Massive Parallelization for Finding Shortest Lattice Vectors Based on Ubiquity Generator Framework

Tateiwa, Nariaki
Graduate School of Mathematics, Kyushu University

Shinano, Yuji
Mathematical Algorithmic Intelligence, Applied Algorithmic Intelligence Methods (A²IM), Zuse Institute Berlin (ZIB)

Nakamura, Satoshi
NTT Secure Platform Laboratories

Yoshida, Akihiro
Graduate School of Mathematics, Kyushu University

他

出版情報：SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, 2021-02-22. Institute of Electrical and Electronics Engineers :IEEE
バージョン：

Massive Parallelization for Finding Shortest Lattice Vectors Based on Ubiquity Generator Framework

Nariaki Tateiwa
Graduate School of Mathematics
Kyushu University
Fukuoka, Japan
n-tateiwa@kyudai.jp

Yuji Shinano
Mathematical Algorithmic Intelligence
Applied Algorithmic Intelligence Methods (A²IM)
Zuse Institute Berlin (ZIB)
Berlin, Germany
ORCID: 0000-0002-2902-882X

Satoshi Nakamura
NTT Secure Platform Laboratories
Tokyo, Japan
satoshi.nakamura.xn@hco.ntt.co.jp

Akihiro Yoshida
Graduate School of Mathematics
Kyushu University
Fukuoka, Japan
akihiro.yoshida.916@kyudai.jp

Shizuo Kaji
Institute of Mathematics for Industry
Kyushu University
Fukuoka, Japan
ORCID: 0000-0002-7856-6536

Masaya Yasuda
Department of Mathematics
Rikkyo University
Tokyo, Japan
ORCID: 0000-0002-1534-5648

Katsuki Fujisawa
Institute of Mathematics for Industry
Kyushu University
Fukuoka, Japan
ORCID: 0000-0001-8549-641X

***Abstract*—Lattice-based cryptography has received attention as a next-generation encryption technique, because it is believed to be secure against attacks by classical and quantum computers. Its essential security depends on the hardness of solving the shortest vector problem (SVP). In the cryptography, to determine security levels, it is becoming significantly more important to estimate the hardness of the SVP by high-performance computing. In this study, we develop the world's first distributed and asynchronous parallel SVP solver, the MAssively Parallel solver for SVP (MAP-SVP). It can parallelize algorithms for solving the SVP by applying the Ubiquity Generator framework, which is a generic framework for branch-and-bound algorithms. The MAP-SVP is suitable for massive-scale parallelization, owing to its small memory footprint, low communication overhead, and rapid checkpoint and restart mechanisms. We demonstrate its performance and scalability of the MAP-SVP by using up to 100,032 cores to solve instances of the Darmstadt SVP Challenge.**

***Index Terms*—Lattice based cryptography, Shortest vector problem, Parallel computation, DeepBKZ, ENUM, Ubiquity Generator Framework**

I. Introduction

A *lattice* is a discrete subgroup of the Euclidean space, $\mathbb{R}^n$. A lattice L of dimension n is spanned by a *basis* $\mathbf{B}$ consisting of linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^n$, and any vector in L can be represented as a linear combination of the $\mathbf{b}_i$s with integer coefficients. In the past few years, lattices have attracted considerable interest in cryptography. In particular, with the recent development of quantum computers, since 2015, the US National Institute of Standards and Technology (NIST) started developing new standards for *post-quantum cryptography (PQC)* and called for proposals to prepare information security systems that can resist quantum computers [1]. (cf., The most popular cryptographic systems, such as RSA, DSA, and ECDSA, could be broken by Shor's algorithms [2] with the use of large-scale quantum computers.) In 2019, NIST allowed 26 proposals for the second round of the NIST PQC Standardization Process, among which 12 were based on lattices.

The most famous computational problem in lattices is the *shortest vector problem (SVP)* that asks us to find a non-zero shortest vector in a given lattice. Its hardness ensures the security of lattice-based cryptography. The Darmstadt SVP Challenge [3] is a recognized venue for testing algorithms for solving SVPs; it publicly lists sample bases of dimensions from 40 up to 200. It is a contest of finding shorter vectors, not necessarily the shortest one. Specifically, any non-zero lattice vector whose length is shorter than $(1.05 \cdot \mathrm{GH}(L))$ can be submitted for each lattice L, where $\mathrm{GH}(L)$ is the expectation of the length of the non-zero shortest vector in L (see Equation (1) below). Specifically, it is an approximate SVP with a factor 1.05 (see [4], [5] for relaxed variants).

There are two main algorithms to find a non-zero shortest lattice vector; *Sieve* and *ENUM*. Both the algorithms perform an exhaustive search of all the short lattice vectors, whose number is exponential in the lattice dimension. In the SVP Challenge, most of the high-dimensional records have been achieved by a variant of the sieve algorithm. The sieve

SC20, November 9-19, 2020, Is Everywhere We Are
978-1-7281-9998-6/20/$31.00 ©2020 IEEE

algorithm searches for the shortest vector by repeatedly storing short differences between the short lattice vectors. A high-dimensional SVP instance requires numerous vectors to be stocked. Specifically, it requires a memory that is exponential in the dimension of the input lattice. According to [6, Table 2], G6K , a solver of the sieve algorithm, uses approximately 246 GB of memory for solving 127-dimensional SVP instances. Hence, it is highly difficult to satisfy memory requirements for solving high-dimensional SVPs using the sieve algorithm, even by increasing the number of processes. In contrast, ENUM is asymptotically slower than the sieve algorithm, but its space-complexity is polynomial in the lattice dimension. Therefore ENUM is more essential for solving SVP in higher dimensions. In particular, it is essential for cryptanalysis of lattice-based cryptography, since it uses very high dimensions such as 256 and 512 for cryptographic security.

In this paper, we propose a new SVP solver called as the ***Ma**ssively **P**arallel Solver for **SVP*** (MAP-SVP), which is suitable for large-scale parallelization. The MAP-SVP is the first practical asynchronous distributed-memory solver of SVP. (A Voronoi-Cell-based asynchronous parallel solver was already proposed by [7]. However, it is difficult for this solver to solve more than even a 21–23-dimensional SVP, because of the large amount of memory required.) We demonstrate the scalability and performance of the MAP-SVP through numerical experiments on SVP instances of up to 127 dimensions. We succeed in employing $100,032$ cores in our experiment. To the best of our knowledge, this is the most massive-scale experiment for solving SVP. In MAP-SVP , multiple processes independently execute two SVP solving algorithms, *ENUM* and *DeepBKZ*, which have a small memory footprint. The memory complexity of each process is $O(n^2)$ with respect to dimension n of SVP, and in our numerical experiments, the memory usage is less than 0.013 GB per process for even a 155-dimensional SVP instance. Therefore, we can execute the MAP-SVP for high-dimensional SVPs and obtain the shortest vector. The MAP-SVP is implemented by the specialized Ubiquity Generator (UG) framework with the *parallelDispatch* function. The UG is a generic framework to parallelize branch-and-bound based solvers and has achieved large-scale MPI parallelism with 80,000 cores [8]. Owing to parallelDispatch of the UG , we can stably run a massive number of processes sharing information asynchronously with low communication overhead. In addition, we extend parallelDispatch by implementing the vector pooling feature, which allows each process to receive short vectors found by other processes as needed.

The overview of the MAP-SVP for a two-dimensional SVP is shown in Fig. 1. Our system is composed of a management process, called as `LoadCoordinator` (abbreviated to `LC` throughout this paper), and multiple `Solvers`. We provide a lattice basis B as an SVP instance to the `LC`, then `LC` distributes it to each `Solver`. `Solver` randomizes basis after receive it. In Fig. 1, the solid arrows, points, and circles represent the lattice basis, lattice, and depth-first search space of ENUM, respectively. The radius of the circle is the length of

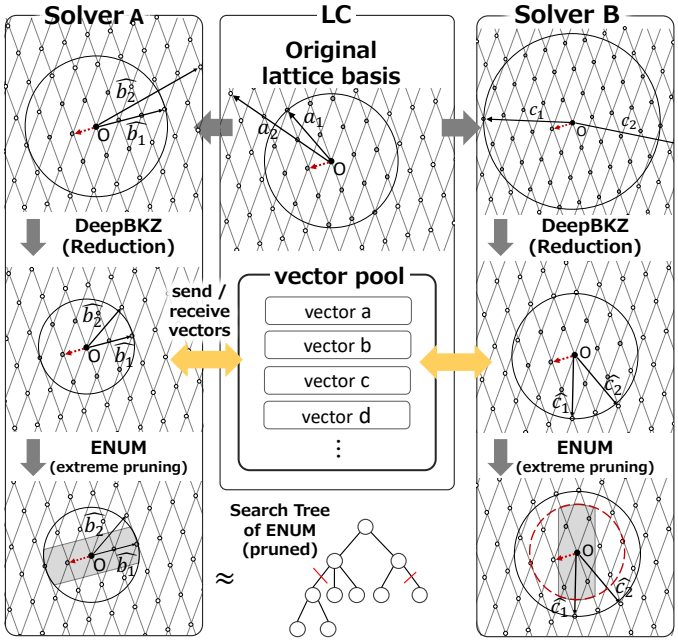

Fig. 1. Overview of our new SVP solver

the shortest vector in the current bases. Each `Solver` executes the DeepBKZ and ENUM algorithms while sharing short vectors via the *vector pool* managed by the `LC`. The DeepBKZ algorithm modifies the basis vector so that each vector is shortened. We apply DeepBKZ as a pre-processing for ENUM, to reduce its search space. We can set the radius of the search space of each `Solver` as that of the smallest solver, as depicted by the dashed red circle for `Solver` B in Fig. 1. This operation does not impair the optimally of the entire system. Subsequently, each `Solver` uses a technique called *extreme pruning* [9], to further reduce the search space in ENUM. Because the input lattice basis is randomized, the reduced search spaces for all the `Solvers` are also fundamentally different. In Fig. 1, the gray area and the dashed red arrow represent the reduced search space and the shortest vector, respectively. The MAP-SVP prunes the search tree of each `Solver` according to the theoretically computed probability of the shortest vector lying within a search sub-tree. Therefore, as the number of `Solvers` increases, a more aggressive pruning can be applied to reduce the computation time of each `Solver`. In addition, implementing the checkpoints and restarts of DeepBKZ and ENUM requires each `Solver` to only record and load the current lattice bases and the node of the ENUM search tree. For an n-dimensional SVP, the data size to be recorded is only $n^2 + n$.

