence, the first part must be executed sequentially step by step. In contrast, it is easy to parallelize the second part (steps 9 to 13) of reducing the vectors $\boldsymbol{\ell}_1, ..., \boldsymbol{\ell}_t$ by $\text{Reduce}(\boldsymbol{\ell}_i, \mathbf{v})$ for fixed $\mathbf{v}$. Therefore, at most, only half of the Gauss_Reduce algorithm appearing in the GS algorithm can be parallelized. We show the flow of the GS algorithm in Figure 1(a).

3.2 Parallel implementation of the Gauss Sieve algorithm [18]

In 2011, Milde *et al.* proposed parallel implementation of the Gauss Sieve algorithm [18]. This method tries to extend the single Gauss Sieve algorithm into a parallel variant.



(a) Gauss Sieve [17] (b) Parallel implementation [18]

Fig. 1. Flows of the Gauss Sieve algorithm (left) and the parallel implementation (right) of the Gauss Sieve algorithm

Let t be the number of threads in this method. Each thread has an instance that consists of list L_i, stack S_i, and queue Q_i, where $L = \cup_i L_i$, $S = \cup_i S_i$, $i = 1, 2, ..., t$, and Q_i is used as a buffer for the next thread. The individual instances are connected together in a ring fashion. Each instance deals with a sample vector $\mathbf{v}$ in the distributed list L_i independently just as in the original Gauss Sieve algorithm. First, an instance runs Reduce$(\mathbf{v}, \boldsymbol{\ell}_i)$, where $\boldsymbol{\ell}_i \in L_i$. Second, the instance runs Reduce$(\boldsymbol{\ell}_i, \mathbf{v})$ inversely. After that, if it is not changed by the above reduction steps, the vector $\mathbf{v}$ is sent to the buffer Q_{i+1} of the next instance. Otherwise, $\mathbf{v}$ is moved to the distributed stack S_i in its own instance. If a vector $\mathbf{v}$ passed through all instances, the vector $\mathbf{v}$ is added to the distributed list L_i. If Q_i is empty, the instance generates a new sample vector. We show the flow of this parallel implementation of the Gauss Sieve algorithm in Figure 1(b).

In this method, each instance runs the Reduce algorithm in parallel. However, this method cannot ensure that the whole list $L = L_1 \cup \cdots \cup L_t$ remains pairwise-reduced. This is because when a vector is added to a distributed list, another instance may add extraneous vectors. Therefore, the number of non-Gauss-reduced pairs increases as the number of threads increases, and thus the overall performance of this parallel algorithm can not be accelerated for a large number of threads. In the experiment by Milde *et al.* [18], once the number of threads increases to more than ten, the speed-up factor does not exceed around five.

3.3 Ideal Gauss Sieve algorithm [24]

Schneider *et al.* proposed an Ideal Gauss Sieve algorithm [24] that uses the structure of an ideal lattice to improve the processing speed of the Gauss Sieve algorithm. The following ideal lattices support the rotation operation without additional cost and are suitable for speed-up[1].

- *Prime cyclotomic lattice*
  If $n+1$ is prime, an ideal lattice generated by the cyclotomic polynomial $\boldsymbol{g}(x) = x^n + x^{n-1} + \cdots + x + 1$ is called a *Prime cyclotomic lattice*. In this type, the rotation of vector $\mathbf{v}$ is $\mathbf{rot}(\mathbf{v}) = (-v_{n-1}, v_0 - v_{n-1}, \ldots, v_{n-2} - v_{n-1})$.
- *Anti-cyclic lattice*
  If n is a power of two, an ideal lattice generated by the cyclotomic polynomial $\boldsymbol{g}(x) = x^n + 1$ is called an *Anti-cyclic lattice*. In this type, the rotation of vector $\mathbf{v}$ is $\mathbf{rot}(\mathbf{v}) = (-v_{n-1}, v_0, \ldots, v_{n-2})$.

[1] Schneider *et al.* also discussed another type of ideal lattice, called the *Cyclic Lattice*. However, this type is generated by a non-cyclotomic polynomial. Because we focus here on the ideal lattice generated by a cyclotomic polynomial for the Ideal Lattice Challenge, we do not deal with the *Cyclic Lattice* in this paper.

The rotation maps of the above ideal lattices can generate new vectors that have a similar norm virtually for free. Therefore, we can implement the Gauss Sieve algorithm using the list L with the rotated vectors $\mathbf{rot}^i(\mathbf{v})$ for $i = 1, 2, ..., n-1$ in addition to $\mathbf{v}$ with a small overhead. The algorithm enables the Gauss Sieve algorithm to run about 25 times faster on 60-dimensional ideal lattices [24].

4 Proposed Parallel Gauss Sieve Algorithm

In this section, we propose the parallelized algorithm derived from the Gauss Sieve algorithm. We design our algorithm so that keeps the list L remains pairwise-reduced as with the Gauss Sieve algorithm, even though this algorithm works in parallel.

4.1 Overview

Let t be the number of threads used in our algorithm. Our algorithm prepares the auxiliary list V of r vectors in $\mathcal{L}(\mathbf{B})$, where each thread treats at most $s = \lfloor r/t \rfloor$ sample vectors for the list V. We also use the same list L and stack S in the Gauss Sieve algorithm, and the vectors in list L remain pairwise-reduced during our algorithm by controlling with list V. Each thread has list V, list L, and stack S, where we write $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_r\}$ and $L = \{\boldsymbol{\ell}_1, \ldots, \boldsymbol{\ell}_m\}$. After each iteration of the loop in our algorithm we pop vectors from the stack S to the list V. If the size of V is smaller than r, we generate new sample vectors by the Multisampling techniques. We explain how to construct the proposed threads in the following. There are three different reduction steps in our algorithm, namely **Reduction sample vectors using list vectors**, **Reduction sample vectors using sample vectors**, and **Reduction list vectors using sample vectors**. Our algorithm requires to use Alg.1 at most $\max(rm, r^2)$ times in each step, in other words, at most $\max(\lfloor rm/t \rfloor, \lfloor r^2/t \rfloor)$ times in each thread.

In the **Reduction sample vectors using list vectors**, let $s = \lfloor r/t \rfloor$ be the number of sample vectors treated by a thread, where r is the size of list V. Each thread has the distributed list $V_i = \{\mathbf{v}_{(i-1)s+1}, \ldots, \mathbf{v}_{is}\}$ and list L, where $V = \cup_i V_i$ and $i = 1, 2, ..., t$. Each thread i independently deals with list L and the sample vectors V_i, and runs $\mathbf{v}'_k = \text{Reduce}(\mathbf{v}_k, \boldsymbol{\ell}_j)$, where $\mathbf{v}_k \in V_i, \boldsymbol{\ell}_j \in L$, identical to a Gauss Sieve algorithm. If $\mathbf{v}'_k \neq \mathbf{v}_k$ holds, then the thread i moves the reduced vector $\mathbf{v}'_k$ into the stack S, otherwise, the thread i moves this vector $\mathbf{v}'_k$ into new list V'. At the end of this part, any vector $\mathbf{v}$ in list V' satisfies $\mathbf{v} = \text{Reduce}(\mathbf{v}, \boldsymbol{\ell})$ for all vectors $\boldsymbol{\ell}$ in list L. We show the flow of this part in Figure 2.

In the **Reduction sample vectors using sample vectors**, each thread has list V', which consists of r' vectors on a lattice. Let $s' = \lfloor r'/t \rfloor$ be the number of sample vectors treated by a thread. Each thread i deals with only a sample list V' and runs $\mathbf{v}'_k = \text{Reduce}(\mathbf{v}_k, \mathbf{v}_j)$, where $\mathbf{v}_k \in \{\mathbf{v}_{(i-1)s'+1}, \ldots, \mathbf{v}_{is'}\}, \mathbf{v}_j \in V'$ with $k \neq j$. If $\mathbf{v}'_k \neq \mathbf{v}_k$ holds, then the thread i moves the reduced vectors $\mathbf{v}'_k$ into the stack S, otherwise, the thread i moves the vectors $\mathbf{v}'_k$ into new list V''. At the end of this part, list V'' becomes pairwise-reduced and we have the relationship $V'' \subset V' \subset V$.

