Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

NARIAKI TATEIWA, YUJI SHINANO, KEIICHIRO YAMAMURA, AKIHIRO YOSHIDA, SHIZUO KAJI, MASAYA YASUDA, KATSUKI FUJISAWA

CMAP-LAP: Configurable Massively Parallel Solver for Lattice Problems

*The work for this article has been partially conducted within the Research Campus MODAL funded by the German Federal Ministry of Education and Research (fund number 05M14ZAM)

ZIB-Report 21-16 (July 2021)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Takustraße 7
D-14195 Berlin-Dahlem

Telefon: 030-84185-0
Telefax: 030-84185-125

e-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

CMAP-LAP: Configurable Massively Parallel Solver for Lattice Problems

Nariaki Tateiwa[*], Yuji Shinano[†], Keiichiro Yamamura[‡], Akihiro Yoshida[§], Shizuo Kaji[¶], Masaya Yasuda[‖], Katsuki Fujisawa[**]

July 8, 2021

Abstract

Lattice problems are a class of optimization problems that are notably hard. There are no classical or quantum algorithms known to solve these problems efficiently. Their hardness has made lattices a major cryptographic primitive for post-quantum cryptography. Several different approaches have been used for lattice problems with different computational profiles; some suffer from super-exponential time, and others require exponential space. This motivated us to develop a novel lattice problem solver, CMAP-LAP, based on the clever coordination of different algorithms that run massively in parallel. With our flexible framework, heterogeneous modules run asynchronously in parallel on a large-scale distributed system while exchanging information, which drastically boosts the overall performance. We also implement full checkpoint-and-restart functionality, which is vital to high-dimensional lattice problems. Through numerical experiments with up to 103,680 cores, we evaluated the performance and stability of our system and demonstrated its high capability for future massive-scale experiments.

1 Introduction

A *lattice* is the set of all integral combinations of n linearly independent vectors in the Euclidean space $\mathbb{R}^n$. *Lattice problems* are a class of discrete optimization problems whose objective functions are defined on the set of lattice points or the set of lattice bases. The

[*]Graduate School of Mathematics, Kyushu University, Fukuoka, 819-0395, Japan
[†]Department of Applied Algorithmic Intelligence Methods (A^{2}IM), Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
[‡]Graduate School of Mathematics, Kyushu University, Fukuoka, 819-0395, Japan
[§]Graduate School of Mathematics, Kyushu University, Fukuoka, 819-0395, Japan
[¶]Institute of Mathematics for Industry, Kyushu University, 819-0395, Japan
[‖]Department of Mathematics, Rikkyo University, Tokyo, 171-8501, Japan
[**]Institute of Mathematics for Industry, Kyushu University, 819-0395, Japan

most fundamental instance of the lattice problems is the *Shortest Vector Problem (SVP)*, which asks to find the shortest non-zero vector in a given lattice. Lattice problems are believed to be computationally hard with both classical and quantum algorithms [5] and have been used to construct various cryptosystems [22], including post-quantum cryptography. Therefore, developing a framework for lattice problems is an important task both in large-scale optimization and cryptanalysis (see [15] for cryptanalysis using high-performance computing). More specifically, the security of many cryptosystems is based on the hardness of an approximate variant of SVP. Lattice problem solvers have been extensively tested at the Darmstadt SVP challenge [25], which asks to find a lattice vector shorter than 1.05 times the expected length of a non-zero shortest lattice vector (see [6] for a choice of the approximate factor 1.05).

There are three basic families of lattice algorithms that have been developed to solve practical lattice problems: basis reduction, enumeration (ENUM), and sieve. We provide a brief description of their variants in Section 2. These algorithms have advantages and disadvantages, and there is no single definite algorithm for lattice problems. Therefore, practical lattice-problem solvers generally rely on two or more algorithms. G6K [2] implements a variety of basis reduction and sieve algorithms, and it is considered the state-of-the-art SVP solver. G6K is equipped with both CPU and GPU highly parallelized implementations, but it runs only on a single machine. Furthermore, the memory requirement is exponential with respect to the dimension of the lattice, which is inevitable for sieve algorithms. On the other hand, MAP-SVP [35] is based on basis reduction and ENUM, which showed efficient scalability above $100,000$ MPI processes.

Existing solvers are limited to a fixed set of algorithms and lack in flexibility. There are two main obstacles in developing a large-scale multi-paradigm solver: the need for an efficient high-level information-sharing scheme across different algorithms, and an adaptive task selection and distribution strategy for hundreds of thousands of processes. The main contribution of this paper is to provide solutions to overcome these obstacles and develop a flexible framework to make various algorithms work cooperatively on a large-scale distributed computing platform. By exploiting the mathematical properties of lattice, a clever vector pooling scheme is introduced to minimize the amount of information communicated among processes. By extending the well-recognized Ubiquity Generator (UG) framework [37] for Branch-and-Bound (B&B) algorithms, we have built a solid backbone to manage hundreds of thousands of processes running heterogeneous algorithms in parallel, where the assignment of algorithms and their parameters can be adaptively tuned according to the available resources and the progress of the whole system. The original UG framework has been successfully utilized for mixed-integer linear programming problems [29, 32, 30, 31], Steiner tree problems [13, 34, 33, 24], and quadratic assignment problems [11] on supercomputers. For lattice problems, the MAP-SVP, as mentioned above, is based on the original UG framework. However, most lattice algorithms are not B&B ones, and hence, MAP-SVP cannot utilize the full features of the original UG. The success of MAP-SVP motivated the UG project to refactor the original UG framework. The original UG codes have been

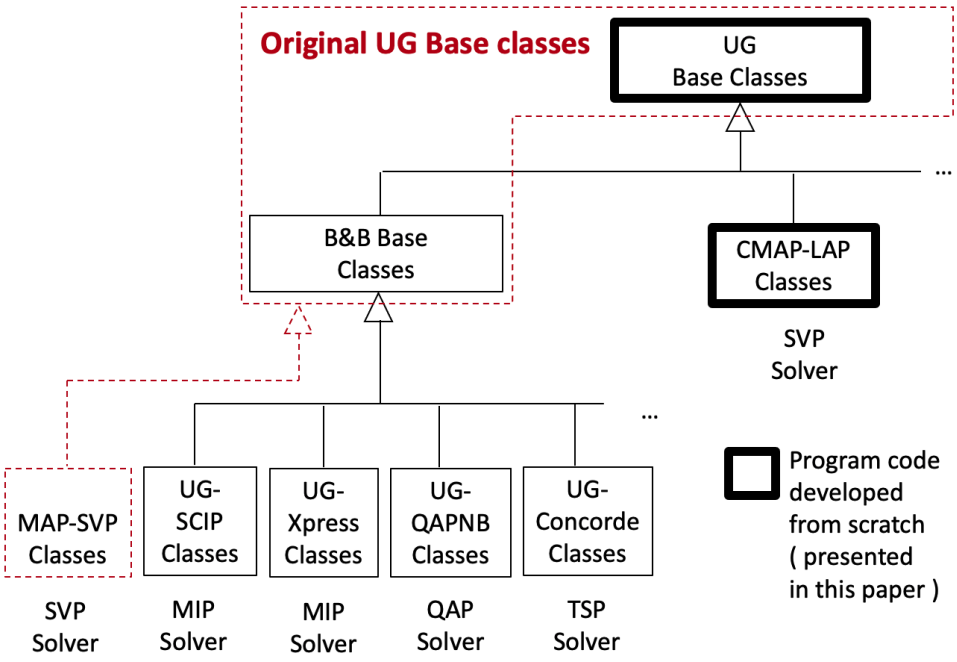

Figure 1: Refactoring of the UG framework

refactored into the *Generalized Ubiquity Generator framework* (Generalized UG)[1] to allow more flexibility necessary for lattice algorithms (see Figure 1). Particular emphasis is put on the efficient and versatile message-sharing mechanics. Based on the Generalized UG framework, we developed the *Configurable Massively Parallel Solver for Lattice Problems (CMAP-LAP)*[2].