In summary, our contributions are as follows: (1) We developed the world's first distributed and asynchronous parallel SVP solver called MAP-SVP which is suitable for large-scale parallelization based on MPI with a small memory footprint. Moreover, it has a low communication overhead with an excellent checkpoint and restart functionality. (2) We extended

the general framework UG to enable the asynchronous sharing of vectors. (3) We demonstrated the scalability of the MAP-SVP through large-scale numerical experiments. The numerical experiments suggest linear scalability between the log of the calculation time and the log of the number of processes. The scale of our experiments using 100,032 cores is the largest in SVP research. We tackled instances of the SVP Challenge using our parallel application and achieved new records for 104, 111, 121 and 127 dimensions.

II. Preliminaries: Lattices and SVP

In this section, we briefly review definitions and properties on lattices. We also present SVP and its solving algorithms.

A. Lattices and their bases

For an integer $n \geq 1$, let $\mathbf{b}_1, \ldots, \mathbf{b}_n$ be n linearly independent (column) vectors in $\mathbb{R}^n$. The set of all *integral* linear combinations of the $\mathbf{b}_i$'s is a (full-rank) *lattice*

$$L = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_n) := \left\{ \sum_{i=1}^{n} v_i \mathbf{b}_i : v_i \in \mathbb{Z} \ (1 \leq \forall i \leq n) \right\}$$

of dimension n with basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n) \in \mathbb{R}^{n \times n}$. A basis is represented by an $n \times n$ matrix whose column vectors span the lattice. Every lattice has infinitely many bases when $n \geq 2$; if two bases $\mathbf{B}_1$ and $\mathbf{B}_2$ span the same lattice, then there exists an $n \times n$ unimodular matrix $\mathbf{U}$ satisfying $\mathbf{B}_1 = \mathbf{B}_2\mathbf{U}$ (An integral square matrix with determinant ± 1 is called unimodular). The *volume* of L is defined as $\mathrm{vol}(L) = |\det(\mathbf{B})|$, which is independent of the choice of bases of L. The *Gram-Schmidt orthogonalization* for a basis $\mathbf{B}$ is the orthogonal family $\mathbf{B}^* = (\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*)$, recursively defined by $\mathbf{b}_1^* = \mathbf{b}_1$ and for $2 \leq i \leq n$

$$\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^* \text{ with } \mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\|\mathbf{b}_j^*\|^2} \ (j < i).$$

Set $\mu = (\mu_{i,j})$, where we let $\mu_{i,j} = 0$ for all $i < j$ and $\mu_{k,k} = 1$ for all k. Then $\mathbf{B} = \mathbf{B}^*\mu$, and thus $\mathrm{vol}(L) = \prod_{i=1}^{n} \|\mathbf{b}_i^*\|$ by the orthogonality of Gram-Schmidt vectors. Let π_ℓ denote the orthogonal projection onto the orthogonal complement of the $\mathbb{R}$-vector space $\langle \mathbf{b}_1, \ldots, \mathbf{b}_{\ell-1} \rangle_{\mathbb{R}}$ defined by

$$\pi_\ell : \mathbb{R}^n \longrightarrow \langle \mathbf{b}_1, \ldots, \mathbf{b}_{\ell-1} \rangle_{\mathbb{R}}^{\perp} = \langle \mathbf{b}_\ell^*, \ldots, \mathbf{b}_n^* \rangle_{\mathbb{R}},$$
$$\pi_\ell(\mathbf{x}) = \sum_{i=\ell}^{n} \frac{\langle \mathbf{x}, \mathbf{b}_i^* \rangle}{\|\mathbf{b}_i^*\|^2} \mathbf{b}_i^* \text{ for } \mathbf{x} \in \mathbb{R}^n.$$

Note that this projection map π_ℓ depends on the basis. We set $\pi_1 = \mathrm{id}$ (the identity map) for convenience.

B. The Shortest Vector Problem: SVP

Given a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ of a lattice L, SVP asks to find a shortest non-zero vector $\mathbf{s}$ in L. The length $\|\mathbf{s}\|$ of a shortest non-zero vector in a lattice L is denoted by $\lambda_1(L)$. It was proven in [10] that SVP is NP-hard under randomized reductions. The approximate SVP of an approximation factor $\alpha > 1$ asks to find a non-zero vector $\mathbf{z}$ in a lattice L such that $\|\mathbf{z}\| \leq \alpha \lambda_1(L)$.

a) An expectation of $\lambda_1(L)$: Given a lattice L of dimension n and a measurable set S in $\mathbb{R}^n$, the *Gaussian Heuristic* predicts that the number of vectors in $L \cap S$ is roughly equal to $\mathrm{vol}(S)/\mathrm{vol}(L)$. By applying to the ball centered at the origin in $\mathbb{R}^n$ with radius $\lambda_1(L)$, it leads to the prediction of the norm of a shortest non-zero vector in L. Specifically, the expectation of $\lambda_1(L)$ according to the Gaussian Heuristic is given by

$$\mathrm{GH}(L) := \nu_n^{-\frac{1}{n}} \mathrm{vol}(L)^{\frac{1}{n}} \sim \sqrt{\frac{n}{2\pi e}} \mathrm{vol}(L)^{\frac{1}{n}}, \qquad (1)$$

where ν_n denotes the volume of the unit ball in $\mathbb{R}^n$. This is only a heuristic, but for "random" lattices, $\lambda_1(L)$ is asymptotically equal to $\mathrm{GH}(L)$ with overwhelming probability [10].

b) The Darmstadt SVP Challenge: To test algorithms solving SVP, sample lattice bases have been presented on the webpage of [3] for dimensions from 40 to 200 (such lattices are random in the sense of [11]). For every lattice L, any non-zero lattice vector with norm less than $1.05\mathrm{GH}(L)$ can be submitted to the hall of fame in the SVP Challenge. Namely, it is not a contest of exact-SVP, but of *approximate-SVP* with factor 1.05 (The name seems somewhat misleading). To enter the hall of fame, a non-zero lattice vector is required to be shorter than previous records in the same dimension (with possibly different seed). Hence not all lattice vectors in the hall of fame are necessarily the shortest in every dimension.

C. Solving SVP algorithms

Here we present algorithms for solving SVP. They are classified into exact and approximate ones, and both are *complementary*. Exact algorithms should apply an approximation algorithm as a preprocessing to reduce their expensive cost.

1) Exact-SVP algorithms: Exact algorithms find a non-zero shortest lattice vector, but they are expensive. They basically perform an exhaustive search of all short vectors, whose number is exponential in dimension. These algorithms can be split into two categories; enumeration and sieve below.

a) Enumeration (Polynomial-space algorithm): It is an exhaustive search for an integer combination of the basis vectors such that the lattice vector is the shortest. It takes as input an enumeration radius $R > 0$ and a basis $\mathbf{B}$ of a lattice L, and outputs all non-zero vectors $\mathbf{s}$ in L such that $\|\mathbf{s}\| \leq R$ (if exists). The radius R is taken as an upper bound of $\lambda_1(L)$, like $1.05\mathrm{GH}(L)$, to find non-zero shortest lattice vectors. It goes through the enumeration tree formed by all vectors in the projected lattices $\pi_n(L)$, $\pi_{n-1}(L), \cdots, \pi_1(L) = L$ with norm at most R. More precisely, the enumeration tree is a tree of depth n, and for each $1 \leq k \leq n+1$, the nodes at depth $n+1-k$ are all the vectors in the projected lattice $\pi_k(L)$ with norm at most R. In particular, the root of the tree is the zero vector because $\pi_{n+1}(L) = \{\mathbf{0}\}$. The parent of a node $\mathbf{u} \in \pi_k(L)$ at depth $n+1-k$ is the node $\pi_{k+1}(\mathbf{u})$ at depth $n-k$. The child nodes are arranged in order of norms.

b) Sieve (Exponential-space algorithm): It has a better asymptotic runtime than enumeration, but it requires exponential space $2^{\Theta(n)}$. The first algorithm of this kind is the randomized sieve algorithm proposed by Ajtai, Kumar and

Sivakumar (AKS) [12]. It outputs a shortest lattice vector with overwhelming probability, and its asymptotic complexity is much better than deterministic enumeration algorithms with $2^{O(n^2)}$ time complexity. The idea is that given a lattice L of dimension n, consider a ball S centered at the origin and of radius r with $\lambda_1(L) \leq r \leq O(\lambda_1(L))$. Then $\#(L \cap S) = 2^{O(n)}$ according to the Gaussian Heuristic. If we could perform an exhaustive search for all vectors in $L \cap S$, we could find a shortest lattice vector within $2^{O(n)}$ polynomial-time operations. In contrast, the AKS algorithm performs a randomized sampling of $L \cap S$. If it was uniformly sampled over $L \cap S$, a short lattice vector would be included in N samples with probability close to 1 for $N \gg \#(L \cap S)$. It can be also shown that there exists a vector $\mathbf{w} \in L \cap S$ such that $\mathbf{w}$ and $\mathbf{w}+\mathbf{s}$ can be sampled with non-zero probability for some shortest lattice vector $\mathbf{s}$. Thus a shortest lattice vector is obtained by computing a shortest difference of any pairs of the N sampled vectors in $L \cap S$.