In the **Reduction list vectors using sample vectors**, let $\bar{s} = \lfloor m/t \rfloor$ be the number of list vectors treated by a thread, where m is the size of list L. Each thread has list $L_i = \{\boldsymbol{\ell}_{(i-1)\bar{s}+1}, \ldots, \boldsymbol{\ell}_{i\bar{s}}\}$ and V'', where $L = \cup_i L_i$, and $i = 1, 2, ..., t$. From our assumption, L is pairwise-reduced before processing this part. Each thread i deals with a distributed list L_i and a list V'' and runs $\boldsymbol{\ell}'_k = \text{Reduce}(\boldsymbol{\ell}_k, \mathbf{v}_j)$, where $\boldsymbol{\ell}_k \in L_i, \mathbf{v}_j \in V''$. If $\boldsymbol{\ell}'_k \neq \boldsymbol{\ell}_k$ holds, then the thread i moves the reduced vector $\boldsymbol{\ell}'_k$ into the stack S, otherwise, the thread i moves the vectors $\boldsymbol{\ell}_k$ into new list L'. At the end of this part, any vector $\boldsymbol{\ell}_k$ in the new list L' satisfies $\boldsymbol{\ell}_k = \text{Reduce}(\boldsymbol{\ell}_k, \mathbf{v}_j)$ for all vectors $\mathbf{v}_j$ in list V''. Here both L' and V'' are pairwise-reduced due to relationship $L' \subset L$ and $V'' \subset V'$, respectively.

After the above three reduction steps, our algorithm merges list L' and list V'' to create the new list $L = L' \cup V''$. Note that $\boldsymbol{\ell} = \text{Reduce}(\boldsymbol{\ell}, \mathbf{v})$ and $\mathbf{v} = \text{Reduce}(\mathbf{v}, \boldsymbol{\ell})$ hold for any vector $\boldsymbol{\ell} \in L'$ and $\mathbf{v} \in V''$. Therefore, any pair of two vectors $(\boldsymbol{\ell}, \mathbf{v})$ in L', V'' is Gauss-reduced by Lemma 1, and thus the union $L = L' \cup V''$ becomes pairwise-reduced by Lemma 2.

We show the algorithm derived from the proposed parallelized Gauss Sieve Algorithm in Alg.4. The inputs of this algorithm are a lattice on basis $\mathbf{B}$, the number of samplings $r \in \mathbb{N}$, and termination conditions

Algorithm 4 Proposed Parallel Gauss Sieve

INPUT: Lattice basis $\mathbf{B}$, the number of sample vectors $r \in \mathbb{N}$, $\alpha, \beta \in \mathbb{R}$
OUTPUT: A shortest vector $\mathbf{v}$ in $\mathcal{L}(\mathbf{B})$

1: $L \leftarrow \{\}, V \leftarrow \{\}, S \leftarrow \{\}, K \leftarrow 0$
/* Steps from 2 to 9 are described in **4.2 Multisampling of vectors** */
2: **while** $K < \alpha|L| + \beta$ **do**
3: **if** $|S| \neq 0$ **then**
4: $t \leftarrow \min(r, |S|)$
5: **for** $j = 1, \ldots, t$ **do**
6: Pop from Stack S to $\mathbf{v}_j$
7: **if** $|S| < r$ **then**
8: **for** $j = |S| + 1, \ldots, r$ **do**
9: Generate a new vector $\mathbf{v}_j$ using Klein's randomized rounding algorithm [13]
10: $V \leftarrow \{\mathbf{v}_1, ..., \mathbf{v}_r\}, V' \leftarrow \{\}, V'' \leftarrow \{\}, L' \leftarrow \{\}$
11: $L = \{\boldsymbol{\ell}_1, ..., \boldsymbol{\ell}_m\}$
/* Steps from 12 to 22 are described in **4.3 Reduction sample vectors using***/
12: **for** $i = 1, \ldots, r$ **do**
13: $\mathbf{w}_i \leftarrow \mathbf{v}_i$
14: **for** $j = 1, \ldots, m$ **do**
15: $\mathbf{w}_i \leftarrow \mathrm{Reduce}(\mathbf{w}_i, \boldsymbol{\ell}_j)$ /* This step can be ran in parallel */
16: **if** $||\mathbf{w}_i|| = 0$ **then**
17: $K{+}{+}$
18: **else if** $\mathbf{w}_i \neq \mathbf{v}_i$ **then**
19: $S \leftarrow S \cup \{\mathbf{w}_i\}$
20: **else**
21: $V' \leftarrow V' \cup \{\mathbf{w}_i\}$
22: $V' = \{\mathbf{v}_1, ..., \mathbf{v}_{r'}\}$
/* Steps from 23 to 34 are described in **4.4 Reduction sample vectors using sample vectors** */
23: **for** $i = 1, \ldots, r'$ **do**
24: $\mathbf{w}_i \leftarrow \mathbf{v}_i$
25: **for** $j = 1, \ldots, r'$ **do**
26: **if** $i \neq j$ **then**
27: $\mathbf{w}_i \leftarrow \mathrm{Reduce}(\mathbf{w}_i, \mathbf{v}_j)$ /* This step can be ran in parallel */
28: **if** $||\mathbf{w}_i|| = 0$ **then**
29: $K{+}{+}$
30: **else if** $\mathbf{w}_i \neq \mathbf{v}_i$ **then**
31: $S \leftarrow S \cup \{\mathbf{w}_i\}$
32: **else**
33: $V'' \leftarrow V'' \cup \{\mathbf{w}_i\}$
34: $V'' = \{\mathbf{v}_1, ..., \mathbf{v}_{r''}\}$
/* Steps from 35 to 45 are described in **4.5 Reduction list vectors using sample vectors***/
35: **for** $i = 1, \ldots, m$ **do**
36: $\mathbf{w}_i \leftarrow \boldsymbol{\ell}_i$
37: **for** $j = 1, \ldots, r''$ **do**
38: $\mathbf{w}_i \leftarrow \mathrm{Reduce}(\mathbf{w}_i, \mathbf{v}_j)$ /* This step can be ran in parallel */
39: **if** $||\mathbf{w}_i|| = 0$ **then**
40: $K{+}{+}$
41: **else if** $\mathbf{w}_i \neq \boldsymbol{\ell}_i$ **then**
42: $S \leftarrow S \cup \{\mathbf{w}_i\}$
43: **else**
44: $L' \leftarrow L' \cup \{\mathbf{w}_i\}$
45: $L' = \{\boldsymbol{\ell}_1, ..., \boldsymbol{\ell}_{m'}\}$
46: $L \leftarrow L' \cup V''$
47: **return** a shortest vector in $\mathcal{L}(\mathbf{B})$

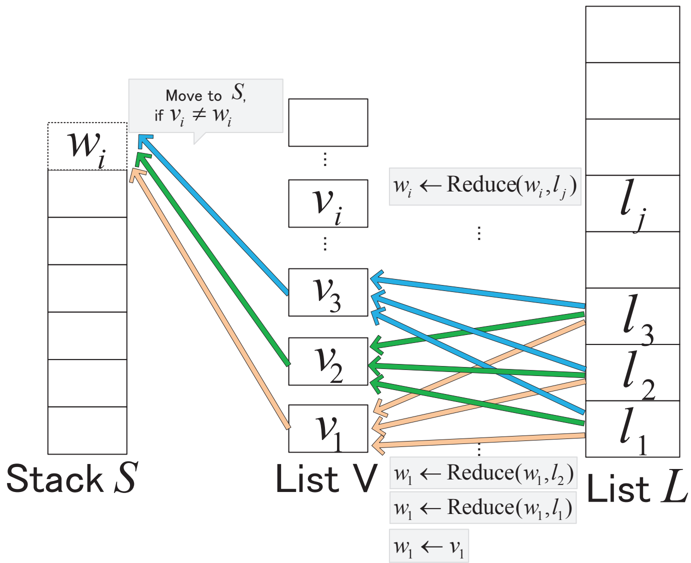


Fig. 2. Flow of the **Reduction sample vectors using list vectors** in the proposed parallel Gauss Sieve algorithm. Each task in different threads is indicated by color (*e.g.* blue, green, orange).