Our contribution is summarized as follows:

- We propose a novel parallel and multi-algorithm scheme for lattice problems, in which several different single- or multi-rank solvers work cooperatively while sharing information efficiently with other solvers even on a large-scale computing platform (See Section 5, and detail for solving SVP is in Section 6.1). To realize the scheme, CMAP-LAP is developed entirely from scratch by fully utilizing the features of the Generalized UG.

- CMAP-LAP with $103,680$ cores stably and continuously ran for more than 42 hours. We tested CMAP-LAP in several environments with different scales and configurations (see Section 6).

- Each process asynchronously performs various lattice algorithms in coordination while sharing information. Processes for different algorithms are adaptively allocated, and

[1]The Generalized UG code will be released within the SCIP Optimization Suite [28] in 2021.

[2]pronounced "see" MAP-LAP to indicate that it provides new insights into the use of massive parallelism for lattice problem solvers.

their parameters are tuned according to the available resources, current progress, and estimated time for finding a solution. In particular, our accurate estimation of memory usage has drastically improved the stability and scalability (see Section 5.1.4).

- The high-level checkpoint-and-restart functionality is implemented to make it possible to save and resume even on different architectures and platforms of various sizes (see Section 5.1.5).

- The efficient information-sharing scheme is developed based on the properties of lattice problems, and is backed with blocking and non-blocking communication mechanisms (see Section 5.2.3).

- Highly modular architecture allows one to incorporate new algorithms easily into the system. Existing implementations that work only in a shared-memory environment can work as modules of CMAP-LAP, which run massively in parallel (see Section 5.2.1).

2 Lattice Problems

A (full-rank) *lattice* of dimension n is the set of all integral linear combinations

$$L = \mathcal{L}(\mathbf{b}_1, \ldots, \mathbf{b}_n) \coloneqq \left\{ \sum_{i=1}^{n} x_i \mathbf{b}_i : x_1, \ldots, x_n \in \mathbb{Z} \right\}, \tag{1}$$

where $\mathbf{b}_1, \ldots, \mathbf{b}_n$ are n linearly independent vectors in $\mathbb{R}^n$ for a positive integer n. The set of the n vectors $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ is called a *basis* of L. When another set of vectors $\{\mathbf{c}_1, \ldots, \mathbf{c}_n\}$ spans the same lattice L, it is also called a basis of L. Furthermore, $\mathcal{L}(\mathbf{B})$ denotes the lattice spanned by the row vectors of an invertible matrix $\mathbf{B}$. The $n \times n$ matrix $\mathbf{B}$ is called a *basis matrix* of L. Two matrices $\mathbf{B}$ and $\mathbf{C}$ span the same lattice if and only if there exists a unimodular matrix $\mathbf{T}$ satisfying $\mathbf{C} = \mathbf{TB}$. (An integral square matrix is called *unimodular* if its determinant equals ± 1.) Given a basis matrix $\mathbf{B}$ of L, the volume of L is defined as $\mathrm{vol}(L) \coloneqq |\det(\mathbf{B})|$, which is independent of the choice of basis matrices.

Lattice problems are algorithmic problems that involve lattices. Among lattice problems, the following is of fundamental importance:

Definition 1 (Shortest Vector Problem (SVP))**.** Find the shortest non-zero vector with respect to the ℓ_2-norm in the lattice $\mathcal{L}(\mathbf{B})$, given a basis matrix $\mathbf{B}$.

SVP is a discrete optimization problem for finding x_i's in (1) and is shown to be NP-hard under randomized reductions [1]. (That is, there exists a probabilistic Turing-machine that reduces any problem in NP to SVP instances in polynomial-time.) Note that the shortest vectors are not unique, and SVP asks to find one of them. The length of the shortest non-zero vector in L is denoted by $\lambda_1(L)$. SVP is the problem of finding a lattice vector $\mathbf{s} \in L$ with $\|\mathbf{s}\| = \lambda_1(L)$. It should be emphasized that there is no known NP algorithm

to check if $\|\mathbf{v}\| = \lambda_1(L)$ for a given $\mathbf{v} \in L$. Therefore, we rely on *Gaussian Heuristic*, which assumes that the number of vectors in $L \cap S$ is roughly equal to $\mathrm{vol}(S)/\mathrm{vol}(L)$ for a measurable set S in $\mathbb{R}^n$. By taking S to be the ball of radius $\lambda_1(L)$ centered at the origin $\mathbf{0}$ in $\mathbb{R}^n$, the Gaussian Heuristic leads to an estimation of $\lambda_1(L)$ as

$$\lambda_1(L) \approx \left(\frac{\mathrm{vol}(L)}{\omega_n}\right)^{1/n},$$

where ω_n denotes the volume of the n-dimensional unit ball. By Stirling's formula, we have $\omega_n \approx \left(\frac{2\pi e}{n}\right)^{n/2}$ as $n \to \infty$, and define

$$\mathrm{GH}(L) := \sqrt{\frac{n}{2\pi e}}\mathrm{vol}(L)^{1/n}. \tag{2}$$

Then, $\lambda_1(L) \approx \mathrm{GH}(L)$ holds for random lattices L in high dimensions $n \geq 40$. (Gaussian Heuristic does not hold in low dimensions.) For a vector $\mathbf{v} \in L$, the value $\|\mathbf{v}\|/\mathrm{GH}(L)$ is called the *approximation factor* of $\mathbf{v}$. Similarly, for a basis matrix $\mathbf{B}$, the value $\min_{1\leq i\leq n} \|\mathbf{b}_i\|/\mathrm{GH}(L)$ is called the approximation factor of $\mathbf{B}$. They are evaluation metrics for the lattice vector and the basis. Based on this observation, an approximate variant of SVP is defined:

Definition 2 (Hermite Shortest Vector Problem (HSVP))**.** Given a basis matrix $\mathbf{B}$ and an approximation factor $\gamma > 0$, find a non-zero vector $\mathbf{v} \in \mathcal{L}(\mathbf{B})$ such that $\|\mathbf{v}\| \leq \gamma \cdot \mathrm{vol}(\mathcal{L}(\mathbf{B}))^{1/n}$.

Another important lattice problem is:

Definition 3 (Closest Vector Problem (CVP))**.** Given a basis of a lattice L and a target vector $\mathbf{t}$, find a vector in L that is closest to $\mathbf{t}$.

CVP is a generalization of SVP because we can easily convert an instance of SVP to one of CVP. This implies that CVP is at least as hard as SVP. From a practical point of view, however, both problems are considered equally hard due to Kannan's embedding technique [16] that can transform CVP into SVP.

A particular case of CVP that we will use later in this paper is

Definition 4 (Bounded Distance Decoding (BDD))**.** Given a basis matrix $\mathbf{B}$ and a target vector $\mathbf{t}$ within distance $\alpha\lambda_1(L)$ of $L = \mathcal{L}(\mathbf{B})$ for a constant $0 < \alpha \leq \frac{1}{2}$, find a vector in L closest to $\mathbf{t}$.

There are other important lattice problems related to the security of modern lattice-based cryptosystems such as the learning with errors and NTRU problems (e.g., see [22]). Most lattice problems can be reduced to SVP or CVP, and hence, SVP and CVP are fundamental. As Kannan's embedding transforms CVP into SVP, we focus on SVP in this paper to simplify the narrative. However, the proposed methods are applicable to other lattice problems.