2) Approximate-SVP algorithms: These algorithms are much faster than exact algorithms, but they output short lattice vectors, not necessarily the shortest ones.

a) LLL and variants: The first efficient approximate-SVP algorithm is the celebrated algorithm by Lenstra, Lenstra and Lovász (LLL) [13]. It is known as the most famous algorithm of *lattice basis reduction*, which finds a basis with short and nearly-orthogonal basis vectors. For $\frac{1}{4} < \delta < 1$, a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ is called *δ-LLL-reduced* if it satisfies two conditions; (i) (Size-reduction condition) $|\mu_{i,j}| \leq \frac{1}{2}$ for all $j < i$. (ii) (Lovász' condition) $\delta \|\mathbf{b}^*_{k-1}\|^2 \leq \|\pi_{k-1}(\mathbf{b}_k)\|^2$ for all k. In LLL, adjacent vectors $\mathbf{b}_{k-1}, \mathbf{b}_k$ are swapped if they do not satisfy the Lovász' condition. The complexity of LLL is proven to be polynomial in n. It is applicable also to linearly dependent vectors to remove its linear dependency. Its straightforward generalization is LLL with deep insertions (DeepLLL), in which *non-adjacent* basis vectors can be changed. Specifically, a basis vector $\mathbf{b}_k$ is inserted between $\mathbf{b}_{i-1}$ and $\mathbf{b}_i$ as $(\ldots, \mathbf{b}_{i-1}, \mathbf{b}_k, \mathbf{b}_i, \ldots, \mathbf{b}_{k-1}, \mathbf{b}_{k+1}, \ldots)$, called a *deep insertion*, if the condition $\|\pi_i(\mathbf{b}_k)\|^2 < \delta \|\mathbf{b}^*_i\|^2$ is satisfied.

b) BKZ and variants: A basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ of a lattice L is called *Hermite-Korkine-Zolotarev (HKZ)-reduced* if it is size-reduced and it satisfies $\|\mathbf{b}^*_k\| = \lambda_1(\pi_k(L))$ for all k. For $i \leq j$, we denote by $\mathbf{B}_{[i,j]}$ the local projected block $(\pi_i(\mathbf{b}_i), \pi_i(\mathbf{b}_{i+1}), \ldots, \pi_i(\mathbf{b}_j))$, and by $L_{[i,j]}$ the lattice spanned by $\mathbf{B}_{[i,j]}$. The notion of Block-Korkine-Zolotarev (BKZ)-reduction is a local block version of HKZ-reduction [14]–[16]; For a blocksize $2 \leq \beta \leq n$, a basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$ of a lattice L is called *β-BKZ-reduced* if it is size-reduced and every local block $\mathbf{B}_{[j,j+\beta-1]}$ is HKZ-reduced for all $1 \leq j \leq n - \beta + 1$. The BKZ algorithm [16] finds a β-BKZ-reduced basis, and it calls LLL to reduce every local block before finding a shortest vector over the block lattice. It has been implemented in software libraries [17], [18]. As β increases, a shorter lattice vector can be found, but the running time is more costly. As an enhancement of BKZ, DeepBKZ was proposed in [19], in which DeepLLL is called as a subroutine alternative to LLL. DeepBKZ finds a short lattice vector by smaller blocksizes than BKZ in practice.

D. Related work for the SVP Challenge

According to information of the hall of fame of SVP Challenge, two strategies are currently known for finding solutions of the SVP Challenge in high dimensions $n \geq 131$; the General Sieve Kernel (G6K) [6] and the recent variant of random sampling reduction [20]. The former is used to solve the current highest dimension 157. However, the solution is not a shortest in the lattice since its approximation factor is reported as 1.04906. Moreover, we see from [6, Table 2] and [20, Table 5] that most of solutions found by [6], [20] have their approximation factor around 1.04 or 1.05, and neither strategy can solve exact-SVP.

In contrast, exact-SVP algorithms such as the enumeration (ENUM) and the sieve, find a shortest vector. The sieve is faster than the enumeration in both theory and practice for higher dimensions n, but its space-complexity is exponential in n, whereas that of the enumeration is polynomial in n. In fact, it was reported in [6, Table 2] that the sieve required 160 and 246 GB for sieving dimensions 124 and 127, respectively. For scalability, we consider the enumeration and its parallelization. Applications of high-performance computing to cryptanalysis for RSA and ECDSA are summarized [21]. The enumeration builds a tree with leaves corresponding to short lattice vectors, including shortest ones. The tree quickly becomes huge as the dimension increases, but substantial speed-ups can be obtained by pruning techniques [9]. A simple parallelization is the divide-and-conquer approach for a (pruned) enumeration tree [22]–[25]. Another technique for parallelization is *randomization* of an input lattice basis to exploit the power of massively parallel computing [24], [25]. Given an input basis, it generates a number of different bases of the same lattice by unimodular transformation, and such bases are distributed to processes. Every process runs lattice basis reduction and searches a (pruned) enumeration tree until some process finds a non-zero shortest vector. Recent work [25] presented a shared-memory parallelized system based on randomization and extreme pruning of [9], but it reports the runtime of solving exact-SVP for dimensions up to at most 100 over quad-socket Intel E7-4890 v2 CPUs (60 cores).

The MAP-SVP is an exact-SVP solver using pruning technique and randomization strategy. The MAP-SVP is based on the ENUM and DeepBKZ algorithms. ENUM is the exact-SVP algorithm, and we use DeepBKZ to reduce the computational cost of ENUM. Furthermore, using the UG framework introduced in the next section, we realize asynchronous large-scale parallelization with information sharing. The MAP-SVP found new solutions for the SVP Challenge by executing 100,032 cores at most. In addition, we report an exact-SVP runtime of up to 120 dimensions.

III. Ubiquity Generator (UG) framework: A software tool to parallelize the MAP-SVP

Ubiquity Generator (UG) framework [26] is a generic software framework to parallelize an existing state-of-the-art branch-and-bound (B&B) based solver, which is referred to as the *base solver*, from "outside." UG is composed of

a collection of base C++ classes, which define customizable interfaces to base solvers and translate solutions and subproblems into a solver independent form. Additionally, there is a base class that defines interfaces for different message-passing protocols corresponding to the parallelization library used. The instantiated parallel solver is denoted as ug [base solver name, parallelization library name]. UG has been developed primarily in concert with a state-of-the-art Mixed Integer Programming (MIP) solver called SCIP [27]. As such, the instantiations ug[SCIP,MPI], known as ParaSCIP , and ug[SCIP,C++11-thread/pthrads], known as FiberSCIP , which run on a distributed computing environment and shared memory computing environment, respectively, are the most mature. Importantly, the distributed memory and shared memory solvers execute the same algorithm in general for the instantiations. This fact greatly aids the development of a parallel solver on a distributed memory environment, since a specific base solver dependent debugging can be performed in a shared memory environment. Since current state-of-the-art MIP solvers are made up of many lines of code, for example, SCIP has over 800,000 lines of C code, the ability to develop a parallel instantiation of the base solver in a shared memory environment is valuable. The largest-scale computation conducted with ParaSCIP is up to 80,000 cores on TITAN at Oak Ridge National Laboratory, which solved 12 previously unsolved MIP instances from the MIPLIB [28] benchmark sets [29]. For more details regarding the instantiation of a parallel version of SCIP using UG , the reader is referred to Shinano et al. [29] and Shinano et al. [30].

SCIP is a plugin-based solver, and it is customizable to develop a B&B based solver to solve a specific combinatorial optimization problem. One of the successful results of using this development mechanism is the SCIP-Jack, a solver for the Steiner Tree Problems and its variants. ug [SCIP-Jack, MPI] solved three open instances from the SteinLib [31] benchmark set [32]. Computations using up to 43,000 cores have been executed to solve open SteinLib instances using ug[SCIP-Jack, MPI] [33].

UG has shown flexibility and scalability for solving optimization problems. The ability of UG motivated the development of a parallel solver using a non-B&B based tree search solver. An instantiated solver by UG has two types of processes (or threads in case of running on shared memory). One is `LC`, which is a controller for parallel tree search, and `Solver` which solves subproblems. The main feature of `LC` is dynamic load balancing. Additionally, a type of randomized parallelization mechanism, called *racing ramp-up*, is also provided by UG (see [30] for details). One fundamental issue regarding the use of UG with a non-B&B based tree search solver is that the `LC` assumes that the `Solvers` are B&B based solvers. When applying UG to parallelize a non-B&B based tree search solver, `LC`'s controller features must be customized to expect non-B&B based solvers. For this purpose, the `LC`'s class code is abstracted, and a new virtual function *parallelDispatch* has been added, and detailed in Section IV-B. This is an experimental implementation, and it is not currently publicly available. The following section shows how the SVP-algorithm is parallelized in UG .

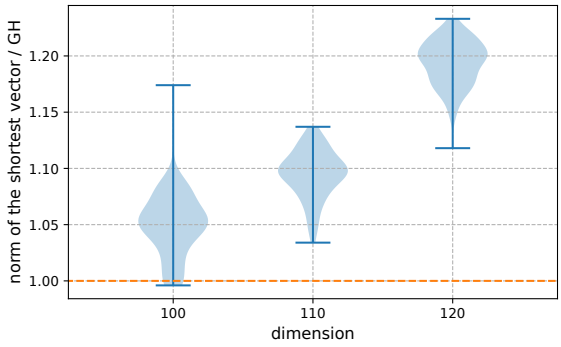

Fig. 2. A violin plot of the short vectors of the after-DeepBKZ lattice basis; GH is the value of the Gaussian Heuristic, which is the estimation of the shortest vector norm.

IV. Development of a massively parallelized system for solving exact-SVP

In this section, we show the architecture of the MAP-SVP for finding shortest lattice vectors. The MAP-SVP is based on randomized parallelization of ENUM and DeepBKZ algorithms, where DeepBKZ is an enhancement algorithm of BKZ. We describe the motivation to use the randomized parallelization in part A and the construction of the MAP-SVP after part B.