α, β. Here r is determined by the experimental scale, for example, the number of CPU cores or the available memory (we discuss the most suitable value based on an experiment described in section 5). In the following, we explain the details of the proposed algorithm.

4.2 Multisampling of vectors (Steps 3 to 9 in Alg.4)

We sample r vectors in lattice $\mathcal{L}(\mathbf{B})$ and construct a list $V = (\mathbf{v}_1, \ldots, \mathbf{v}_r)$ at the beginning of the iteration from step 3 to 9 in Alg.4. Sample vector $\mathbf{v}_i$ is samples in two ways, (*i.e.,* popping from stack S or newly generating just as in the case the Gauss Sieve algorithm). If $|S| \geq r$, all vectors $\mathbf{v}_i$ are popped from the stack S, where $1 \leq i \leq r$. If $0 < |S| < r$, we pop $|S|$ vectors from the stack S and generate $(r - |S|)$ vectors using Klein's sampling algorithm. If S is empty, all vectors $\mathbf{v}_i$ are newly generated using Klein's sampling algorithm.

4.3 Reduction of sample vectors using list vectors (Steps 12 to 22 in Alg.4)

In this part, by reducing the sample vectors in V using all vectors in list L we will construct the list V', which consists of vectors $\mathbf{v}_i \in V$ that satisfy $\mathrm{Reduce}(\mathbf{v}_i, \boldsymbol{\ell}_j) = \mathbf{v}_i$ for all $\boldsymbol{\ell}_j \in L$. Here denote $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_r\}$ and $L = \{\boldsymbol{\ell}_1, \ldots, \boldsymbol{\ell}_m\}$. At the beginning of this part we assign $\mathbf{w}_i \leftarrow \mathbf{v}_i$ at Step 13 in Alg.4. For $i = 1, 2, ..., r$, this part runs $\mathrm{Reduce}(\mathbf{w}_i, \boldsymbol{\ell}_j)$ from $j = 1$ to m for the fixed first input $\mathbf{w}_i$ and updates $\mathbf{w}_i$ using its output repeatedly. After running $\mathrm{Reduce}(\mathbf{w}_i, \boldsymbol{\ell}_j)$ for $\boldsymbol{\ell}_j \in L$, if $\mathbf{w}_i$ is changed (*i.e.,* $\mathbf{w}_i \neq \mathrm{Reduce}(\mathbf{w}_i, \boldsymbol{\ell}_j)$ for some $\boldsymbol{\ell}_j$), this vector $\mathbf{w}_i$ is moved to stack S, otherwise, $\mathbf{w}_i (= \mathbf{v}_i)$ is moved to the distributed list V'. This part runs the Reduce algorithm in the following order.

$\mathbf{w}_1 \leftarrow Reduce(\mathbf{w}_1, \boldsymbol{\ell}_1)$	$\mathbf{w}_i \leftarrow Reduce(\mathbf{w}_i, \boldsymbol{\ell}_1)$	$\vdots$
$\mathbf{w}_1 \leftarrow Reduce(\mathbf{w}_1, \boldsymbol{\ell}_2)$	$\mathbf{w}_i \leftarrow Reduce(\mathbf{w}_i, \boldsymbol{\ell}_2)$	$\mathbf{w}_r \leftarrow Reduce(\mathbf{w}_r, \boldsymbol{\ell}_1)$
$\vdots$	$\vdots$	$\mathbf{w}_r \leftarrow Reduce(\mathbf{w}_r, \boldsymbol{\ell}_2)$
$\mathbf{w}_1 \leftarrow Reduce(\mathbf{w}_1, \boldsymbol{\ell}_m)$	$\mathbf{w}_i \leftarrow Reduce(\mathbf{w}_i, \boldsymbol{\ell}_m)$	$\vdots$
$\vdots$	$\vdots$	$\mathbf{w}_r \leftarrow Reduce(\mathbf{w}_r, \boldsymbol{\ell}_m)$

At the end of this part, we re-index the vectors in V' form 1 to r' in no particular order, and rename the vectors in list V' from $\{\mathbf{w}_1, ..., \mathbf{w}_{r'}\}$ to $\{\mathbf{v}_1, ..., \mathbf{v}_{r'}\}$ at Step 22 in Alg.4. Recall that any vector $\mathbf{v}_i$ in list V' satisfies $\mathbf{v}_i$ =Reduce($\mathbf{v}_i, \boldsymbol{\ell}_j$) for all vectors $\boldsymbol{\ell}_j$ in list L. We have the relationship $V' \subseteq V$ and $|V'| = r' \le r$.

This part can be simply parallelized without heavy overhead. Let t be the number of threads and s be the number of sample vectors treated by a thread, where $s = \lfloor r/t \rfloor$. While a thread $i (1 \le i \le t)$ computes Reduce($\mathbf{w}_i, \boldsymbol{\ell}_1$) to Reduce($\mathbf{w}_i, \boldsymbol{\ell}_m$), another thread $j (j \ne i)$ can compute Reduce($\mathbf{w}_j, \boldsymbol{\ell}_1$) to Reduce($\mathbf{w}_j, \boldsymbol{\ell}_m$), because the vectors $\boldsymbol{\ell}_k$ in list L are not changed in this part. Therefore, the inner loop (from step 14 to 21) can be fully parallelized and the degree of parallelization is at most r, if we set $s = 1$. If $s > 1$, the thread i has $V_i = \{\mathbf{v}_{(i-1)s+1}, \ldots, \mathbf{v}_{is}\}$ and list L, where $V = \cup_i V_i$. And then the thread i runs Reduce($\mathbf{w}_{(i-1)s+1}, \boldsymbol{\ell}_1$) to Reduce($\mathbf{w}_{is}, \boldsymbol{\ell}_m$) sequentially in the following order.

$$
\begin{array}{ccc}
\boxed{\text{Thread } 1} & \boxed{\text{Thread } 2} & & \boxed{\text{Thread } t} \\
\begin{array}{c}
\mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \boldsymbol{\ell}_1) \\
\mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \boldsymbol{\ell}_2) \\
\vdots \\
\mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \boldsymbol{\ell}_m) \\
\vdots \\
\mathbf{w}'_s \leftarrow \mathit{Reduce}(\mathbf{w}_s, \boldsymbol{\ell}_1) \\
\mathbf{w}'_s \leftarrow \mathit{Reduce}(\mathbf{w}_s, \boldsymbol{\ell}_2) \\
\vdots \\
\mathbf{w}'_s \leftarrow \mathit{Reduce}(\mathbf{w}_s, \boldsymbol{\ell}_m)
\end{array}
&
\begin{array}{c}
\mathbf{w}_{s+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{s+1}, \boldsymbol{\ell}_1) \\
\mathbf{w}_{s+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{s+1}, \boldsymbol{\ell}_2) \\
\vdots \\
\mathbf{w}_{s+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{s+1}, \boldsymbol{\ell}_m) \\
\vdots \\
\mathbf{w}_{2s} \leftarrow \mathit{Reduce}(\mathbf{w}_{2s}, \boldsymbol{\ell}_1) \\
\mathbf{w}_{2s} \leftarrow \mathit{Reduce}(\mathbf{w}_{2s}, \boldsymbol{\ell}_2) \\
\vdots \\
\mathbf{w}_{2s} \leftarrow \mathit{Reduce}(\mathbf{w}_{2s}, \boldsymbol{\ell}_m)
\end{array}
& \cdots &
\begin{array}{c}
\mathbf{w}_{s(t-1)+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{s(t-1)+1}, \boldsymbol{\ell}_1) \\
\mathbf{w}_{s(t-1)+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{s(t-1)+1}, \boldsymbol{\ell}_2) \\
\vdots \\
\mathbf{w}_{s(t-1)+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{s(t-1)+1}, \boldsymbol{\ell}_m) \\
\vdots \\
\mathbf{w}_{st} \leftarrow \mathit{Reduce}(\mathbf{w}_{st}, \boldsymbol{\ell}_1) \\
\mathbf{w}_{st} \leftarrow \mathit{Reduce}(\mathbf{w}_{st}, \boldsymbol{\ell}_2) \\
\vdots \\
\mathbf{w}_{st} \leftarrow \mathit{Reduce}(\mathbf{w}_{st}, \boldsymbol{\ell}_m)
\end{array}
\end{array}
$$