3 Related Work

We summarize existing lattice problem solvers with a particular emphasis on SVP. The Darmstadt SVP challenge [25] has been considered to be an established venue for assessing algorithms for SVP. Lattice bases for dimensions from 40 to 200 are made publicly available. More precisely, for each dimension and for an integer called the *seed*, a unique lattice basis is generated and listed. For each listed lattice L, any non-zero lattice vector with length less than $1.05\mathrm{GH}(L)$ is considered as a solution (recall Equation (2) for $\mathrm{GH}(L)$). In other words, it is a contest for solving HSVP with $\gamma = 1.05\sqrt{n/2\pi e}$, or equivalently, with approximation factor 1.05. Solutions with smaller approximation factors are closer to shortest. Precisely, the number of vectors with approximation factor f is estimated by f^n by the Gaussian Heuristic. For example, finding a vector in an 130 dimensional lattice with approximation factor 1.04 is $3.5(\approx 1.05^{130}/1.04^{130})$ times more difficult than finding one with 1.05. There is a fundamental difference in exact and approximate SVP solvers. In the latter, algorithms search for short vectors within a given approximate factor. In contrast, (even probabilistic) exact SVP solvers find a shortest vector with a positive probability.

3.1 Approximate-SVP solvers

We present recent works for solving the SVP challenge in high dimensions, where $n \geq 150$. Note that the approximation factors of most of the current records for dimensions greater than or equal to 150 are over 1.02, so they are not likely to be the shortest vectors. In early 2017, an SVP instance in dimension 150 was first solved with an approximation factor 1.04192. It was reported in [36] that it took 394 days using up to 864 machines. The work is based on the random sampling [26], which samples small x_i's in (1) until a short vector is found. After August 2018, a number of records for the SVP challenge in dimensions up to 155 were updated using the general sieve kernel, called G6K [2]. G6K supports a variety of lattice basis reductions and sieve algorithms (see Section 4). It provides a highly optimized, multi-threaded, and tweakable implementation as an open-source C++ and Python library. Most of the records for $n \geq 130$ and notably the current highest dimension record (180 dimensional) have been found using G6K. It was reported in [10] that the 180 dimensional record took 51.6 days on a single machine with 4 NVIDIA Turing GPUs, and its approximation factor was 1.04002. They used the sub-sieve strategy [9] of G6K, which is an approximate algorithm (see Section 4.3), and worked with projected lattices of up to dimension 146 to fit within 1.5 TB of RAM.

3.2 Exact-SVP solvers

We present several works solving exact-SVP based on ENUM (see Section 4). As described in the previous section, ENUM is asymptotically slower than sieve. However, it is a deterministic algorithm with polynomial-space (cf., sieve requires exponential-space). Parallelization for ENUM is conducted for traversing the enumeration tree by divide-and-conquer [7, 14, 17]. Another parallelization approach has been pursued by randomization.

Applying unimodular transformation to the basis vectors does not change the lattice, but it alters the enumeration tree. Hence, a parallel search can be conducted on the bases obtained by applying randomly generated unimodular matrices to the basis. Based on this idea, a shared-memory parallelized ENUM system based on randomization and pruning techniques was presented in 2019 [4]. It reported the running time of solving exact-SVP over 60 cores for dimensions up to 100. In 2020, a massive parallel exact-SVP solver, called MAP-SVP, was developed in [35] using the Ubiquity Generator framework [37]. It was the first distributed asynchronous cooperative solver based on randomization and ENUM with pruning techniques. MAP-SVP found solutions for many instances of the SVP challenge in dimensions up to 127. In particular, it took 147 hours to find a new solution in dimension 127 (with seed 3) using $100,032$ cores. The approximation factor of the solution is 0.97573, which is the smallest among the current records with dimensions over 120.

4 Lattice Algorithms

We summarize practical algorithms solving lattice problems, mainly SVP (see [21, 39] for a survey). We also discuss our extension of these algorithms for parallel computation.

The *Gram-Schmidt orthogonalization* of a basis $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ is the set of orthogonal vectors $\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*$ defined recursively by

$$\mathbf{b}_1^* \coloneqq \mathbf{b}_1, \quad \mathbf{b}_i^* \coloneqq \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{ij}\mathbf{b}_j^*, \quad \mu_{ij} \coloneqq \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\|\mathbf{b}_j^*\|^2} \quad (i > j) \tag{3}$$

for $2 \leq i \leq n$. Let $\mathbf{B}^*$ denote the matrix whose rows are the Gram-Schmidt orthogonalization of the basis with the basis matrix $\mathbf{B}$. Let $\mathbf{U} = (\mu_{ij})$ denote the lower triangular matrix given by (3) and $\mu_{ii} = 1$. Then, we have $\mathbf{B} = \mathbf{U}\mathbf{B}^*$, and hence, $\mathrm{vol}(L) = \prod_{i=1}^{n} \|\mathbf{b}_i^*\|$ for the lattice $L = \mathcal{L}(\mathbf{B})$. For each $1 \leq k \leq n$, define an orthogonal projection map as

$$\pi_k : \mathbb{R}^n \longrightarrow \langle \mathbf{b}_k^*, \ldots, \mathbf{b}_n^* \rangle_{\mathbb{R}}, \quad \pi_k(\mathbf{v}) = \sum_{i=k}^{n} \frac{\langle \mathbf{v}, \mathbf{b}_i^* \rangle}{\|\mathbf{b}_i^*\|^2} \mathbf{b}_i^* \quad (\mathbf{v} \in \mathbb{R}^n),$$

where $\langle \mathbf{b}_k^*, \ldots, \mathbf{b}_n^* \rangle_{\mathbb{R}}$ is the sub-vector space spanned by $\{\mathbf{b}_k^*, \ldots, \mathbf{b}_n^*\}$. The lattice in $\mathbb{R}^n$ spanned by projected vectors $\pi_k(\mathbf{b}_k), \ldots, \pi_k(\mathbf{b}_n)$ is denoted by $\pi_k(L)$ and called the *projected lattice*. The lattice $\pi_k(L)$ is of dimension $n - k + 1$ and its volume is equal to $\prod_{i=k}^{n} \|\mathbf{b}_i^*\|$.

4.1 Enumeration (ENUM)

ENUM is a deterministic algorithm solving SVP exactly. For an SVP instance of dimension n, the time complexity is $2^{O(n^2)}$, but the space complexity is a polynomial in n. Given a basis $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ of a lattice L, ENUM is based on a depth-first tree search for an integer combination $(v_1, \ldots, v_n)$ such that $\mathbf{s} = v_1\mathbf{b}_1 + \cdots + v_n\mathbf{b}_n$ has the shortest norm in $L \setminus \{\mathbf{0}\}$.

With the Gram-Schmidt information (3), the target vector can be written as

$$\mathbf{s} = \sum_{i=1}^{n} v_i \left(\mathbf{b}_i^* + \sum_{j=1}^{i-1} \mu_{ij} \mathbf{b}_j^* \right) = \sum_{j=1}^{n} \left(v_j + \sum_{i=j+1}^{n} \mu_{ij} v_i \right) \mathbf{b}_j^*$$

By the orthogonality of $\mathbf{b}_i^*$'s, the projected vector $\pi_k(\mathbf{s})$ has length

$$\|\pi_k(\mathbf{s})\|^2 = \sum_{j=k}^{n} \left(v_j + \sum_{i=j+1}^{n} \mu_{ij} v_i \right)^2 \|\mathbf{b}_j^*\|^2 \quad (1 \le k \le n).$$

Given a search radius $R > 0$, ENUM constructs an enumeration tree of depth n, whose nodes at depth $n - k + 1$ correspond to the set of all vectors in $\pi_k(L)$ with a maximum length of R. The key observation is that if a shortest vector satisfies $\|\mathbf{s}\| < R$, its projections satisfy $\|\pi_k(\mathbf{s})\|^2 \le R^2$ for all $1 \le k \le n$; hence, it appears as a leaf of the tree. These n inequalities provide an efficient enumeration of the tree. The total number of nodes to be searched can be estimated using the Gaussian Heuristic as $\sum_{\ell=1}^{n} H_\ell$, where

$$H_\ell \coloneqq \frac{R^\ell \omega_\ell}{\mathrm{vol}(\pi_{n+1-\ell}(L))} = \frac{R^\ell \omega_\ell}{\prod_{i=n+1-\ell}^{n} \|\mathbf{b}_i^*\|} \quad (1 \le \ell \le n).$$

Therefore, it is crucial to choose a good R, which is sufficiently small but larger than the shortest norm. One useful strategy is pruning [12] where a smaller tree is built by replacing the inequalities $\|\pi_k(\mathbf{s})\|^2 \le R^2$ by $\|\pi_k(\mathbf{s})\|^2 \le R_{n+1-k}^2$ with a shorter radii $R_1 \le \cdots \le R_n = R$ at each depth defined by a pruning strategy. This is a probabilistic method because it is not certain that $\mathbf{s}$ can be found in this pruned tree.