A. Randomized parallelization of independent serial DeepBKZ-ENUM algorithm

The randomized parallelization of the BKZ-ENUM algorithm was proposed in [24]. The method randomizes a lattice basis by unimodular matrix and runs the BKZ and ENUM algorithms independently in multiple processes. This parallelization method makes efficient use of parallel processes because each instance executes the algorithm separately. This method is based on the fact that there is an infinite number of basis forming the same lattice and that BKZ is a black-box algorithm. That is, the lattice basis after BKZ depends on the input basis, and we cannot predict the output basis. In Fig. 2, we show the variance of the lattice basis after DeepBKZ by randomizing the input basis. We generated 126 randomized lattice basis instances for each of the three SVP Challenge basis of dimensions $n = 100$, 110, and 120 (seed is 1). Then, we ran DeepBKZ with $\beta = 30$ and plotted the norm of the shortest vector of each instance. Fig. 2 shows that the lattice basis after DeepBKZ is different; especially in the 100-dimensional results, some instances give a vector whose norm is less than GH, where GH is the estimation of the shortest vector norm of the lattice. This result suggests that it is an effective strategy to execute DeepBKZ and ENUM in parallel from multiple randomized lattice basis.

Another advantage of independent parallelization is that the cost of ENUM can be reduced by *extreme pruning* of the enumeration tree. Extreme pruning is a technique of pruning

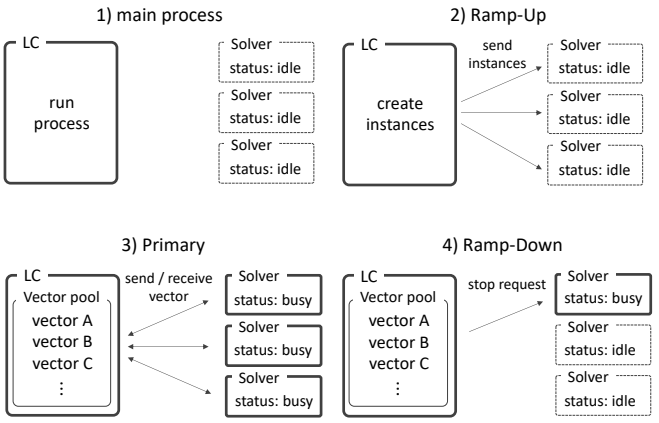

Fig. 3. Basic phases of the parallelDispatch

the enumeration tree so that the probability of the shortest vector lying within the pruned tree is theoretically p, and p is called the pruning parameter. Because each process has a randomized basis, we can assume that each ENUM succeeds in finding the shortest vector with independent probability. Therefore, we set the pruning parameter p to $1 - \sqrt[m]{1-P}$ so that the probability of success for the overall system is P, where m is the number of processes. It shows that by increasing the number of processes, we can set a smaller parameter p for each process while maintaining the overall success probability. This makes the pruning of each enumeration tree more aggressive and lowers the cost of each ENUM search.

B. ParallelDispatch: Sharing information among independent `Solvers` *in* UG

We applied UG (detailed in Section III) as a framework to achieve parallelization of ENUM and DeepBKZ. We describe the newly developed *parallelDispatch* function in UG . The UG has two types of processes: `LC` and multiple `Solvers`. In the parallelDispatch, `LC` sends instances to multiple `Solvers`, and while the `Solvers` process the instance, they share information asynchronously through `LC`. The parallelDispatch consists of four execution phases: *main process*, *Ramp-Up*, *Primary*, and *Ranp-Down*, as shown in Fig 3. With these four phases as one cycle, parallelDispatch executes the cycle multiple times.

1) Main process phase: Herein, only the `LC` runs the process, and all `Solvers` are idle. `LC` obtains the results of the parallel computation, prepares for the next parallel computation, and performs other operations.

2) Ramp-Up phase: This is the period from when all `Solvers` are idle until when all `Solvers` start processing after receiving the instance. `LC` creates an instance and sends it to the `Solvers` in turn. Therefore, some `Solvers` are delayed in receiving instances. We call the waiting time until these `Solvers` start processing *start idle time*.

3) Primary phase: All `Solvers` are processing the given instance. During this and the Ramp-Down phase, the `Solvers` send or receive vectors objects to and from the `LC` asynchronously. It allows all `Solvers` to share information through `LC`. `LC` has a priority queue called the *vector pool*, which stores vector objects. When a `Solver` sends the vector objects to the `LC`, the `LC` stores them in the vector pool if necessary. Conversely, if a `Solver` sends a receive request to the `LC`, the `LC` sends the appropriate vector objects from the vector pool to the `Solver`. Each `Solver` can send and receive the vector objects at its own convenient timing. The vector pool is managed with customized priorities, and when the pool is full, the vectors with lower priorities are removed. Because the `LC` receives send and receive requests from multiple `Solvers`, a time lag occurs during sending and receiving. We call it *wait idle time*.

4) Ramp-Down phase: This is the period when at least one `Solver` is in an idle state. Information is shared between busy `Solvers` through `LC`. The end time of a `Solver` depends on the instance given, or the vector received from the `LC`. If there are many `Solvers`, the end time variance generally becomes large. Therefore, `LC` has a function that can send a stop request to a `Solver`. When a `Solver` receives a stop request, it ends the process immediately.

5) Construction the MAP-SVP using parallelDispatch: The MAP-SVP uses the parallelDispatch function to parallelize DeepBKZ and ENUM as follows. The MAP-SVP reads the SVP instance, performs only one Ramp-Up along with the Primary and Ramp-Down, and outputs the result. In the Ramp-Up phase, the `LC` sends a lattice basis to each `Solver`, and the `Solvers` randomize it by a unimodular matrix. Then, each `Solver` executes DeepBKZ and ENUM for its own randomized lattice basis. As discussed in the following subsection, to realize the parallelization of ENUM and DeepBKZ with sharing information, it is necessary to share the vectors between `Solvers`. The vectors are stored in the vector pool, and the vector pool manages vectors by the length. Because the timing of each `Solver` finding short vectors is different, the asynchronous communication between the `Solvers` and the `LC` allows the `Solvers` to share information more efficiently. The MAP-SVP outputs the shortest vector from the vector pool at the end of the Ramp-Down phase.

C. Parallelized ENUM with information sharing

In this subsection, we propose an ENUM algorithm for information sharing parallelization. The ENUM is a brute force algorithm to enumerate non-zero vectors in a lattice whose norm is shorter than a given R. We can represent the search space of ENUM as a tree structure whose leaves correspond to lattice vectors, called an enumeration tree (detailed in Section II-A). By setting R to the norm of the shortest vector in the bases, we can obtain the shortest vector reliably by depth-first search of the enumeration tree. We assume a situation where ENUMs run independently in multiple processes for different lattice basis under the control of UG . When we find the vector $\mathbf{v}^*$ whose norm is shorter than R during the search enumeration tree, even if we update R to $\|\mathbf{v}^*\|$, the shortest vector remains in the enumeration tree. Therefore, we can reduce the search space without the miss to find the shortest

vector. We can also prune a self-enumerated tree by updating its search parameter R to $\|\mathbf{w}^*\|$ when the short vector $\mathbf{w}^*$ found in the other process is shorter than R. This also keeps the optimality of the shortest vector search for the overall system. That is, each process reduces the search space based on the vector found by not only the own process but also other process. In the MAP-SVP , each process sends a receive request to the `LC` at regular intervals to see if other processes have updated the short vector. Conversely, when the short vector is updated in their process, they send the vector to the `LC`. We can use the same search space reduction method in the *extreme pruning* technique with ENUM.

D. Parallelization of DeepBKZ with shared information

We also propose a DeepBKZ algorithm with information sharing as well as ENUM. DeepBKZ repeats generating a new vector from a subset of the basis and then inserts it into the basis, removing the linear dependence by the MLLL algorithm [34]. Besides, we reduce the basis by DeepLLL. In generating a new vector, we try to find the shortest orthogonal projection vector $\mathbf{v}^*$ on $L_{[k,k+\beta-1]}$ by ENUM to $L_{[k,k+\beta-1]}$, where $L_{[k,l]}$ is the lattice generated from $(\pi_k(\mathbf{b}_k), \ldots, \pi_k(\mathbf{b}_l))$. Generally, if the norm of the vector is short, the norm of the orthogonal projection is also short because we have $\|\mathbf{v}\| \geq \|\pi_k(\mathbf{v})\|$ for any vector $\mathbf{v}$ and k. Thus, a shorter vector $\mathbf{v}$ found in other processes is also likely to be short on $L_{[k,k+\beta-1]}$ with some k for its basis. Thus, every process running DeepBKZ also sends a receive-request to `LC` at certain intervals to obtain the short vectors found in other processes, which is inserted into its basis instead of performing a generation of new vector. We then remove the linear independence by MLLL. After this operation, we can continue the procedure of DeepBKZ. Also, when a process finds short vectors during three steps, the process sends them to the `LC`. This allows DeepBKZ to execute cooperatively in all processes.

We show a pseudo-code of the parallelization of DeepBKZ in Algorithm 1. DeepLLL$((\mathbf{b_1}, \ldots, \mathbf{b_m}), \delta)$ is the function which executes δ-DeepLLL algorithm for the basis $(\mathbf{b_1}, \ldots, \mathbf{b_m})$ and returns a δ-DeepLLL reduced basis. The same is true for MLLL.

E. Checkpoint and restart

We describe the checkpoint and restart mechanism in our parallel solver MAP-SVP . It is sufficient to store only the current lattice basis for the restart of the DeepBKZ. Besides, we can restart ENUM with only the search node at the end of the previous execution, since the depth-first-search of ENUM is executed in strict order according to a criteria. Specifically, only the coefficients of basis and lattice basis are needed. Also, each algorithm can restart the calculation with a little precalculation from the above information. Therefore, the MAP-SVP can perform checkpoints by storing a small amount of information whose data size is n^2+n for the n-dimensional SVP. Also, there is almost no additional cost required for restart. This is a great advantage of the MAP-SVP in the case we execute a large computer many times with limited time.