4.4 Reduction of sample vectors using sample vectors (Step 23 to 34 in Alg.4)

In this part we try to convert the list $V' = \{\mathbf{v}_1, \ldots, \mathbf{v}_{r'}\}$ to be a pairwise-reduced list V''. We reduce sample vectors $\mathbf{v}_i \in V'$ using all vectors in $V' \setminus \{\mathbf{v}_i\}$ and construct list V'', which consists of vectors $\mathbf{v}_i$ that satisfy Reduce($\mathbf{v}_i, \mathbf{v}_j$) $= \mathbf{v}_i$ for all $\mathbf{v}_j \in V''$ with $i \ne j$. At the beginning of this part we assign $\mathbf{w}_i \leftarrow \mathbf{v}_i$ at Step 24 in Alg.4. For $i = 1, 2, ..., r'$, this part runs Reduce($\mathbf{w}_i, \mathbf{v}_j$) from $j = 1$ to m without $j = i$ for the fixed first input $\mathbf{w}_i$ and updates $\mathbf{w}_i$ using its output repeatedly. During all reductions, just after $\mathbf{w}_i$ is reduced even once, this vector $\mathbf{w}_i$ is moved to stack S as in the first reduction part. If $\mathbf{w}_i$ is not reduced ($\mathbf{w}_i$ = Reduce($\mathbf{w}_i, \mathbf{v}_j$)), this vector $\mathbf{w}_i (= \mathbf{v}_i)$ is moved to list V''. This part runs the Reduce algorithm in the following order.

$$
\begin{array}{ccc}
\begin{array}{c}
\mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \mathbf{v}_2) \\
\mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \mathbf{v}_3) \\
\vdots \\
\mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \mathbf{v}_{r'}) \\
\vdots
\end{array}
&
\begin{array}{c}
\mathbf{w}_i \leftarrow \mathit{Reduce}(\mathbf{w}_i, \mathbf{v}_1) \\
\mathbf{w}_i \leftarrow \mathit{Reduce}(\mathbf{w}_i, \mathbf{v}_2) \\
\vdots \\
\mathbf{w}_i \leftarrow \mathit{Reduce}(\mathbf{w}_i, \mathbf{v}_{r'}) \\
\vdots
\end{array}
&
\begin{array}{c}
\vdots \\
\mathbf{w}_{r'} \leftarrow \mathit{Reduce}(\mathbf{w}_{r'}, \mathbf{v}_1) \\
\mathbf{w}_{r'} \leftarrow \mathit{Reduce}(\mathbf{w}_{r'}, \mathbf{v}_2) \\
\vdots \\
\mathbf{w}_{r'} \leftarrow \mathit{Reduce}(\mathbf{w}_{r'}, \mathbf{v}_{r'-1})
\end{array}
\end{array}
$$

At the end of this part, we re-index the vectors in V'' form 1 to r'' in no particular order, and rename the vectors in list V'' from $\{\mathbf{w}_1, ..., \mathbf{w}_{r''}\}$ to $\{\mathbf{v}_1, ..., \mathbf{v}_{r''}\}$ at Step 34 in Alg.4. Recall that list V'' becomes pairwise-reduced because Reduce($\mathbf{v}_i, \mathbf{v}_j$) $= \mathbf{v}_i$ holds for all vectors $\mathbf{v}_i, \mathbf{v}_j \in V''$ with $i \ne j$. We then have relationship $V'' \subseteq V' \subseteq V$ and $|V''| = r'' \le r' \le r$.

This part also can be parallelized in a similar way as the first part. Let t be the number of threads and s' be the number of sample vectors treated by a thread, where $s' = \lfloor r'/t \rfloor$. Each thread i deals with only a sample list V' and runs $\mathbf{w}_k \leftarrow$ Reduce($\mathbf{w}_k, \mathbf{v}_j$), where $(i-1)s' + 1 \le k \le is', \mathbf{v}_j \in V'$ with $k \ne j$. When thread i computes $\mathbf{w}_i \leftarrow$ Reduce($\mathbf{w}_i, \mathbf{v}_j$), another thread h can compute $\mathbf{w}_h \leftarrow$Reduce($\mathbf{w}_h, \mathbf{v}_j$) for all $\mathbf{v}_j \in V'$. More specifically, in this part, each thread runs the Reduce algorithm in the following order in parallel.

$$\boxed{\text{Thread } 1}\quad \begin{array}{l} \mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \mathbf{v}_2) \\ \mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \mathbf{v}_3) \\ \vdots \\ \mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \mathbf{v}_{r'}) \\ \vdots \\ \mathbf{w}_s \leftarrow \mathit{Reduce}(\mathbf{w}_s, \mathbf{v}_2) \\ \mathbf{w}_s \leftarrow \mathit{Reduce}(\mathbf{w}_s, \mathbf{v}_3) \\ \vdots \\ \mathbf{w}_s \leftarrow \mathit{Reduce}(\mathbf{w}_s, \mathbf{v}_{r'}) \end{array}$$

$$\boxed{\text{Thread } 2}\quad \begin{array}{l} \mathbf{w}_{s'+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{s'+1}, \mathbf{v}_1) \\ \mathbf{w}_{s'+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{s'+1}, \mathbf{v}_2) \\ \vdots \\ \mathbf{w}_{s'+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{s'+1}, \mathbf{v}_{r'}) \\ \vdots \\ \mathbf{w}_{2s'} \leftarrow \mathit{Reduce}(\mathbf{w}_{2s'}, \mathbf{v}_1) \\ \mathbf{w}_{2s'} \leftarrow \mathit{Reduce}(\mathbf{w}_{2s'}, \mathbf{v}_2) \\ \vdots \\ \mathbf{w}_{2s'} \leftarrow \mathit{Reduce}(\mathbf{w}_{2s'}, \mathbf{v}_{r'}) \end{array}$$

$$\cdots$$

$$\boxed{\text{Thread } t}\quad \begin{array}{l} \mathbf{w}_{s'(t-1)+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{s'(t-1)+1}, \mathbf{v}_1) \\ \mathbf{w}_{s'(t-1)+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{s'(t-1)+1}, \mathbf{v}_2) \\ \vdots \\ \mathbf{w}_{s'(t-1)+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{s'(t-1)+1}, \mathbf{v}_{r'-1}) \\ \vdots \\ \mathbf{w}_{s't} \leftarrow \mathit{Reduce}(\mathbf{w}_{s't}, \mathbf{v}_1) \\ \mathbf{w}_{s't} \leftarrow \mathit{Reduce}(\mathbf{w}_{s't}, \mathbf{v}_2) \\ \vdots \\ \mathbf{w}_{s't} \leftarrow \mathit{Reduce}(\mathbf{w}_{s't}, \mathbf{v}_{r'}) \end{array}$$