Another strategy is parallelization. We start with a sufficiently big R. When one instance finds a short vector, its norm R' is shared across all instances. Because the shortest norm should be less than or equal to R', we can replace R with R' to reduce the size of the enumeration tree.

We combine both strategies in CMAP-LAP.

4.2 Sieve

Given a lattice L of dimension n, sieve is a probabilistic algorithm that solves SVP exactly with a time complexity $2^{O(n)}$, which is asymptotically faster than ENUM. The downside is that it requires exponential space of $2^{\Theta(n)}$. Consider a ball S centered at $\mathbf{0}$ and of radius R with $\lambda_1(L) \le R \le O(\lambda_1(L))$. Then, Equation (2) implies $\#(L \cap S) = 2^{O(n)}$. ENUM performs an exhaustive search of $L \cap S$ by going through all the vectors in the union set $\cup_{k=1}^{n} (\pi_k(L) \cap S)$, whose total number is $2^{O(n^2)}$. In contrast, the sieve relies on the following observation. Let M be a set of vectors uniformly sampled from $L \cap S$. The shortest lattice vector would be included in M with a probability close to 1 if $\#M \gg \#(L \cap S)$. More precisely, there exists a vector $\mathbf{w} \in L \cap S$ such that $\mathbf{w}$ and $\mathbf{w} + \mathbf{s}$ are both contained in M with a positive probability for some shortest vector $\mathbf{s} \in L \setminus \{\mathbf{0}\}$. Therefore, $\mathbf{s}$ can be

found by computing differences of pairs in M. There are various implementations of sieve algorithms that differ mainly in how to sample M such as GaussSieve [19]. Similarly to ENUM, the choice of R is crucial to the sieve.

4.3 Project-and-lift

The computational complexity of every known algorithm for SVP is exponential. A workaround is to work with a smaller dimensional lattice and lift its shortest vector to find a short vector in the original lattice. A straightforward but effective approach is to project the original basis vectors by π_k for some $1 \leq k < n$. First, find shortest vectors in the projected $(n-k+1)$-dimensional lattice by, for example, ENUM or sieve, and lift them to the original lattice so that their projections by π_k coincides with the shortest vectors in the projected lattice. The latter lifting process is equivalent to BDD. In this manner, however, it is not guaranteed that a shortest vector will be found.

Sub-sieve is proposed in [9] which implements this idea using sieve. Specifically, a sieve algorithm is performed in a projected lattice $\pi_k(L)$ to obtain a list of short lattice vectors:

$$D_{k,\tau} := \{\mathbf{0} \neq \mathbf{v} \in \pi_k(L) : \|\mathbf{v}\| \leq \tau \cdot \mathrm{GH}(\pi_k(L))\}$$

for a constant τ such as $\tau = \sqrt{\frac{4}{3}}$. In practice, k is chosen to be around $n - 30$ for high-dimensional lattices [2, 10]. Then, by Babai's algorithms [3], the short vectors in the inverse image $\pi_k^{-1}(D_{k,\tau}) \subset L$ are enumerated.

We introduce the novel sub-ENUM algorithm, which is suitable for massive parallelization. The first part is very similar to sub-sieve. An ENUM algorithm is performed in a projected lattice $\pi_k(L)$ to obtain a list of short lattice vectors $D_{k,\tau}$. Then, instead of Babai's algorithms, an ENUM algorithm is again used to find a shortest vector for a k-dimensional lattice spanned by $\{\mathbf{b}_1, \ldots, \mathbf{b}_{k-1}, \mathbf{v}\}$.

4.4 Basis reduction

Given a basis of a lattice, basis reduction algorithms seek for a new basis of the same lattice with short and nearly orthogonal basis vectors (such basis is called *reduced* or *good*). Below, we introduce several practical algorithms. These algorithms do not always find the shortest vector, but they are much faster than exact-SVP solving algorithms, such as ENUM and sieve. In practice, lattice basis reduction is performed as a pre-processing step of ENUM and sieve to reduce their expensive cost. In contrast, short (not necessarily shortest) vectors found by ENUM and sieve can be used in conjunction with lattice basis reduction algorithms to obtain better bases. Our CMAP-LAP cleverly manages this mutual dependency.

Lenstra-Lenstra-Lovász (LLL) Given a parameter $\frac{1}{4} < \delta < 1$, a basis $\{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$ is δ-*LLL-reduced* if it satisfies the following two conditions: (i) (Size-reduced) The Gram-Schmidt coefficients satisfy $|\mu_{ij}| \leq \frac{1}{2}$ for all $i > j$. (ii) (Lovász' condition) $\delta\|\mathbf{b}_{k-1}^*\|^2 \leq \|\pi_{k-1}(\mathbf{b}_k)\|^2$

for all $2 \leq k \leq n$. An LLL-reduced basis can be found by the LLL algorithm [18], which swaps adjacent basis vectors $\mathbf{b}_{k-1}$ and $\mathbf{b}_k$ iteratively if Lovász' condition does not hold.

LLL with deep insertions (DeepLLL) It is a simple generalization of LLL [27], in which the swapping is replaced with so-called *deep insertion*. If $\|\pi_k(\mathbf{b}_i)\|^2 < \delta\|\mathbf{b}_i^*\|^2$ for some $i < k$, the k-th vector $\mathbf{b}_k$ is inserted before $\mathbf{b}_i$ as $\{\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \mathbf{b}_k, \mathbf{b}_i, \ldots, \mathbf{b}_{k-1}, \mathbf{b}_{k+1}, \ldots, \mathbf{b}_n\}$. In this case, the new Gram-Schmidt vector at the i-th position is given by $\pi_i(\mathbf{b}_k)$, which is strictly shorter than the old Gram-Schmidt vector $\mathbf{b}_i^*$.

Block-Korkine-Zolotarev (BKZ) Given a basis $\mathbf{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_n\}$, let $\mathbf{B}_{[i,j]} = \{\mathbf{b}_i, \ldots, \mathbf{b}_j\}$ for $i \leq j$. Denote by $\mathbf{B}^*_{[i,j]}$ the projected vectors $\{\pi_i(\mathbf{b}_i), \pi_i(\mathbf{b}_{i+1}), \ldots, \pi_i(\mathbf{b}_j)\}$, and by $L_{[i,j]}$ the projected lattices spanned by $\mathbf{B}^*_{[i,j]}$. For a block size $2 \leq \beta \leq n$, the basis is *β-BKZ-reduced* if it is size-reduced and it satisfies $\|\mathbf{b}_j^*\| = \lambda_1(L_{[j,j+k]})$ for every $1 \leq j < n$ with $k = \min(\beta - 1, n)$. A β-BKZ-reduced basis can be found by the BKZ algorithm [27], which calls LLL to reduce every local block $\mathbf{B}_{[j,j+k]}$ and ENUM or sieve to find the shortest vector in the projected lattice $L_{[j,j+k]}$. When $\beta = n$, BKZ finds the shortest vector of L. In general, the parameter β controls the trade-off between the quality of the basis and the computational cost. The computational complexity of BKZ is not known.