Algorithm 1: Parallelization of β-DeepBKZ

INPUT: $(\mathbf{b}_1, \ldots, \mathbf{b}_n)$ - lattice basis, β - block size, δ - reduction parameter.
OUTPUT: $(\mathbf{b}_1, \ldots, \mathbf{b}_n)$ - β-DeepBKZ-reduced lattice basis.

$(\mathbf{b}_1, \ldots, \mathbf{b}_n) \leftarrow \mathrm{DeepLLL}((\mathbf{b}_1, \ldots, \mathbf{b}_n), \delta)$;
$z = 1, k = 0, s = \|\mathbf{b}_1\|$;
while $z < n - 1$ **do**
  `/* main β-DeepBKZ loop */`
  $k \leftarrow k+1, l \leftarrow \min(k+\beta-1, n), h \leftarrow \min(l+1, n)$;
  $\mathbf{v} \leftarrow \mathrm{ENUM}\left(L_{[k,l]}\right)$
  if $\|\mathbf{b}_k^*\| > \|\pi_k(\mathbf{v})\|$ **then**
    $z \leftarrow 0$;
    $(\mathbf{b}_1, \ldots, \mathbf{b}_h) \leftarrow$
     $\mathrm{MLLL}((\mathbf{b}_1, \ldots \mathbf{b}_{k-1}, \mathbf{v}, \mathbf{b}_k, \ldots \mathbf{b}_h), \delta)$
  else
    $z \leftarrow z + 1$;
    $(\mathbf{b}_1, \ldots, \mathbf{b}_h) \leftarrow \mathrm{DeepLLL}((\mathbf{b}_1, \ldots, \mathbf{b}_h), \delta)$
  end
  if $k = n - 1$ **then**
    `/* obtain a short vector from LC and insert it into the self basis */`
    $\mathbf{v} \leftarrow$ receiveVectorFromLC();
    $(\mathbf{b}_1, \ldots, \mathbf{b}_n) \leftarrow \mathrm{MLLL}((\mathbf{v}, \mathbf{b}_1, \ldots, \mathbf{b}_n), \delta)$;
    `/* send short vectors to LC */`
    sendVectorsToLC$(\{\mathbf{b}_k; \|\mathbf{b}_k\| < s, 1 \leq \forall k \leq n\})$
    $s \leftarrow \|\mathbf{b}_1\|$
    $k \leftarrow 0$;
  end
end

F. System overview

We show an overview of MAP-SLAVE based on lattice basis reduction and enumeration in Fig. 4. Given a lattice basis $\mathbf{B}$ of a lattice L, a basic procedure of MAP-SLAVE for finding a shortest non-zero vector in L is as follows:

1) The `LC` distributes the input basis $\mathbf{B}$ to all the `Solvers`. Every `Solver` independently randomizes the input basis by multiplying with a random unimodular matrix. By means of this procedure, all the `Solvers` can have a different basis of the same lattice L.
2) Then, all the `Solvers` perform lattice basis reduction for their own basis to obtain a reduced basis of L. To obtain a more reduced basis, our system shares the shortest vectors among all the `Solvers`. Specifically, during lattice basis reduction, the `LC` collects short lattice vectors from all the `Solvers` and stores them in the vector pool. The `Solver` receives vectors from the vector pool, which is managed by the `LC`. Then, all the `Solvers` insert the vector as the first basis vector on their own basis. This procedure enables our system to share the shortest basis vector among all the `Solvers`.
3) After lattice basis reduction, all the `Solvers` perform

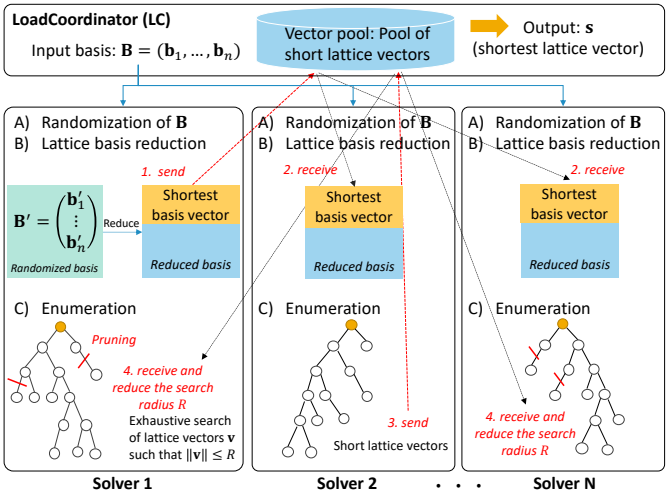

Fig. 4. An overview of our parallel SVP solving system

an exhaustive search of all short lattice vectors $\mathbf{v} \in L$ over their own reduced basis, satisfying $\|\mathbf{v}\| \leq R$ for a search radius R. In particular, we adopt an enumeration algorithm for our system to find a shortest lattice vector *deterministically*. (cf., the sieve is asymptotically faster than enumeration, but it is probabilistic, as described in Subsection II-C1). Like in performing lattice basis reduction, the `LC` stores a current shortest lattice vector in the pool during enumeration, and it shares the vector with all the `Solvers`. Thanks to such information, all the `Solvers` can prune their own enumeration trees to reduce their enumeration costs. After termination of enumeration in all the `Solvers`, we can acquire a shortest lattice vector from the pool of `LC`.

V. Numerical Experiments

In this section, we present several experimental results to demonstrate the effects of our parallelization for solving exact-SVP. We first note that we performed numerical experiments under the following conditions.

- Only MPI processes are used.
- We assign one process to one core strictly, and we do not use hyperthreading.
- `LC` and each `Solvers` are assigned to one process each.

In our MAP-SVP , most of the communications are performed between `LC` and `Solvers`, and there is little communication in a node. Therefore, only the MPI parallel is sufficient in the MAP-SVP . The computing platform used in the following numerical experiments includes the HLRN IV at Zuse Institute Berlin, the HPE SGI 8600 (ISM) at the Institute of Statistical Mathematics, and the ITO at the Kyushu University. The specifications of these are summarized in Table I. We conducted the following experiments with the size of the vector pool set to 1. That is, the incumbent vector is shared between all `Solvers`.

A. Effects of parallelization on lattice basis reduction

In computational lattice theory, the *Hermite factor* is known as a good index to measure the output quality of a practical reduction algorithm, such as LLL and BKZ (see [35] for their exhaustive experiments). It is defined by

$$\gamma = \frac{\|\mathbf{b}\|}{\mathrm{vol}(L)^{1/n}}, \tag{2}$$

where $\mathbf{b} \in L$ is the shortest *basis vector* output by a reduction algorithm for a basis of a lattice L of dimension n. In practice, the first basis vector $\mathbf{b}_1$ is the shortest among a reduced basis $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$. Shorter γ means that shorter basis vectors can be found by the reduction algorithm. That is, it finds a more reduced basis $\mathbf{B}$ over which the cost of enumeration becomes lower for finding shortest vectors $\mathbf{s} \in L$. For a practical reduction algorithm, its root Hermite factor $\gamma^{1/n}$ converges to a constant in practice for high dimensions $n \geq 100$ (e.g., it is experimentally shown in [35, Table 1] that $\gamma^{1/n} \approx 1.0219$ for LLL and 1.0128 for BKZ with blocksize $\beta = 20$).

In Figs. 5, 6, and 7, we show the transition of root Hermite factors $\gamma^{1/n}$ during the running of our parallel DeepBKZ in the MAP-SVP with blocksize $\beta = 30$ for random lattice bases of dimensions $n = 100$, 110 and 120. More precisely, we plot the average of root Hermite factors for random bases from the SVP Challenge [3] with seeds 0 to 9 in every dimension. These were calculated using machine CAL B and CAL C. In addition, we also plot the results for the 120 dimensions in larger-scale experiments using the ITO and HLRN IV. We note that we only calculate a base whose seed is 0 for the $100,032$ parallel experiment on HLRN IV. In our experiments, we used as input a reduced basis by BKZ with blocksize $\beta = 20$, and hence the initial root Hermite factor is approximately 1.0128 in every dimension. It can be seen from the figures that as the total number of processes p increases, the root Hermite factor becomes smaller. In particular, compared with results without parallelization (i.e., the case $p = 1$), root Hermite factors for $p = 128$ are much smaller. This is due to sharing with shortest basis vectors among all the `Solvers` in the MAP-SVP of lattice basis reduction (see Fig. 4 for an overview of our system). More precisely, we see that our parallel DeepBKZ with $p = 128$ can achieve Hermite factors

$$\gamma \approx \ 1.0092^n, \ 1.0089^n \text{ and } 1.0092^n \tag{3}$$

in average for $n = 100$, 110, and 120, respectively. This means that our parallel DeepBKZ can find a lattice vector $\mathbf{b} \in L$ with length $\|\mathbf{b}\| = \gamma \cdot \mathrm{vol}(L)^{1/n}$ from Equation (2). Moreover, the gap with the first minimum $\lambda_1(L) \approx \mathrm{GH}(L)$ is calculated as

$$\frac{\|\mathbf{b}\|}{\mathrm{GH}(L)} = \frac{\gamma}{\sqrt{n/2\pi e}} \approx \ 1.0326, \ 1.0443 \text{ and } 1.1321 \tag{4}$$

for $n = 100$, 110, and 120, respectively. This means that the parallel DeepBKZ in the MAP-SVP can solve approximate-SVP with factors from (4) for the corresponding dimensions. In particular, our parallel DeepBKZ with blocksize $\beta = 30$ is sufficient to enter the hall of fame of the SVP Challenge for