4.5 Reduction of list vectors using sample vectors (Step 35 to 45 in Alg.4)

In this part, by reducing the vectors $\boldsymbol{\ell}_i$ in L using all sample vectors in $V'' = \{\mathbf{v}_1, \ldots, \mathbf{v}_{r''}\}$, we will construct the list L', which consists of vectors $\boldsymbol{\ell}_i \in L$ that satisfy $\mathrm{Reduce}(\boldsymbol{\ell}_i, \mathbf{v}_j) = \boldsymbol{\ell}_i$ for all $\mathbf{v}_j \in V''$. At the beginning of this part we assign $\mathbf{w}_i \leftarrow \boldsymbol{\ell}_i$ at Step 36 in Alg.4. For $i = 1, 2, ..., m$, this part runs $\mathrm{Reduce}(\mathbf{w}_i, \mathbf{v}_j)$ from $j = 1$ to r'' for the fixed first input $\mathbf{w}_i$ and updates $\mathbf{w}_i$ using its output repeatedly. During all reduction steps, if $\mathbf{w}_i$ is changed (*i.e.,* $\mathbf{w}_i \neq \mathrm{Reduce}(\mathbf{w}_i, \mathbf{v}_i)$ for some $\mathbf{v}_i$), this vector $\mathbf{w}_i$ is moved to stack S, otherwise, this vector $\mathbf{w}_i (= \boldsymbol{\ell}_i)$ is moved to the distributed list L'. This part runs the Reduce algorithm in the following order.

$$\begin{array}{lll} \mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \mathbf{v}_1) & \mathbf{w}_i \leftarrow \mathit{Reduce}(\mathbf{w}_i, \mathbf{v}_1) & \vdots \\ \mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \mathbf{v}_2) & \mathbf{w}_i \leftarrow \mathit{Reduce}(\mathbf{w}_i, \mathbf{v}_2) & \mathbf{w}_m \leftarrow \mathit{Reduce}(\mathbf{w}_m, \mathbf{v}_1) \\ \vdots & \vdots & \mathbf{w}_m \leftarrow \mathit{Reduce}(\mathbf{w}_m, \mathbf{v}_2) \\ \mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \mathbf{v}_{r''}) & \mathbf{w}_i \leftarrow \mathit{Reduce}(\mathbf{w}_i, \mathbf{v}_{r''}) & \vdots \\ \vdots & \vdots & \mathbf{w}_m \leftarrow \mathit{Reduce}(\mathbf{w}_m, \mathbf{v}_{r''}) \end{array}$$

At the end of this part, we re-index the vectors in L' form 1 to m' in no particular order, and rename the vectors in list L' from $\{\mathbf{w}_1, ..., \mathbf{w}_{m'}\}$ to $\{\boldsymbol{\ell}_1, ..., \boldsymbol{\ell}_{m'}\}$ at Step 45 in Alg.4. Recall that any vector $\boldsymbol{\ell}_i$ in list L' satisfies $\mathrm{Reduce}(\boldsymbol{\ell}_i, \mathbf{v}_j) = \boldsymbol{\ell}_i$ for all vectors $\mathbf{v}_j$ in list V''. We then have relationships $L' \subseteq L$ and $|L'| = m' \leq m$. After this part, our algorithm merges list L' and list V'' to become the new list $L = L' \cup V''$ at Step 46 in Alg.4.

This step can be simply parallelized without heavy overhead in a similar way as the first part, and the degree of parallelization is at most r''. Each thread of index i updates $\bar{s}$ vectors in list L_i (*i.e.,* $L_i = \{\boldsymbol{\ell}_{(i-1)\bar{s}+1}, \ldots, \boldsymbol{\ell}_{i\bar{s}}\}$, where $\bar{s} = \lfloor m/r'' \rfloor$). This part runs the Reduce algorithm in the following order in parallel.

$$\boxed{\text{Thread } 1}\quad \begin{array}{l} \mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \mathbf{v}_1) \\ \mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \mathbf{v}_2) \\ \vdots \\ \mathbf{w}_1 \leftarrow \mathit{Reduce}(\mathbf{w}_1, \mathbf{v}_{r''}) \\ \vdots \\ \mathbf{w}_{\bar{s}} \leftarrow \mathit{Reduce}(\mathbf{w}_{\bar{s}}, \mathbf{v}_1) \\ \mathbf{w}_{\bar{s}} \leftarrow \mathit{Reduce}(\mathbf{w}_{\bar{s}}, \mathbf{v}_2) \\ \vdots \\ \boldsymbol{\ell}_{\bar{s}} \leftarrow \mathit{Reduce}(\mathbf{w}_{\bar{s}}, \mathbf{v}_{r''}) \end{array}$$

$$\boxed{\text{Thread } 2}\quad \begin{array}{l} \mathbf{w}_{\bar{s}+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{\bar{s}+1}, \mathbf{v}_1) \\ \mathbf{w}_{\bar{s}+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{\bar{s}+1}, \mathbf{v}_2) \\ \vdots \\ \mathbf{w}_{\bar{s}+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{\bar{s}+1}, \mathbf{v}_{r''}) \\ \vdots \\ \mathbf{w}_{2\bar{s}} \leftarrow \mathit{Reduce}(\mathbf{w}_{2\bar{s}}, \mathbf{v}_1) \\ \mathbf{w}_{2\bar{s}} \leftarrow \mathit{Reduce}(\mathbf{w}_{2\bar{s}}, \mathbf{v}_2) \\ \vdots \\ \mathbf{w}_{2\bar{s}} \leftarrow \mathit{Reduce}(\mathbf{w}_{2\bar{s}}, \mathbf{v}_{r''}) \end{array}$$

$$\cdots$$

$$\boxed{\text{Thread } t}\quad \begin{array}{l} \mathbf{w}_{\bar{s}(t-1)+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{\bar{s}(t-1)+1}, \mathbf{v}_1) \\ \mathbf{w}_{\bar{s}(t-1)+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{\bar{s}(t-1)+1}, \mathbf{v}_2) \\ \vdots \\ \mathbf{w}_{\bar{s}(t-1)+1} \leftarrow \mathit{Reduce}(\mathbf{w}_{\bar{s}(t-1)+1}, \mathbf{v}_{r''}) \\ \vdots \\ \mathbf{w}_{\bar{s}t} \leftarrow \mathit{Reduce}(\mathbf{w}_{\bar{s}t}, \mathbf{v}_1) \\ \mathbf{w}_{\bar{s}t} \leftarrow \mathit{Reduce}(\mathbf{w}_{\bar{s}t}, \mathbf{v}_2) \\ \vdots \\ \mathbf{w}_{\bar{s}t} \leftarrow \mathit{Reduce}(\mathbf{w}_{\bar{s}t}, \mathbf{v}_{r''}) \end{array}$$

4.6 Properties of the proposed algorithm

In our algorithm, list L remains pairwise-reduced at any iteration for the following reasons. After the three reduction steps, our algorithm merges list L' and list V'' to become the new list $L = L' \cup V''$. Note that $\boldsymbol{\ell} = \text{Reduce}(\boldsymbol{\ell}, \mathbf{v})$ and $\mathbf{v} = \text{Reduce}(\mathbf{v}, \boldsymbol{\ell})$ hold for any vector $\boldsymbol{\ell}$ in L' and $\mathbf{v} \in V''$ by the first and third reduction part. And then, V'' is pair-wise reduced by the second part. Therefore, any pair of two vectors $(\boldsymbol{\ell}, \mathbf{v})$ in L', V'' is Gauss-reduced by Lemma 1, and thus the union $L = L' \cup V''$ becomes pairwise-reduced by Lemma 2.