DeepBKZ This is an enhancement of BKZ algorithm [38], in which DeepLLL is called a subroutine alternative to LLL. Experiments in [38, 40] show that short lattice vectors can be found by DeepBKZ with a smaller block size β than BKZ.

Multi-Share DeepBKZ We introduce a novel parallelized extension of DeepBKZ, which exploits the data-sharing scheme of CMAP-LAP. The DeepBKZ algorithm searches for a vector $\mathbf{v}$ satisfying $0 < \|\pi_j(\mathbf{v})\| < \|\pi_j(\mathbf{b}_j)\|$ from a restricted search range $\{\mathbf{v} \mid \pi_j(\mathbf{v}) \in L_{[j,j+k]}\}$. It relies on ENUM or sieve for the search of $\mathbf{v}$, and increasing the search range (via the choice of the parameter β) results in a better lattice basis with a higher computational cost. If there is a pool of short vectors, the DeepBKZ algorithm can look up the pool for $\mathbf{v}$. The pool is shared with other instances of the DeepBKZ algorithm, even with different lattice algorithms.

5 Design of CMAP-LAP

It is essential for a practical solver to utilize the multiple lattice algorithms introduced in Section 4. Most of the existing solvers discussed in Section 3 rely on either the combination of lattice reduction and sieve or the combination of lattice reduction and ENUM. These algorithms are inter-dependent and executed sequentially. In contrast, CMAP-LAP is built on a new multi-algorithm paradigm in which multiple lattice algorithms are executed cooperatively and yet asynchronously in parallel. The key idea is that each lattice algorithm described in Section 4 can be considered a *sampler* of short lattice vectors. Furthermore,

each algorithm benefits from the knowledge of short vectors; for example, the enumeration tree of ENUM shrinks according to the upper bound R of the shortest norm. Using different algorithms and randomly transformed bases, we can increase the number of samplers, which mutually boosts the sampling performance by sharing the information of short vectors found (see Figure 2). To realize the novel multi-algorithm paradigm, CMAP-LAP was developed entirely from scratch utilizing the full power of the Generalized UG, which is a generic high-level task parallelization framework.

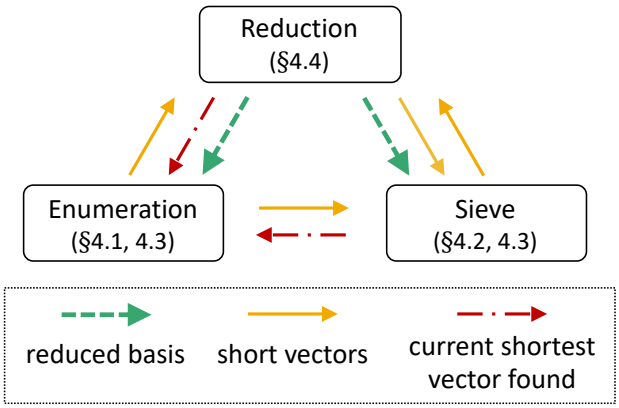

Figure 2: Interaction among SVP algorithms: Basis reduction generates a reduced basis, over which enumeration and sieve can find short vectors efficiently. In contrast, enumeration and sieve find short vectors so that basis reduction accelerates to find a more reduced basis.

5.1 Architecture of CMAP-LAP

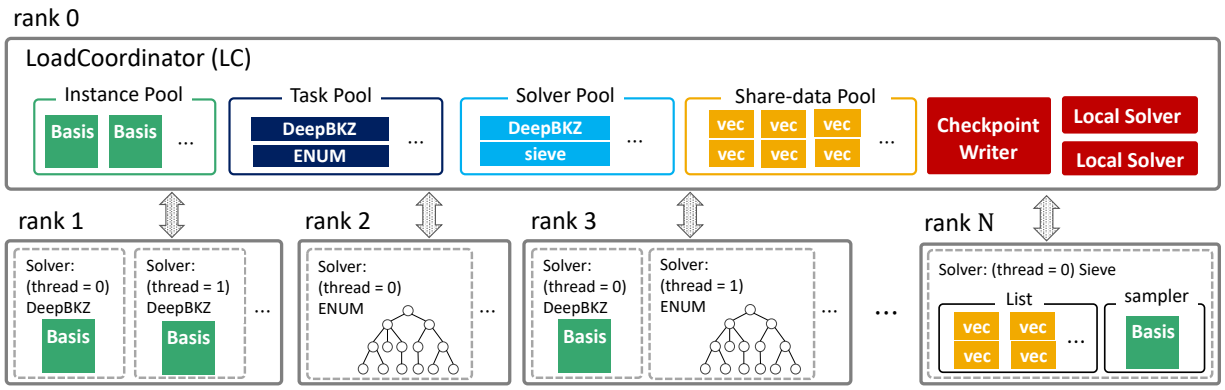

Figure 3: System overview of CMAP-LAP for SVP

We describe the architecture of CMAP-LAP. The Generalized UG consists of a controller process, *LoadCoordinator* (`LC`), and multiple ***`Solver`***s. Each `Solver` communicates with `LC`

asynchronously. This system is suitable for multiple processes that run different algorithms and share information, as needed. CMAP-LAP adopts the Supervisor-Worker load coordination paradigm (see [23]), where LC is Supervisor and Solvers are Workers. The main difference to the typical Master-Worker paradigm is that the Supervisor's task is limited and Workers act more independently by exchanging small messages with Supervisor as needed, avoiding unnecessary overhead to manage Workers. The LC has the following data pools: 1) Instance Pool, 2) Solver Pool, 3) Task Pool, and 4) Share-Data Pool. (See Figure 3). The LC creates special purpose local threads as needed: 1) Checkpoint Writer thread 2) Local Solver threads.

Each Solver carries a *Task*, which is a triple of:

- *Instance* is the data that represents the problem to solve, which in the case of SVP is a lattice basis, and in the case of CVP is a lattice basis and a target vector.
- *Parameters* describe the type of algorithm and the parameters of the algorithm. For example, an ENUM algorithm with a pruning strategy from *Parameters*.
- *Status* represents the algorithm's progress, e.g., for the depth-first search of the enumeration algorithm, it is the node currently being searched.

Given a lattice problem, each Solver is created in one core and assigned a *Task* by LC. The basic flow of CMAP-LAP is as follows (see Figure 4):

1. LC stores given *Instance* in the instance pool.
2. LC pops an *Instance* from the instance pool, sets *Parameters* for *Instance*, and initializes *Status*. The created *Task* = (*Instance*, *Parameters*, *Status*) is stored in the task pool.
3. If there exists an idle Solver, LC pops a *Task* in the task pool and sends it to the idle Solver, and stores it to the solver pool.
4. Each Solver takes the algorithm and its input from the received *Task*, and occasionally shares information to LC, such as *Instance*, Data, *Status*. The information sent depends on the algorithm, as shown in Figure 2. LC stores the information in the pool according to this type. In addition, Solver sends its *Status* to LC, and LC updates *Task* in the solver pool for the checkpoints.
5. Information in the share-data pool is occasionally retrieved from LC, and shared among Solvers. Each Solver updates its *Parameters* according to the shared information. See Section 4 for how the shared information is utilized by each algorithm run by the Solver.
6. When a Solver finishes the assigned *Task*, it sends its final *Status* to LC and becomes idle.