TABLE I
COMPUTING PLATFORMS USED

Machine	Memory / node	CPU	CPU frequency	# of nodes	# of cores
HLRN IV	384 GB	Intel Xeon Platinum 9242 (CLX-AP)	2.30 GHz	1,042	100,032 (96 × 1,042)
ISM	384 GB	Intel Xeon Gold 6154	3.00 GHz	144	5,184 (36 × 144)
ITO	192 GB	Intel Xeon Gold 6154 (Skylake-SP)	3.00 GHz	2000	72,000 (36 × 2000)
CAL A	32 GB	Intel(R) Xeon(R) CPU E3-1284L v3	1.80 GHz	45	180 (4 × 45)
CAL B	256 GB	Intel(R) Xeon(R) CPU E5-2640 v3	2.60 GHz	4	64 (16 × 4)
CAL C	256 GB	Intel(R) Xeon(R) CPU E5-2650 v3	2.30 GHz	4	80 (20 × 4)

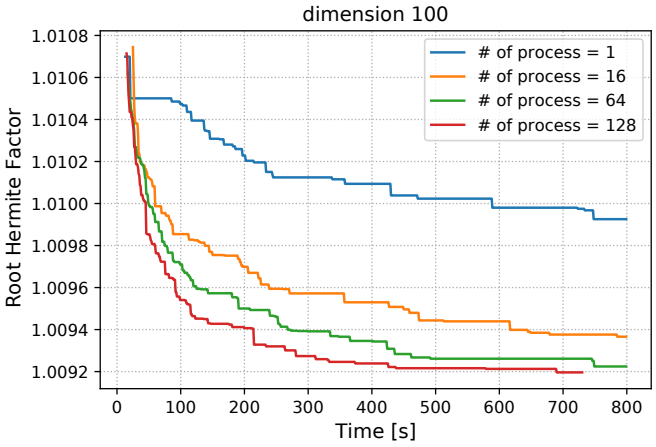

Fig. 5. Transition of average root Hermite factors $\gamma^{1/n}$ by our parallelized lattice reduction using DeepBKZ [19] with blocksize $\beta = 30$ for random lattice bases of dimension $n = 100$.

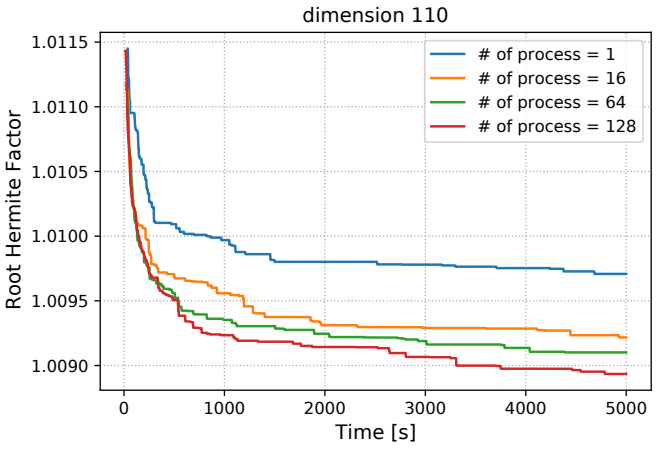

Fig. 6. Same as Fig. 5, but dimension $n = 110$

dimension $n = 110$ because its approximation factors are less than 1.05 (although such lattice vectors are not necessarily the shortest). The 120 dimensional experiments show that the $\gamma^{1/n}$ reduces quickly with the increase of the number of processes, even on the larger-scale parallelization. It indicates that the scalability of our parallel DeepBKZ keeps even in the larger-scale, and means that we can stop the reduction algorithm faster as the number of processes increases.

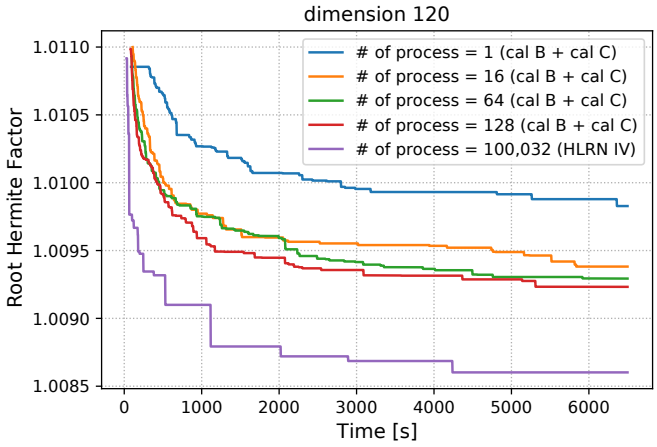

Fig. 7. Same as Fig. 5, but dimension $n = 120$

a) Comparison with BKZ: The BKZ algorithm and its variants such as BKZ 2.0 [36], are the de facto standard in lattice basis reduction for cryptanalysis of lattice-based cryptography (see [37] for cryptanalysis). For example, they were used in [6], [24], [25] to process exact-SVP algorithms (such as enumeration and sieve) to reduce their cost. Under the Gaussian Heuristic and certain assumptions, a limiting value of the root Hermite factor of BKZ (or BKZ 2.0) with blocksize β is predicted in [38] as

$$\lim_{n\to\infty} \gamma^{\frac{1}{n}} = \left(\nu_\beta^{-\frac{1}{\beta}}\right)^{\frac{1}{\beta-1}} \sim \left(\frac{\beta}{2\pi e}(\pi\beta)^{\frac{1}{\beta}}\right)^{\frac{1}{2(\beta-1)}}. \quad (5)$$

There is experimental evidences to support this prediction for high blocksizes such as $\beta > 50$ and $\beta \ll n$ (see [36], [38], [39]). (note that the Gaussian Heuristic holds in practice for random lattices in high dimensions, but unfortunately it is violated in low dimensions). The prediction (5) implies that approximately $\beta = 106$ is required for BKZ to achieve the Hermite factor $\gamma \approx 1.009^n$. On the other hand, only $\beta = 30$ is sufficient to achieve Hermite factors (3) in our parallelized DeepBKZ. This is due to the use of DeepBKZ [19] and sharing of the first basis vector among all the solvers. The most dominant part of BKZ and DeepBKZ is the subroutine of calling an exact-SVP oracle over projected lattices of dimension β, and its cost is $O(2^{\beta^2})$ in using the enumeration as an exact-SVP oracle. This argument shows that our parallelized DeepBKZ is much faster than BKZ without parallelization.

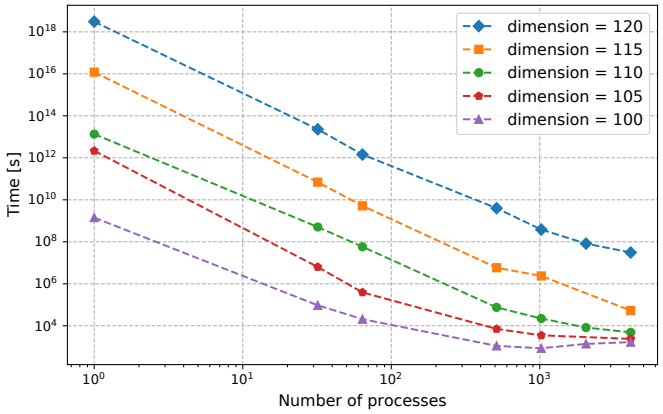

Fig. 8. The average solving times of exact-SVP for each dimension 100, 105, 110, 115, and 120.

B. Total cost of solving exact-SVP with the MAP-SVP

We evaluate the performance of the MAP-SVP . Each Solver performs DeepBKZ and ENUM for the instance given by the LC. To reduce the calculation time, each solver applies the *extreme pruning* technique to ENUM so that the probability of finding the shortest vector search in the overall system is $P = 0.95$. We define the time at which all solvers finish the calculation as *total cost*. The calculation time of a solver consists of the time of running DeepBKZ and ENUM. If the solver process has not been completed, add the remaining estimated ENUM cost to the calculation time. The ENUM Cost is calculated based on the program codes in [40] that implement the formula described in [41]. Fig. 8 shows how the estimated average total cost of the MAP-SVP changes with the number of processes and the dimension of the SVP instance. Specifically, we plot the average total cost of the SVP Challenge with seeds 0, 1, and 2 in every pair of dimension and number of processes.

We can find the proportional relationship between the log order of the total cost and the log order of the number of processes from Fig. 8. The slope of each line shows the scalability of our parallel system. For dimension 100 results, the total cost stops decreasing for the number of processes above $1,000$, which indicates that the number of processes is too large for the size of this instance. This suggests that there is a number of processes for which the total cost reduction effect stagnates and that the number also increases as the dimension increases. Although the predicted value of it is not known from this experiment, it is expected from the experimental results that it is $4,000$ or more for at least 120 dimensions or more.

C. Parallelization performance of UG

We now show the parallel efficiency of the implementation using UG . We assign a `Solver` to each process, and each `Solver` executes DeepBKZ and ENUM for the given random instance. The short vectors are shared among the `Solvers` through the vector pool in `LC` during the execution of the algorithm. Fig. 9 shows the results of the experiment for the

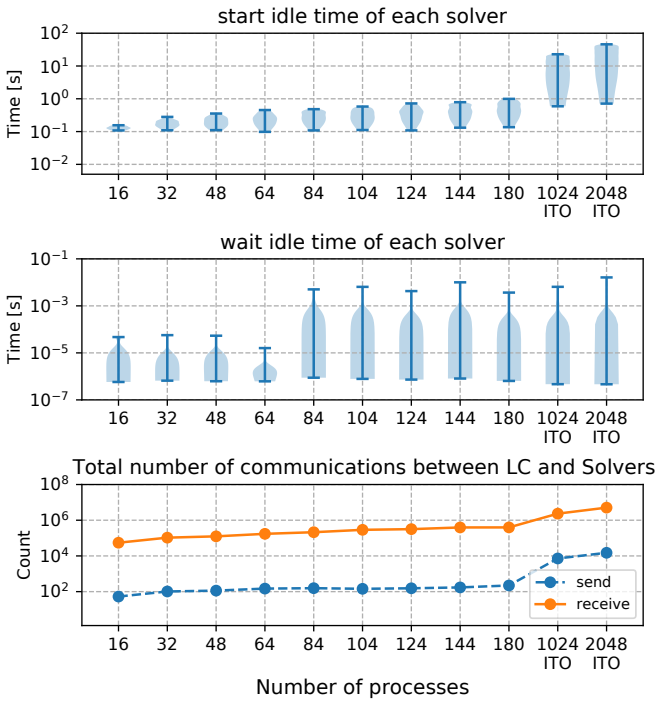

Fig. 9. Idle times and the total number of send and receive request.