Our algorithm is a natural extension of the Gauss Sieve algorithm. If only one vector is sampled (*i.e.,* $r = 1$), all the pairs of $(\boldsymbol{\ell}_j, \mathbf{v}_1)$ and $(\mathbf{v}_1, \boldsymbol{\ell}_j)$ are Gauss-reduced by the first and third reduction part, where $\boldsymbol{\ell}_j \in L$. There is nothing to do in the second reduction part. Therefore, this algorithm is equal to the Gauss Sieve algorithm when $r = 1$.

5 Implementation and Experimental Results

In this section, we explain the parallel implementations of the proposed parallel Gauss Sieve algorithm on a multicore CPU, and we also present some algorithmic improvement in our experiment.

5.1 Implementation using Amazon EC2

We use the instance cc1.8xlarge in AmazonEC2 [4]. Our experimental environment is shown in Table 1. Our implementation is based on the **gsieve** library, published by Voulgaris [30] and written in C++. We assume the following properties from our preliminary experiment:

- *all absolute values of entries of vectors are less than* 2^{16}
- *the computational cost of the inner product is dominant (Step 1 in Alg.1)*

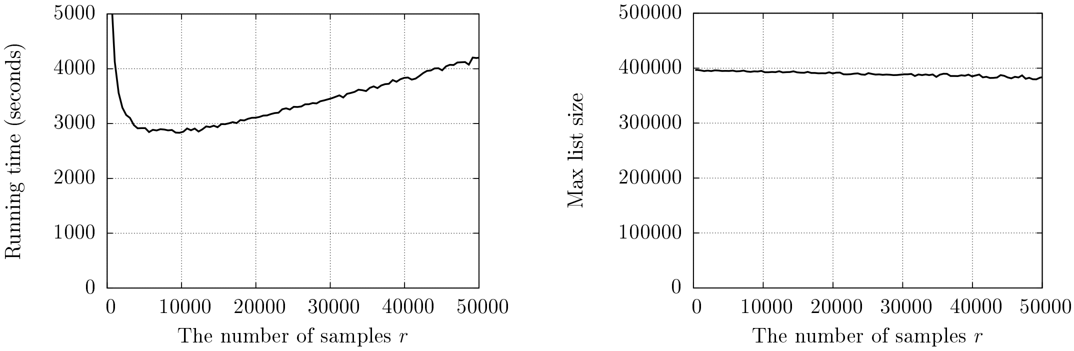
We optimize the code for the inner product (Step 1 in Alg.1) using the SIMD operation. Intel Xeon E5-2670 and g++4.1.2 support SSE4.2, and we can use a 128-bit SSE register. Using the SSE, we can treat 8 elements in one SSE operation in parallel. This technique enables our program to run about 4 times faster.

In this experiment, we fixed the number of threads at 32 per an instance, *i.e.,* at double the number of CPU cores, because the instance supports hyper-threading technology. For examples, we can use 2,688 threads in total for 84 instances.

In our algorithm, the total number of sample vectors is chosen as r. In the following, we derive a suitable value r from our experiment using one instance which solved the SVP of an 80-dimensional lattice from the SVP Challenge [25]. Figure 3(a) shows that the running time of solving the 80-dimensional SVP becomes relatively fast when the number of sample vectors r is in the range from about 4,000 to 10,000. Therefore, we select $r = 8192$, *i.e.,* one thread i treats $s = 256$ sample vectors in list $V_i = (\mathbf{v}_{256(i-1)+1}, \ldots, \mathbf{v}_{256i})$ on a lattice in the **Reduction sample vectors using list vectors** part, where $i = 1, 2, ..., 32$. On the other hand, Figure 3(b) shows the maximum size of the list L for solving the 80-dimensional SVP on the same lattice. Interestingly, the size of the list L in our parallel Gauss Sieve algorithm does not grow, even though the number of samples r increases.

Table 1. Experiment environment used in our experiment

Instance type	cc1.8xlarge
CPU	Intel Xeon E5-2670 2.6GHz ×2
Core	8 core ×2
Memory	64GB
OS	Ubuntu12.10
Compiler	g++ 4.1.2
Library	OpenMP, OpenMPI, NTL5.5.2



(a) Running time (b) Maximum size of list L

Fig. 3. Results for solving the 80-dimensional SVP on a lattice. Fig (a) shows the running time using one instance (32 threads). Fig (b) shows the maximum size of list L. The horizontal axis indicates the number of samples r.

5.2 Sampling Short Vectors(Step 8 in Alg.4)

In the **gsieve** library [30], Klein's randomized rounding algorithm [13] is implemented. The details of the algorithm are explained by Gentry *et al.* [11].

If we are able to sample shorter vectors at Step 9 in Alg.4, then the running time of the proposed Gauss Sieve algorithm can be improved. However, it takes longer time to sample such shorter vectors on a lattice in general. Therefore, we try to adjust the parameter which determines the tradeoff between the length of the norm of sample vectors and the running time of our algorithm.

Indeed we adjust the parameter of the core subroutine, namely the *SampleD* algorithm described in [11]. For the two inputs (u, c), *SampleD* chooses an integer x from the range $[c - u \cdot d, c + u \cdot d]$, where $d = \log(n)$ in the **gsieve** library. We determine a more suitable value of d instead of $d = \log(n)$. The *SampleD* outputs x with probability $\rho_{u,c}(x - c)$, otherwise repeats choosing x, where $\rho_{u,c}(x)$ denoted a Gaussian function on $\mathbb{R}$ that is defined by $\rho_{u,c}(x) = \exp(-\pi|x - c|^2/u^2)$ for any $x \in \mathbb{R}$. Klein's sampling algorithm generates a new vector on a lattice that is a linear combination of the basis vector and the coefficient vector $\mathbf{x}$ using the *SampleD* algorithm n times. Therefore, if the *SampleD* algorithm outputs a smaller integer, Klein's sampling algorithm outputs a shorter vector. However, the computational time of the *SampleD* algorithm increases as the length of the output vector decreases. In our experiment, we found the parameter $d = \log n/70$ which is most suitable for speeding up of our algorithm. This technique enables our program to run about two times faster.

5.3 Improvement of the Ideal Gauss Sieve

In this section, we show some techniques for accelerating the Ideal Gauss Sieve algorithm [24] by using the bi-directional rotation structure of some cyclotomic polynomials.

Selecting Cyclotomic Polynomials In [24], there are three types of ideal lattices generated by specific polynomials (including two cyclotomic polynomials), which are suitable for the rotate operation $\mathbf{rot}(\mathbf{v})$ of a vector $\mathbf{v}$. We define new types of an ideal lattice, which is called a *Trinomial lattice.* A *Trinomial lattice* is generated by the trinomials in the cyclotomic polynomials. There are two conditions for a *Trinomial lattice*, as follows:

- *Condition 1*
  If $n/2$ is a power of three, where n is an even dimension of a lattice, an ideal lattice generated by a cyclotomic polynomial $\boldsymbol{g}(x) = x^n + x^{n/2} + 1$ is called a *Trinomial lattice.* In this type, the rotation of vector $\mathbf{v}$ is $\mathbf{rot}(\mathbf{v}) = (-v_{n-1}, v_0, \ldots, v_{\frac{n}{2}-2}, v_{\frac{n}{2}-1} - v_{n-1}, v_{\frac{n}{2}}, \ldots, v_{n-2})$.
- *Condition 2*

Table 2. Results of the SVP Challenge [25] and Ideal Lattice Challenge [20]. The 96 and 128-dimensional SVPs in the Ideal Lattice Challenge are generated by cyclotomic polynomials $x^{96} - x^{48} + 1$ and $x^{128} + 1$, respectively. The other SVPs are non-special type.

	dimension	CPU hours	#instance	#thread t	#sample vectors r	type
SVP Challenge	80	0.9	1	32	8,192	*Random lattice*
	96	200	4	128	32,768	*Random lattice*
Ideal Lattice Challenge	80	0.9	1	32	8,192	*Ideal lattice*
	96	8	1	32	8,192	*Trinomial lattice*
	128	29,994	84	2,688	688,128	*Anti-cyclic lattice*

If the dimension n is the product of both a power of two and a power of three, an ideal lattice generated by the cyclotomic polynomial $\boldsymbol{g}(x) = x^n - x^{n/2} + 1$ is called a *Trinomial lattice*. In this type, the rotation of vector $\mathbf{v}$ is $\mathbf{rot}(\mathbf{v}) = (-v_{n-1}, v_0, \ldots, v_{\frac{n}{2}-2}, v_{\frac{n}{2}-1} + v_{n-1}, v_{\frac{n}{2}}, \ldots, v_{n-2})$.