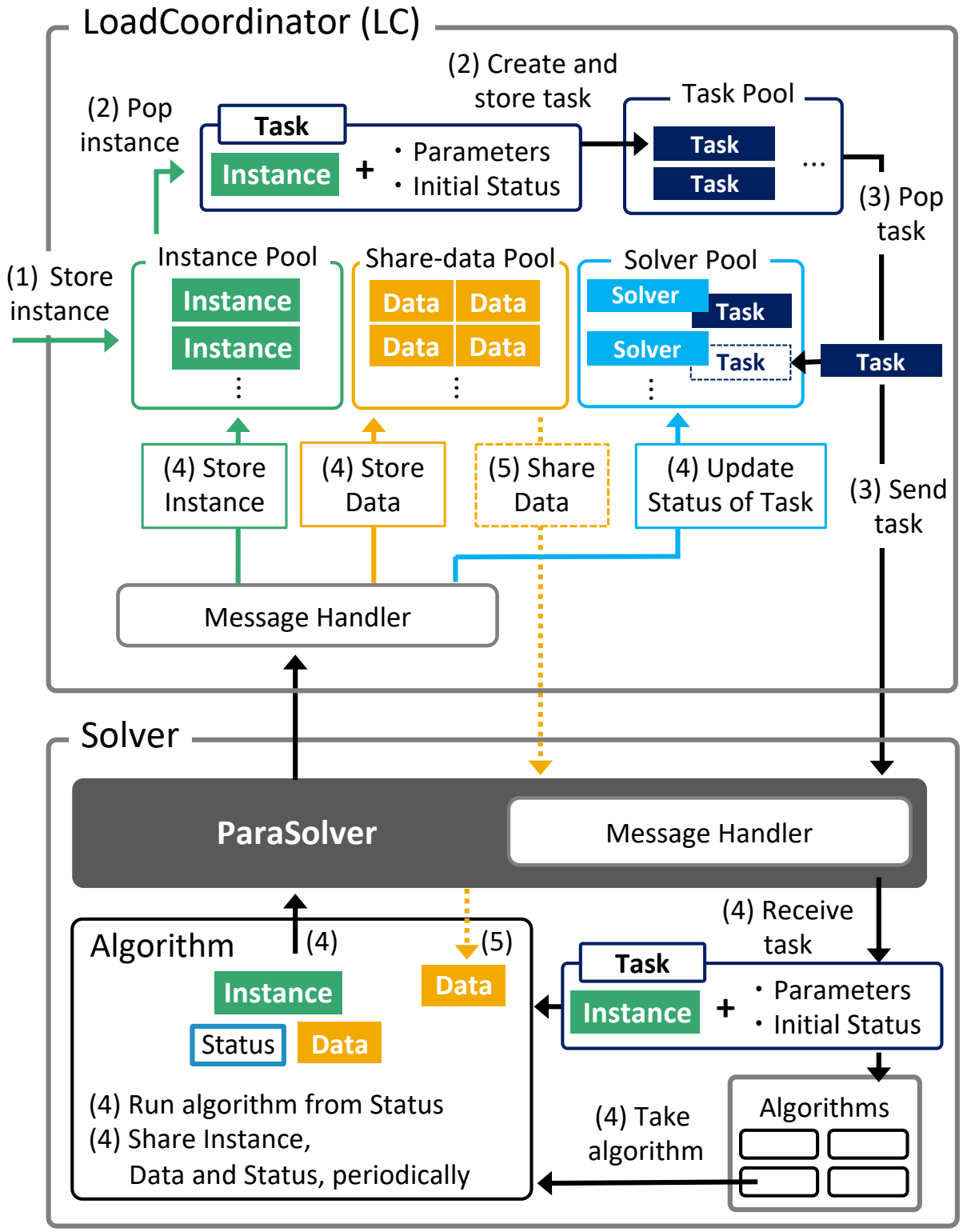

Figure 4: Execution flow of CMAP-LAP

LC always checks for messages from Solver. Messages received by the LC are processed it through the message handler according to the type of message. As described above, Solver only communicates with LC, and Solver does not share information with other Solvers directly. This communication via the share-data pool is an effective solution for massive parallelization to achieve 1) the reduction in the number of communication paths, 2) the management of the total amount of communication, 3) the control over the memory usage, and 4) I/O for checkpoint and progress takes place solely within LC.

The detail of the components of CMAP-LAP is given as follows.

5.1.1 Instance Pool

Instance pool stores instances of the problem together with their priorities. For example, bases transformed by unimodular matrices give the same lattice and represent different instances of the same lattice problem. Provided a lattice basis that specifies the lattice problem, the instance pool is initialized with the single basis. LC stores bases sent from Solvers, which run the reduction algorithm. In the case of SVP, the priority can be computed by the estimated total number of nodes in the enumeration tree described in Section 4.1 such that the shortest vector will be found more efficiently with an instance of higher priority. LC pops an instance with the highest priority from the instance pool and creates a *Task* from it.

5.1.2 Task Pool

Task pool stores *Task*s, which are triples of (*Instance*, *Parameters*, *Status*). It manages the *Task*s waiting to be executed. LC assigns the *Task* with the highest priority to a Solver. In this way, the *Task*s which would lead to better solutions quickly, are prioritized. Multiple *Task*s may be generated from a single instance using different algorithms and parameters.

5.1.3 Solver Pool

Solver pool stores information of the running Solvers. Each Solver is managed by (Solver Id, *Task*). The *Status* of *Task* is periodically updated by the *Status* message sent from Solver. This allows LC to grasp the status of all Solvers. When Solver finishes the assigned *Task*, it is registered as idle. In addition, when LC wants to assign a new *Task* of high priority immediately, LC chooses a running Solver to interrupt the current *Task*.

The number of active Solvers that runs on a single machine node is determined by LC according to the computational cost of *Task*. For example, sieve algorithms have a large memory footprint to maintain a large number of lattice vectors; a single Solver becomes active and runs on a single machine node. Meanwhile, ENUM and reduction algorithms use little memory, and the same number of Solvers as that of the cores run on a single node.

5.1.4 Share-Data Pool

Share-data pool stores information that is shared across multiple `Solver`s. In the case of CMAP-LAP, a typical type of information sent from `Solver`s is a lattice vector of small norm. The size of the message is equal to the product of the dimension (e.g., 130) and the size of the scalar (e.g., long integer). `LC` checks if the sent vector is already in the pool. If it is not in the pool, an entry (Data, Sent-Solvers, priority) is created in the pool, where Data is the sent vector. Sent-Solvers is a set that records the Solver Ids to which Data has been sent. The priority is computed by its norm. When the pool size gets bigger, `LC` decides which entries remain stored in the pool according to their priorities. At an interval, `LC` selects an entry according to the priority and pushes it to those `Solver`s whose Solver Ids are not in Sent-Solvers and adds their Solver Id to Sent-Solvers. In this way, information is shared among all `Solver`s efficiently while controlling the total amount of communication. The interval at which `Solver`s and `LC` push information can be tuned depending on the configuration of the machine. There is no danger of locking regarding the order of messages in our scheme.

The share-data pool is the most memory-consuming part of the `LC`. The size of share-data pool increases over time, and the limit of the pool size must be set appropriately according to the available memory. In particular, the size of the Sent-Solvers is dominant and should be estimated carefully in case of massive parallelization. Moreover, the cost of Data retrieval increases when the pool size and the number of `Solver`s are large. In this case, the limit of the pool size and the frequency of data sharing are suppressed.

5.1.5 Fully Checkpoint Functionality with Checkpoint Writer thread

One of the most powerful features of CMAP-LAP is the checkpoint mechanism for storing high-level information of the whole system. Lattice problems are hard and often require millions of core hours. Thus, it is critical to have the functionality to record the progress and resume after interruption. Our checkpoint functionality is carefully designed so that high-level, platform-independent information is stored to enable restart even on different platforms.

When a checkpoint is requested, the data in the pools in `LC` are serialized and stored in checkpoint files using zlib [8], a portable compression library. At the time of restart, CMAP-LAP reads the checkpoint files to restore pools. The task pool contains *Task*s, including the progress information *Status*, which can be assigned to `Solver`s to resume. When the checkpoint files are loaded in a different environment from the one that has saved them, the number of cores and the available memory may be different. In this case, `LC` distributes the *Task*s in the task pool to `Solver`s as much as possible, leaving the other *Task*s in the task pool. At the same time, `LC` creates new *Task*s when a large number of `Solver`s are available.