100-dimension SVP instance, which are the idle time until each solver receives instance from `LC`, the total time lags that occur while sending and receiving information, and the number of sending and receiving requests of each solver (the definition of start and wait idle time are described in IV-B). We execute the MAP-SVP from 16 to 180 processes at CAL A and more than $1,024$ processes at the ITO supercomputer. Because the `Solver` receives short vectors of `LC` at a given interval in the MAP-SVP , the number of receive requests is constant for each process regardless of the number of processes. Also, the `Solver` sends a vector to the `LC` when its shortest vector is updated. However, the number of sent vectors is generally much less than the number of receive requests. Therefore, as the number of processes increases, the `LC` load increases linearly with increasing receive requests. Although the load on the `LC` increases, idle time is much less than the total running time per hour (see the second part of Fig. 9). This result shows that the MAP-SVP can communicate with low overhead.

D. New solutions for the SVP Challenge

In Table II, we list new solutions for the hall of fame in the Darmstadt SVP Challenge [3], found by the MAP-SVP . For the SVP Challenge, we had run the MAP-SVP for each target dimension with several seeds. As shown in Table II, we had succeeded in finding shorter lattice vectors in dimensions $n = 104$, 111, 121, and 127; note that a shorter lattice vector than a previous one can be entered in the hall of fame in the same dimension with a *different seed*. In particular, we expect that all the norms of our lattice vectors in Table II are shortest in each lattice due to our parameter setting of

pruned enumeration. In fact, all the approximation factors of our lattice vectors are close to or less than 1 (the approximation factor is defined as $\|\mathbf{v}\|/\mathrm{GH}(L)$ for a non-zero vector $\mathbf{v}$ in a lattice L, where $\mathrm{GH}(L)$ is an estimation of the shortest length $\lambda_1(L)$ in the lattice).

a) New solution for the 127-dimensional SVP Challenge: We tested solving an SVP Challenge instance of the 127-dimensions. It is hard to solve 127-dimensional instances accurately; for example, the G6K [6] solver described later took about 14 days to find a sufficiently short vector for the 127-dimensional lattice, according to [6, Table 2]. Therefore, we experimented only with the basis of seed at 0, 1, and 3.

- For the instance of seed 3, MAP-SVP found the short vector that updates the 127-dimensional SVP Challenge record with seven executions. These executions are summarised in Table III. After the second try, the executions were restarted from the lattice base stored as a checkpoint of the previous execution. The approximation factor 0.9757 of this record is the lowest value any record with more than 127 dimensions. This is suggested that MAP-SVP can solve SVPs with high accuracy.
- For the instance of seed 0, we ran the MAP-SVP thrice: 12 hours with $24,576$ processes on HLRN IV, 12 hours with $49,152$ processes on HLRN IV, and an hour with $5,184$ processes on ISM with checkpoints and restarts. Thus, in only approximately a day, the MAP-SVP found a vector of almost the same length as listed in the record of the SVP Challenge (the length of the record is 2897.58, and that found by the MAP-SVP is 2897.83). The vector found by the MAP-SVP is presented in the Artifact Description Appendix.
- For the instance of seed 1, we found the vector whose length was the same as that of the 127-dimensional top record of the SVP Challenge with following five executions involving checkpoints and restarts: four executions of 6 hours with $4,608$ processes and one of 6.7 hours with $9,980$ processes on ITO.

The MAP-SVP succeeded in finding a sufficiently short lattice vector in a sufficiently short time for any three instances.

E. Comparison with G6K

The current highest dimension solved in the SVP Challenge is 157, but the vector is somewhat far from the shortest one in the lattice since its approximation factor is 1.04906. Specifically, almost all lattice vectors in the hall of fame of the SVP Challenge for dimensions $n > 130$ were found by using the General Sieve Kernel (G6K) [6] with the sub-sieve strategy. The strategy was proposed in [42], and its basic idea is to perform the sieve in a projected lattice $\pi_\ell(L)$ for generating short *projected* lattice vectors and then to lift them into vectors in the whole lattice L by the simple operations of linear algebra. Such lattice vectors are short enough to be entered into the hall of fame of the SVP Challenge, but they are not at all guaranteed as the shortest in the lattice. In fact, as reported in [6, Table 2], almost all records by G6K have approximation factors close to 1.05. If we compare with

TABLE II
NEW SOLUTIONS FOR THE HALL OF FAME IN THE SVP CHALLENGE [3], FOUND BY MAP-SVP

Dim.	Seed	Norm	App. factor	#Process	Total time
104	35	2516	0.97173	120	551 seconds
	85	2520	0.97010	120	214 seconds
	82	2529	0.97719	120	432 seconds
111	29	2597	0.96979	2000	792 seconds
	30	2635	0.98382	2000	541 seconds
	8	2660	0.99467	2000	611 seconds
121	4	2780	0.99706	2304	682 minutes
	2	2809	1.00820	2304	481 minutes
127*	3*	2790	0.97573	91,200	147 hours
	1†	2890	1.01429	9,980	31 hours
	0†	2898	1.01626	49,152	25 hours

*† We executed the MAP-SVP several times on multiple computers, as described in paragraph V-D0a. We list the maximum number of processes and total approximate wall time among these executions in the table.
† These solutions are not new records, but they are the same solution as the previous record or very nearly close to it.

TABLE III
EXECUTION HISTORY FOR THE SVP CHALLENGE OF 127-DIMENSIONAL LATTICE WITH SEED = 3

try	Norm	App. factor	#Process	Wall Time	Machine
1	3186	1.11435	4,608	6 hours	ITO
2	3186	1.11435	180	11 hours	CAL B, C
3	3037	1.06218	4,608	6 hours	ITO
4	2956	1.03397	4,608	6 hours	ITO
5	2956	1.03397	49,152	12 hours	HLRN IV
6	2922	1.02202	5,184	100 hours	ISM
7	2790	0.97573	91,200	6.3 hours	HLRN IV
Total	2790	0.97573		$\approx$ 147 hours	

G6K [6], it seems reasonable for us to look at the dimension $m = n - \ell + 1$ of the projected lattice $\pi_\ell(L)$ since the sieve over the projected lattice is dominant in the sub-sieve strategy. For example, it was reported in [6, Table 2] that for the SVP Challenge with dimensions $n = 151$, 153, and 155, it makes use of maximal sieve dimensions $m = 124$, 123, and 127, respectively. It took about 11.6 (resp., 11.8 and 14.7) days of wall time for the maximal sieve dimension $m = 123$ (resp., 124 and 127) to find lattice vectors short enough for the hall of fame over 2×Intel Xeon Gold 6138 at 2.0 GHz with 40 cores and 80 threads (resp., 2×Intel Xeon E5-2650v3 at 2.3 GHz with 20 cores and 20 threads, and 4×Intel Xeon E7-8860v4 at 2.2 GHz with 72 cores and 72 threads). In contrast, Table II shows that our system took around 10 hours over 2,304 processors to find the shortest vector for dimension 121. Hence our system seems faster than G6K for dimensions around 120 (probably around 130) if we ignore computational resources. Moreover, while G6K requires exponential-space in dimension, our system requires polynomial-space, due to the difference of exact-SVP algorithms (i.e., sieve vs. enumeration in Subsection II-C1). While it was reported in [6, Table 2] that the memory usage of G6K increases exponentially for the 93–127 dimensions, the memory usage of MAP-SVP during three hours executions were about the same at 123–155 dimensions as about 0.01 GB (see Fig. 10). In addition, due to the

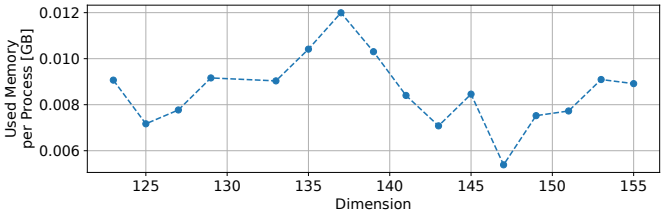

Fig. 10. The memory usage in the MAP-SVP for each dimension of SVP

algorithm of the MAP-SVP , the memory required by each process changes little during execution. This suggests that the MAP-SVP can be applied to large-scale computers with a small memory footprint without limiting the memory in the process.

VI. Conclusion and Future work

Based on the specialized UG with parallelDispatch, we proposed a new SVP solver called MAP-SVP and demonstrated its performance. The MAP-SVP is the first distributed asynchronous cooperative system for SVP solving applications. The features of the MAP-SVP are a low communication overhead, a small memory footprint, and the ease of checkpoint and restart. In addition, we demonstrated the scalability of a number of processes through numerical experiments. Therefore, the MAP-SVP can improve the performance by increasing the parallelization scale when the dimension of the SVP instance is sufficiently large.

Further, we have planned to execute a more large-scale experiment of solving SVP using several millions of processes in the Fugaku supercomputer at RIKEN. We aim to stably execute the MAP-SVP even in this massively large-scale and to solve 130–140 dimensions SVP. Besides, we are developing a *subENUM* algorithm as an improvement of the ENUM algorithm. This algorithm solves SVP of a low-dimensional projected lattice, similar to the sub-sieving strategy used in G6K . By incorporating this algorithm into the MAP-SVP , we can solve a higher dimension of SVP. With this algorithmic improvements, we also aim to break the record of more than 150 dimensions of the SVP Challenge.