Gallot proved that the polynomials that satisfy the above conditions are cyclotomic polynomials (Theorem 3.1 in [7]). The rotate operation $\mathbf{rot}(\mathbf{v})$ using the *Trinomial lattice* requires no greater computational cost than that using the *Anti-cyclic lattice*.

Inverse Rotation In [24], the rotation $\mathbf{rot}(\mathbf{v})$ of vector $\mathbf{v}$ is used for accelerating the Gauss Sieve algorithm. In the same manner, we define the *inverse rotation* $\mathbf{rot}^{-1}(\mathbf{v})$ of vector $\mathbf{v}$. The inverse rotation is defined by $\mathbf{rot}^{-1}(\mathbf{v}) = x^{-1}\mathbf{v} \bmod \boldsymbol{g}(x)$, where $\mathbf{v}(x)$ is a polynomial representation of vector $\mathbf{v}$. In *Trinomial lattice*, x^{-1} is $-x^{n-1} \pm x^{\frac{n}{2}-1}$. The computational costs of $\mathbf{rot}(\mathbf{v})$ and $\mathbf{rot}^{-1}(\mathbf{v})$ using the *Prime cyclotomic lattice*, *Anti-cyclic lattice* and the *Trinomial lattice* are quite similar.

Updating Vectors In a *Trinomial lattice*, we can find shorter vectors using rotation and inverse rotation. If $\mathbf{v} \in \mathcal{L}(\mathbf{B})$, the difference of the Euclidean norm between vector $\mathbf{v}$ and rotated vector $\mathbf{rot}(\mathbf{v})$ (or inversely rotated vector $\mathbf{rot}^{-1}(\mathbf{v})$) is represented as follows:

$$||\mathbf{rot}(\mathbf{v})|| - ||\mathbf{v}|| = (v_{n-1})^2 + 2v_{\frac{n}{2}-1}v_{n-1},$$
$$||\mathbf{rot}^{-1}(\mathbf{v})|| - ||\mathbf{v}|| = (v_0)^2 + 2v_{\frac{n}{2}}v_0.$$

Therefore, the Euclidean norm of a(n) (inversely) rotated vector becomes shorter than that of the original vector with a non-negligible probability.

In a *Trinomial lattice*, repeating the rotate operation increases the norm gradually. Therefore, the total running time of our algorithm increases with too large number of the rotate operation. Then we derived the most suitable number of the rotate operation from the experiment to solve a 72-dimensional SVP with each number of rotations. In our experiment, it was found that the most suitable number was 6, and this technique enables our parallel Gauss Sieve algorithm to run about 5.5 times faster.

5.4 Solving the Challenges

We have solved several problems in the SVP Challenge [25] and Ideal Lattice Challenge [20]. The problem setting in these challenges has been published in [21]. We pre-computed the BKZ-reduced basis with a block size of 30 using NTL library [29]. Because this precomputation requires much less time than the Gauss Sieve algorithm, we do not include the timing in the following. In our experiment, we used the instance cc1.8xlarge described in Table 1. We fix the number of threads at 32 per an instance. We show the results of our experiments in Table 2.

In the SVP Challenge, we solved the 80- and 96-dimensional SVPs. Both of these problems are random lattices given as filename “svpchallengedim80seed0.txt” and “svpchallengedim90seed0.txt” on the SVP Challenge [25]. As we explained in section 5.1, our parallel algorithm solved the 80-dimensional SVP in about one CPU hour using one instance which deploys 32 threads and 8,192 sample vectors. According to the results of Schneider *et al.* [22], their program for the Gauss Sieve requires about 10^6 seconds $\approx$ 278 hours using one thread for the same problem. Hence, our parallel algorithm enables the Gauss Sieve algorithm to run about like 200 times faster. We also solved the 96-dimensional SVP using four instances of 128 threads and 32,768 sample vectors. As a result, our parallel algorithm required about 200 CPU hours. This 96-dimensional SVP is the largest problem that has been solved to date by using a sieving algorithm on a random lattice.

In the Ideal Lattice Challenge, we solved the 80-, 96- and 128-dimensional SVPs [20]. In this challenge, a basis of n-dimensional ideal lattice is generated from one of cyclotomic polynomials of degree n. In our experiment we chose the 80-dimensional lattice generated by cyclotomic polynomial $g(x) = x^{80} + x^{78} - x^{70} - x^{68} + x^{60} - x^{56} - x^{50} + x^{46} + x^{40} + x^{34} - x^{30} - x^{24} + x^{20} - x^{12} - x^{10} + x^2 + 1$ given as a filename "ideallatticedim80index220seed0.txt". The basis of 96-dimensional lattice was selected to be a *Trinomial lattice* generated by $g(x) = x^{96} - x^{48} + 1$ given as filename "ideallatticedim96index288seed0.txt", and that of 128-dimensional SVP was selected to be an *Anti-cyclic lattice* generated by cyclotomic polynomial $g(x) = x^{128} + 1$ given as filename "ideallatticedim128index256seed0.txt". In our experiment of the 80-dimensional ideal lattice our parallel algorithm required about one CPU hour using 32 threads and 8,192 sample vectors, which are the same time cost compared with our above experiment for a random lattice in the SVP Challenge. Additionally, in our experiment of the 96-dimensional ideal lattice, our parallel algorithm required about 8 CPU hours using 32 threads and 8,192 sample vectors. The proposed two techniques (*inverse rotation* and *updating vectors*) enable us to speedup about 25 times faster than the random lattice of the same dimension.

In our experiment of the 128-dimensional ideal lattice, our parallel algorithm require $29,994$ CPU hours using 84 instances, where we can set that the number of total threads and sample vectors are $t = 2,688$ and $r = 688,128$, respectively. As a result, our parallel algorithm outputs a short vector:

```
(-613, -20, -146, -249, 237, 161, 290, 518, -204, -207, -39, 333, -97, 30, 53, 579,
93, -634, 297, 223, -201, 75, -98, -85, -68, 100, 21, -87, -442, -63, -211, 358, -143,
239, -39, 240, -9, -382, -38, -285, -10, 275, 108, 116, -288, -165, 509, 589, 445,
-137, -230, -131, -84, -26, -37, 442, -115, 267, 642, 168, -226, 361, 212, -193, 379,
59, 45, 215, -48, -12, 53, 48, 83, -156, 184, -103, 102, -427, -400, 363, -69, -142,
562, -145, -118, -51, -31, -96, 604, 260, -371, -361, -553, -292, -222, 74, -51, 179,
-162, -431, -24, 159, -180, 8, -85, 57, 264, 157, 4, -232, 272, -638, -58, 68, 3, 314,
-11, -395, -88, -129, -29, 219, -223, -186, 42, 73, 399, -146).
```

The Euclidean norm of this vector is 2,959 which is shorter than the heuristic bound $(1.05/\sqrt{\pi})\Gamma(\frac{n}{2}+1)^{\frac{1}{n}} \cdot \det(\mathcal{L}(\mathbf{B}))^{\frac{1}{n}}$ of the shortest vector in $\mathcal{L}(\mathbf{B})$ of $n = 128$.