The technically important point is that the message processing from `Solver`s to `LC` is blocked when `LC` writes checkpoint files. With many MPI packages, this is problematic because the size of the queue of MPI messages waiting to be received becomes large and

eventually leads to an error when the upper limit is reached. This problem becomes more pronounced for larger-scale execution. To avoid this problem, LC temporarily creates a copy of the pools on memory, and a dedicated thread in LC, called *Checkpoint Writer*, is created to write the copy in the checkpoint files. This has significantly reduced the block time for checkpoints and enabled CMAP-LAP to run stably on large-scale platforms.

5.1.6 Local Solver threads

Some solvers can be created as a thread in LC. These *Local Solver*s work on lightweight tasks requiring access to the entire pools. For example, Local Solvers list the projected vectors in the share-data pool, which are found by Solvers performing sub-ENUM and sub-sieve. Because Local Solvers have access to the share-data pool without communication, the total amount of communication is reduced in this way.

5.2 Implementation Technicalities

5.2.1 Extendability

There are many lattice problem solvers, including the state-of-the-art sieve solver G6K, which is available as open-source software. CMAP-LAP's flexible and highly modular design allows solvers to be incorporated as a part of the system. For the ease of incorporation, an interface class *ParaSolver* is provided, with which existing solvers can be turned into Solvers with minimum effort. Each Solver has a ParaSolver object that takes care of all the communication, and existing solvers only have to receive input data and send the results via ParaSolver's API (see bottom of Figure 4). The solvers are not limited to single-rank applications. The UG has a feature to parallelize multi-rank applications. See [20] as an example.

5.2.2 Hybrid Parallelization

CMAP-LAP uses hybrid parallelization that combines MPI communication with C++11 thread communication. LC and Solver have two kinds of communicators: one is *ParaComm*, which wraps MPI communication functions, and the other is *LocalComm*, which wraps C++11 communication functions. ParaComm is used for inter-process communication, and LocalComm is used for inter-thread communication within a process (see Figure 5). Because all Solvers know the MPI rank of LC, Solvers send messages directly to LC using ParaComm and ISendQueue, which is described in the following section. In contrast, when LC sends a message to Solver, LC first sends a message via ParaComm to the MPI rank where the Solver resides. The solver with 0 thread-Id receives the message; we call this the *rootSolver*. Then, the rootSolver sends the message to the Solver using LocalComm. Therefore, the rootSolver receives more messages than the other Solvers, the received messages must be checked frequently, even during the execution of the algorithm. However, the idle time for message processing can be reduced by using non-blocking communication, as described below.

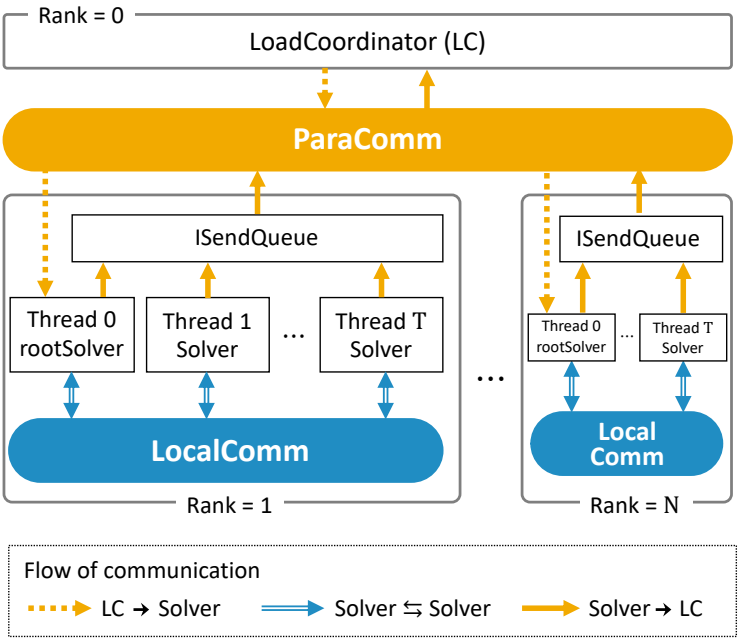

Figure 5: Communicators between and within MPI processes: ParaComm and LocalComm

5.2.3 MPI_ISend Communication

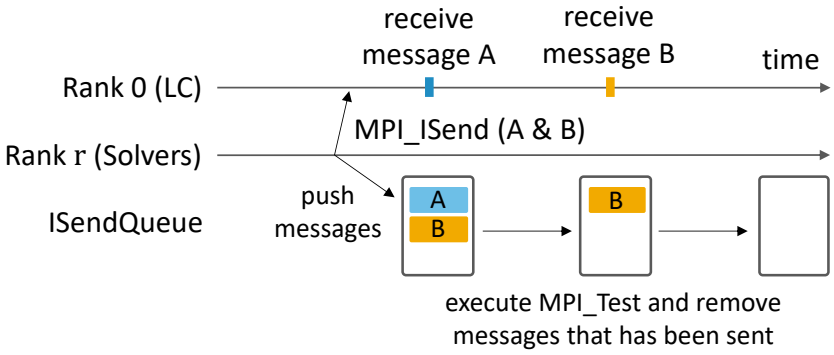

Figure 6: MPI_ISend Communication between `Solver` and `LC`

Because `LC` receives messages from all busy `Solvers`, the `LC`'s load is the highest of all the processes in the case of large-scale computation. In addition, depending on the type of messages received, processing such as inserting Data into the share-data pool occurs in `LC`. This blocks the `LC` message processing and delays the receiving of the messages. Note that the load coordination paradigm used in CMAP-LAP is Supervisor-Worker [23] and then small message communications are performed between `LC` and `Solvers` for load balancing. Although the frequency for the small message communications can be controlled by run-time parameters, they are crucial in large-scale computations such as over $100,000$ `Solvers` used. Therefore, in CMAP-LAP, to reduce the idle time of communication in `Solver`, we send all

messages from `Solver` to `LC` by using MPI_ISend, the non-blocking communication. This leads `Solver` to resume the algorithm without waiting for the check that `LC` receives the message. To prevent the objects deleted before they are sent, we copy the objects sent by MPI_ISend to a queue called *ISendQueue* in the memory of that process. We remove them from ISendQueue as soon as the transmission is confirmed by MPI_Test (see Figure 6). By examining the size of each ISendQueue, we can determine the number of unreceived messages of `LC`. Therefore, we set an upper limit on the size of ISendQueue and do not send messages exceeding the limit, thereby preventing many messages from accumulating in `LC`.

6 Numerical Experiments

In this section, we evaluate the performance of CMAP-LAP with the SVP challenge. The computing platform used in the following numerical experiments includes the Lisa and Emmy at Zuse Institute Berlin, and ITO at Kyushu University. These specifications are summarized in Table 1.