Acknowledgment

This research project was supported by the Japan Science and Technology Agency (JST), the Core Research of Evolutionary Science and Technology (CREST), the Center of Innovation Science and Technology based Radical Innovation and Entrepreneurship Program (COI Program), JSPS KAKENHI Grant No. JP 16H01707 and 20H04142, Japan, "Advanced Computational Scientific Program" of Research Institute for Information Technology, Kyushu University, the Research Campus Modal funded by the German Federal Ministry of Education and Research (fund number 05M20ZBM), and the North-German Supercomputing Alliance (HLRN). We are grateful to the HLRN IV supercomputer staff, especially Matthias Läuter. We would also like to thank Yoshinori Aono for several programs and his helpful advices.

References

[1] The National Institute of Standards and Technology (NIST), "Post-quantum cryptography." [Online]. Available: https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization

[2] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Symposium on Foundations of Computer Science (FOCS 1994)*. IEEE, 1994, pp. 124–134.

[3] M. Schneider, N. Gama, P. Baumann, and L. Nobach, "SVP challenge (2010)," *URL: http://latticechallenge.org/svp-challenge*.

[4] P. Q. Nguyen, "Hermite's constant and lattice algorithms," in *The LLL Algorithm*. Springer, 2009, pp. 19–69.

[5] D. Micciancio and S. Goldwasser, *Complexity of lattice problems: A cryptographic perspective*. Springer Science & Business Media, 2012, vol. 671.

[6] M. Albrecht, L. Ducas, G. Herold, E. Kirshanova, E. W. Postlethwaite, and M. Stevens, "The general sieve kernel and new records in lattice reduction," in *Advances in Cryptology–EUROCRYPT 2019*, ser. Lecture Notes in Computer Science, vol. 11477. Springer, 2019, pp. 717–746.

[7] G. Falcao, F. Cabeleira, A. Mariano, and L. Paulo Santos, "Heterogeneous implementation of a voronoi cell-based svp solver," *IEEE Access*, vol. 7, pp. 127 012–127 023, 2019.

[8] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, T. Koch, and M. Winkler, "Solving open mip instances with parascip on supercomputers using up to 80,000 cores," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 770–779.

[9] N. Gama, P. Q. Nguyen, and O. Regev, "Lattice enumeration using extreme pruning," in *Advances in Cryptology–EUROCRYPT 2010*, ser. Lecture Notes in Computer Science, vol. 6110. Springer, 2010, pp. 257–278.

[10] M. Ajtai, "Generating hard instances of lattice problems," in *Symposium on Theory of Computing (STOC 1996)*. ACM, 1996, pp. 99–108.

[11] D. Goldstein and A. Mayer, "On the equidistribution of Hecke points," in *Forum Mathematicum*, vol. 15, no. 2. De Gruyter, 2003, pp. 165–190.

[12] M. Ajtai, R. Kumar, and D. Sivakumar, "A sieve algorithm for the shortest lattice vector problem," in *Symposium on Theory of Computing (STOC 2001)*. ACM, 2001, pp. 601–610.

[13] A. K. Lenstra, H. W. Lenstra, and L. Lovász, "Factoring polynomials with rational coefficients," *Mathematische Annalen*, vol. 261, no. 4, pp. 515–534, 1982.

[14] C.-P. Schnorr, "A hierarchy of polynomial time lattice basis reduction algorithms," *Theoretical computer science*, vol. 53, no. 2-3, pp. 201–224, 1987.

[15] ——, *Block Korkin-Zolotarev bases and successive minima*. International Computer Science Institute, 1992.

[16] C.-P. Schnorr and M. Euchner, "Lattice basis reduction: Improved practical algorithms and solving subset sum problems," *Mathematical programming*, vol. 66, pp. 181–199, 1994.

[17] V. Shoup, "NTL: A Library for doing Number Theory," available at http://www.shoup.net/ntl/. [Online]. Available: http://www.shoup.net/ntl/

[18] The FPLLL development team, "fplll, a lattice reduction library," 2016. [Online]. Available: https://github.com/fplll/fplll

[19] J. Yamaguchi and M. Yasuda, "Explicit formula for Gram-Schmidt vectors in LLL with deep insertions and its applications," in *Number-Theoretic Methods in Cryptology (NuTMiC 2017)*, ser. Lecture Notes in Computer Science, vol. 10737. Springer, 2017, pp. 142–160.

[20] T. Teruya, K. Kashiwabara, and G. Hanaoka, "Fast lattice basis reduction suitable for massive parallelization and its application to the shortest vector problem," in *Public Key Cryptography (PKC 2018)*, ser. Lecture Notes in Computer Science, vol. 10769. Springer, 2018, pp. 437–460.

[21] A. Joux, "A tutorial on high performance computing applied to cryptanalysis (invited talk)," in *Advances in Cryptology–EUROCRYPT 2012*, ser. Lecture Notes in Computer Science, vol. 7237. Springer, 2012, pp. 1–7.

[22] Ö. Dagdelen and M. Schneider, "Parallel enumeration of shortest lattice vectors," in *Euro-Par 2010–Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 6272. Springer, 2010, pp. 211–222.

[23] J. Hermans, M. Schneider, J. Buchmann, F. Vercauteren, and B. Preneel, "Parallel shortest lattice vector enumeration on graphics cards," in *Progress in Cryptology–AFRICACRYPT 2010*, ser. Lecture Notes in Computer Science, vol. 6055. Springer, 2010, pp. 52–68.

[24] P.-C. Kuo, M. Schneider, Ö. Dagdelen, J. Reichelt, J. Buchmann, C.-M. Cheng, and B.-Y. Yang, "Extreme enumeration on GPU and in clouds," in *Cryptographic Hardware and Embedded Systems–CHES 2011*, ser. Lecture Notes in Computer Science, vol. 6917. Springer, 2011, pp. 176–191.

[25] M. Burger, C. Bischof, and J. Krämer, "p3Enum: A new parameterizable and shared-memory parallelized shortest vector problem solver," in *Computational Science–ICCS 2019*, ser. Lecture Notes in Computer Science, vol. 11540. Springer, 2019, pp. 535–542.

[26] "UG: Ubiquity Generator framework," http://ug.zib.de/.

[27] "SCIP: Solving Constraint Integer Programs," http://scip.zib.de/.

[28] "MIPLIB 2017," 2018, http://miplib.zib.de.

[29] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, T. Koch, and M. Winkler, "Solving open MIP instances with ParaSCIP on supercomputers using up to 80,000 cores," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2016, pp. 770–779.

[30] Y. Shinano, S. Heinz, S. Vigerske, and M. Winkler, "Fiberscip—a shared memory parallelization of scip," *INFORMS Journal on Computing*, vol. 30, no. 1, pp. 11–30, 2018. [Online]. Available: https://doi.org/10.1287/ijoc.2017.0762

[31] T. Koch, A. Martin, and S. Voß, "SteinLib: An updated library on Steiner tree problems in graphs," in *Steiner Trees in Industries*, D.-Z. Du and X. Cheng, Eds. Kluwer, 2001, pp. 285–325.

[32] G. Gamrath, T. Koch, S. Maher, D. Rehfeldt, and Y. Shinano, "SCIP-Jack—a solver for STP and variants with parallelization extensions," *Mathematical Programming Computation*, vol. 9, no. 2, pp. 231–296, 2017.

[33] Y. Shinano, D. Rehfeldt, and T. Koch, "Building optimal steiner trees on supercomputers by using up to 43,000 cores," in *Integration of Constraint Programming, Artificial Intelligence, and Operations Research. CPAIOR 2019*, vol. 11494, 2019, pp. 529–539.

[34] C. C. Sims, *Computation with finitely presented groups*. Cambridge University Press, 1994, vol. 48.

[35] N. Gama and P. Q. Nguyen, "Predicting lattice reduction," in *Advances in Cryptology–EUROCRYPT 2008*, ser. Lecture Notes in Computer Science, vol. 4965. Springer, 2008, pp. 31–51.

[36] Y. Chen and P. Q. Nguyen, "BKZ 2.0: Better lattice security estimates," in *Advances in Cryptology–ASIACRYPT 2011*, ser. Lecture Notes in Computer Science, vol. 7073. Springer, 2011, pp. 1–20.

[37] M. R. Albrecht, B. R. Curtis, A. Deo, A. Davidson, R. Player, E. W. Postlethwaite, F. Virdia, and T. Wunderer, "Estimate all the {LWE, NTRU} schemes!" in *Security and Cryptography for Networks (SCN 2018)*, ser. Lecture Notes in Computer Science, vol. 11035, 2018, pp. 351–367.

[38] Y. Chen, "Réduction de réseau et sécurité concrete du chiffrement completement homomorphe," Ph.D. dissertation, Paris 7, 2013.

[39] Y. Yu and L. Ducas, "Second order statistical behavior of LLL and BKZ," in *Selected Areas in Cryptography (SAC 2017)*, ser. Lecture Notes in Computer Science, vol. 10719. Springer, 2017, pp. 3–22.

[40] Y. Aono, Y. Wang, T. Hayashi, and T. Takagi, "Progressive bkz library," 2018.

[41] ——, "Improved progressive BKZ algorithms and their precise cost estimation by sharp simulator," in *Advances in Cryptology–EUROCRYPT 2016*, ser. Lecture Notes in Computer Science, vol. 9665. Springer, 2016, pp. 789–819, progressive BKZ library is available from https://www2.nict.go.jp/security/pbkzcode/.

[42] L. Ducas, "Shortest vector from lattice sieving: A few dimensions for free," in *Adavances in Cryptology–EUROCRYPT 2018*, ser. Lecture Notes in Computer Science, vol. 10820. Springer, 2018, pp. 125–145.