According to the estimated complexity $2^{0.52n}$ [17] and the our implementation result of the 96-dimensional SVP, if the time complexity of our algorithm is the same as the Gauss Sieve algorithm, solving the 128-dimensional SVP over a lattice requires $200 \times 2^{0.52\Delta n} \approx 2.0 \times 10^7$ CPU hours using the parallel Gauss Sieve algorithm. Therefore, we heuristically estimate that solving the 128-dimensional SVP over an *Anti-cyclic lattice* is about 600 times faster than that over random lattices.

6 Conclusion

In this paper, we proposed a parallel Gauss Sieve algorithm, which can solve the shortest vector problem (SVP) using a large number of threads. We implemented the proposed parallel Gauss Sieve algorithm by the SIMD operation in AmazonEC2 which supports hyper-threading technology. Our experiment deploys 32 threads per instance cc1.8xlarge of 16 CPU cores. Then we tried to solve the SVP Challenge and the Ideal Lattice Challenge from TU Darmstadt (`http://www.latticechallenge.org/`).

In the case of solving the SVP of 80 dimensions, the proposed parallel algorithm enables the Gauss Sieve algorithm to run about 200 times faster than the previous implementation using a single thread by Schneider *et al.* in the same lattice. Then we have solved the SVP Challenge of 96 dimensions by the proposed parallel Gauss Sieve algorithm using 32 threads in 200 CPU hours. That is a new record for solving the SVP using the sieving algorithms in the SVP Challenge.

We further considered some speed-up of the proposed parallel Gauss Sieve algorithm by the bi-directional rotation property of ideal lattices. Then we successfully solved a 128-dimensional SVP on the ideal lattice generated by the cyclotomic polynomial $x^{128} + 1$, where this type of ideal lattice is often used for efficient implementation of lattice-based cryptography. Our experiment requires 29,994 CPU hours by executing 2,688 threads over 84 instances in total. To the best of our knowledge, this is currently the highest dimensions of solving the SVP in ideal lattices. We believe that our results will contribute to estimating a secure key length for lattice-based cryptography.

References

1. M. Ajtai. The Shortest Vector Problem in L^2 is NP-hard for Randomized Reductions (Extended Abstract). In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, STOC'98, pages 10–19. ACM, 1998.
2. M. Ajtai and C. Dwork. A Public-key Cryptosystem with Worst-case/average-case Equivalence. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, STOC'97, pages 284–293. ACM, 1997.
3. M. Ajtai, R. Kumar, and D. Sivakumar. A Sieve Algorithm for the Shortest Lattice Vector Problem. In *Proceedings of the 33th Annual ACM Symposium on Theory of Computing*, STOC'01, pages 601–610. ACM, 2001.
4. Amazon. Amazon Elastic Compute Cloud. Available at `http://aws.amazon.com/jp/ec2/`.
5. V. Arvind and P. S. Joglekar. Some Sieving Algorithms for Lattice Problems. In *Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, FSTTCS'08, volume 2 of *LIPIcs*, pages 25–36. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008.
6. J. Blömer and S. Naewe. Sampling Methods for Shortest Vectors, Closest Vectors and Successive Minima. *Journal of Theoretical Computer Science*, volume 410, issue 18, pages 1648–1665, 2009.
7. Y. Gallot. Cyclotomic Polynomials and Prime Numbers, 2000. Available at Gallot's homepage `http://yves.gallot.pagesperso-orange.fr/papers/cyclotomic.pdf`.
8. N. Gama, P. Nguyen, and O. Regev. Lattice Enumeration Using Extreme Pruning. In *Proceedings of the 29th Annual International Conference on Theory and Application of Cryptographic Techniques*, Eurocrypt'10, volume 6110 of *LNCS*, pages 257–278. Springer, 2010.
9. S. Garg, C. Gentry, and S. Halevi. Candidate Multilinear Maps from Ideal Lattices. Cryptology ePrint Archive, Report 2012/610, 2012.
10. C. Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, STOC'09, pages 169–178. ACM, 2009.
11. C. Gentry, C. Peikert, and V. Vaikuntanathan. Trapdoors for Hard Lattices and New Cryptographic Constructions. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, STOC'08, pages 197–206. ACM, 2008.
12. J. Hoffstein, J. Pipher, and J. Silverman. NTRU: A Ring-based Public Key Cryptosystem. In *Algorithmic Number Theory*, volume 1423 of *LNCS*, pages 267–288. Springer, 1998.
13. P. Klein. Finding the Closest Lattice Vector When it's Unusually Close. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'00, pages 937–941. ACM, 2000.
14. A. Lenstra, H. Lenstra, and L. Lovász. Factoring Polynomials with Rational Coefficients. *Journal of Mathematische Annalen*, volume 261, issue 4, pages 515–534, 1982.
15. D. Micciancio. The Shortest Vector in a Lattice is Hard to Approximate to within Some Constant. In *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, FOCS'98, pages 92–98. IEEE Computer Society, 1998.
16. D. Micciancio and P. Voulgaris. A Deterministic Single Exponential Time Algorithm for Most Lattice Problems Based on Voronoi Cell Computations. In *Proceedings of the 42nd ACM Symposium on Theory of Computing*, STOC'10, pages 351–358. ACM, 2010.
17. D. Micciancio and P. Voulgaris. Faster Exponential Time Algorithms for the Shortest Vector Problem. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'10, volume 65, pages 1468–1480. SIAM, 2010.
18. B. Milde and M. Schneider. A Parallel Implementation of GaussSieve for the Shortest Vector Problem in Lattices. In *Proceedings of the 11th International Conference on Parallel Computing Technologies*, volume 6873 of *LNCS*, pages 452–458. Springer, 2011.
19. P. Q. Nguyen and T. Vidick. Sieve Algorithms for the Shortest Vector Problem Are Practical. *Journal of Mathematical Cryptology*, volume 2, pages 181–207, 2008.
20. T. Plantard and M. Schneider. Ideal Lattice Challenge. `http://www.latticechallenge.org/ideallattice-challenge/`.

21. T. Plantard and M. Schneider. Creating a Challenge for Ideal Lattices. Cryptology ePrint Archive, Report 2013/039, 2013.
22. M. Schneider. Analysis of Gauss-Sieve for Solving the Shortest Vector Problem in Lattices. In *Proceedings of the 5th International Workshop of Algorithms and Computation*, WALCOM'11, volume 6552 of *LNCS*, pages 89–97. Springer, 2011.
23. M. Schneider. *Computing Shortest Lattice Vectors on Special Hardware*. PhD thesis, Technische Universität Darmstadt, 2011.
24. M. Schneider. Sieving for Shortest Vectors in Ideal Lattices. Cryptology ePrint Archive, Report 2011/458, 2011.
25. M. Schneider and N. Gama. SVP Challenge. `http://www.latticechallenge.org/svp-challenge/`.
26. C.-P. Schnorr. A Hierarchy of Polynomial Time Lattice Basis Reduction Algorithms. *Journal of Theoretical Computer Science*, volume 53, issue 2-3, pages 201–224, 1987.
27. C.-P. Schnorr. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. *Journal of Mathematical programming*, pages 181–191. Springer, 1993.
28. C.-P. Schnorr and H. H. Horner. Attacking the Chor-Rivest Cryptosystem by Improved Lattice Reduction. In *Proceedings of the 14th annual international conference on Theory and application of cryptographic techniques*, Eurocrypt'95, pages 1–12. Springer, 1995.
29. V. Shoup. Number Theory Library (NTL) for C++. Available at Shoup's homepage `http://shoup.net/ntl`.
30. P. Voulgaris. Gauss Sieve beta 0.1, 2010. Available at Voulgaris' homepage at the University of California, San Diego `http://cseweb.ucsd.edu/~pvoulgar/impl.html`.