Table 1: Computing platforms used

Machine	Memory / node	CPU	CPU frequency	# of nodes	# of cores
Lisa (HLRN IV)	384 GB	Intel Xeon Platinum 9242 (CLX-AP)	2.30 GHz	1,080	103,680 (96 × 1,080)
Emmy (HLRN IV)	384 GB	Intel Xeon Platinum 9242 (CLX-AP)	2.30 GHz	128	12,288 (96 × 128)
ITO	192 GB	Intel Xeon Gold 6154 (Skylake-SP)	3.00 GHz	128	4,608 (36 × 128)
CAL A	256 GB	Intel(R) Xeon(R) CPU E5-2640 v3	2.60 GHz	4	64 (16 × 4)
CAL B	256 GB	Intel(R) Xeon(R) CPU E5-2650 v3	2.30 GHz	4	80 (20 × 4)

6.1 Solving SVP with CMAP-LAP

We briefly describe the overall behavior of CMAP-LAP for solving SVP. Recall that an SVP is specified by a lattice basis matrix. At the beginning of the execution, the `LC` reads the basis matrix from a file and stores it in the instance pool. `LC` creates a Local Solver to transform the basis with random unimodular matrices and stores the resulting bases in the instance pool. Then, `LC` generates DeepBKZ *Task*s for the bases in the instance pool. The reduced bases are sent from `Solver`s performing DeepBKZ *Task*s to `LC`, and `LC` stores them in the instance pool. `LC` also generates ENUM and sieve *Task*s using the bases in the instance pool. Short lattice vectors are occasionally sent from `Solver`s to `LC`, which are inserted into the share-data pool. At regular intervals, `Solver`s request `LC` to send short vectors from the share-data pool. DeepBKZ *Task*s insert the received short vectors into the basis, sieve *Task*s use the received short vectors as sampling seeds, while ENUM adjusts the search radius according to the norm of the shortest vector ever found. Some of the contents of the pools in `LC` are written to checkpoint files at regular intervals: vectors in the data-share pool, basis matrices in the instance pool, and *Task*s in the solver pool. The *Task* mainly contains the basis matrix and the vector and parameters needed to run the algorithm. When restarting, as described in Section 5.1.5, there are few processes other

than reading checkpoint files. We calculate the communication interval and the number of vectors shared from the number of cores, and the maximum MPI buffer size to relax the communication delay.

Since computing the exact norm of a shortest vector of a given lattice is as hard as computing a shortest vector, we evaluate the progress of solving an SVP instance by the approximation factor defined in Section 4. A smaller value of the approximation factor indicates a better (temporary) solution. With the Gaussian Heuristics, the approximation factor should be about 1.0 for a good candidate of a shortest vector. From a cryptanalysis viewpoint, an approximate factor of 1.05 is often set as a goal as in the SVP challenge. The numbers of lattice vectors having smaller approximation factors decrease quickly; for example, in dimension $n = 130$, the ratio of the numbers of lattice vectors having approximation factors 1.20 and 1.30 is approximately $(1.20^n/1.30^n) \approx 3.03 \times 10^{-5}$. In other words, it is $33,000$ times harder to reach an approximate factor of 1.20 than of 1.30. It becomes increasingly harder to find lattice vectors with smaller approximate factors; for example, the ratio of the numbers of lattice vectors having approximation factors 1.10 and 1.20 is approximately $(1.10^n/1.20^n) \approx 1.22 \times 10^{-5}$.

6.2 Information sharing

We evaluate the effect of our novel information-sharing scheme and the parallelization with the lattice reduction algorithm. We performed experiments running DeepBKZ with $\beta = 30$ for five instances of the SVP challenge of dimension 130 with seeds from 0 to 4. We executed all computations on the CAL A and CAL B with 144 cores.

We show the efficiency of the information sharing with CMAP-LAP. In CMAP-LAP, `Solver`s share multiple short lattice vectors via the share-data pool in `LC`. The amount of information shared among `Solver`s can be controlled by the size of the share-data pool. Figure 7 compares the transition of the approximation factor (averaged over 5 instances) overtime with the size of the share-data pool 0, 1, and $100,000$. When the size of the share-data pool is set to zero, no information is shared and all the `Solver`s are executed independently. When the size of the share-data pool is set to 1, only the current shortest lattice vector (the current solution) is shared among `Solver`s. This is equivalent to the sharing scheme of MAP-SVP. We observe that the approximation factor is drastically reduced when the size of the share-data pool is set to $100,000$. This shows the effectiveness of our data sharing scheme.

6.3 Coordination of heterogeneous algorithms

We show the effectiveness of CMAP-LAP's multi-algorithm paradigm, in which heterogeneous lattice algorithms are executed concurrently in coordination. In this experiment, we fix the number of `Solver`s assigned to each *Task*, that is, DeepBKZ, sub-ENUM, and GaussSieve. Each `Solver` is assigned the the same type *Task* when it completes the current *Task*. Figure 8 and 9 shows the results for a 110- and 130-dimensional SVP with four different configurations of the *Task* assignment, respectively. We ran the experiment on the CAL

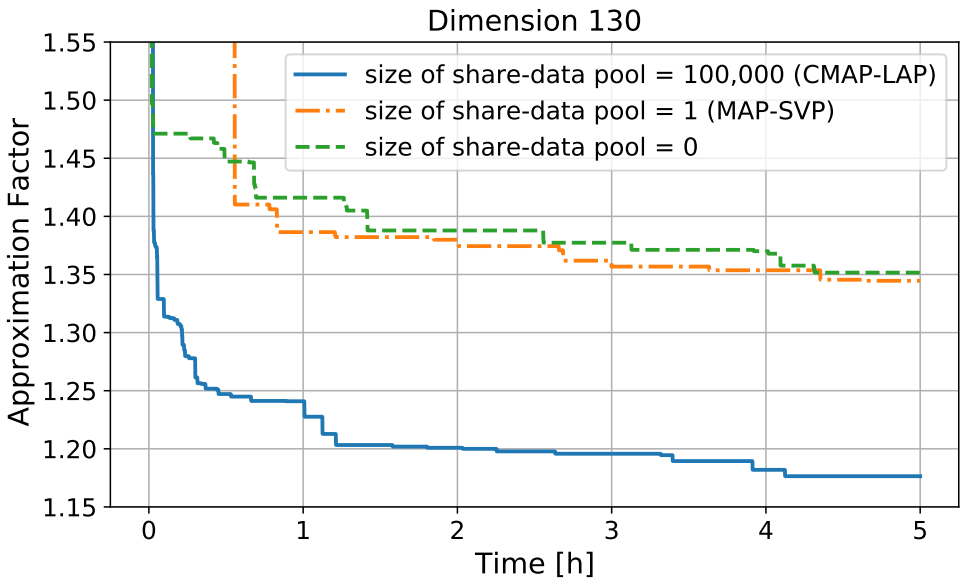

Figure 7: Transition of the approximation factors for different share-data pool sizes; execution were done on the CAL A and CAL B with 144 cores. The solid blue lines in Figure 7, 9 and 11 represent the same experimental result.

A and CAL B with 144 cores for an hour or five hours, and the 1 core was assigned to `LC`, and the other 143 cores were assigned to three types of *Task*s. We set the size of the share-data pool to be infinity. The best result was obtained with the combination of (DeepBKZ, sub-ENUM, GaussSieve) = (110, 32, 1). To investigate the reason, we examine the distribution of vector norms in the share-data pool for two configurations of 130-dimensional experiments (see Figure 10). The total number of vectors shared through the share-data pool for (DeepBKZ, sub-ENUM, GaussSieve) = (143, 0, 0) was 36,055, and that for (DeepBKZ, sub-ENUM, GaussSieve) = (110, 32, 1) was 101,952. In both configurations, shorter vectors were found by DeepBKZ `Solver`. However, a large number of relatively short vectors found by sub-ENUM and GaussSieve helped DeepBKZ find shorter vectors.

6.4 Scalability

To see the scalability of CMAP-LAP, we experimented with the same 130-dimensional SVP instances as in Section 6.2 on Lisa using 2,976, 6,048, 12,192, 24,480, and 49,056 `Solvers` with DeepBKZ ($\beta = 30$). We measured the average number of the main iterations (called the *tour*) performed by each `Solver` within six hours. The number of tours provides a good estimation of the progress of the DeepBKZ algorithm. As we observe from Table 2, the average number of tours stay almost constant when the number of `Solvers` increases. Therefore, even if the number of `Solvers` becomes large-scale, there is no significant change in the performance of each `Solver`.

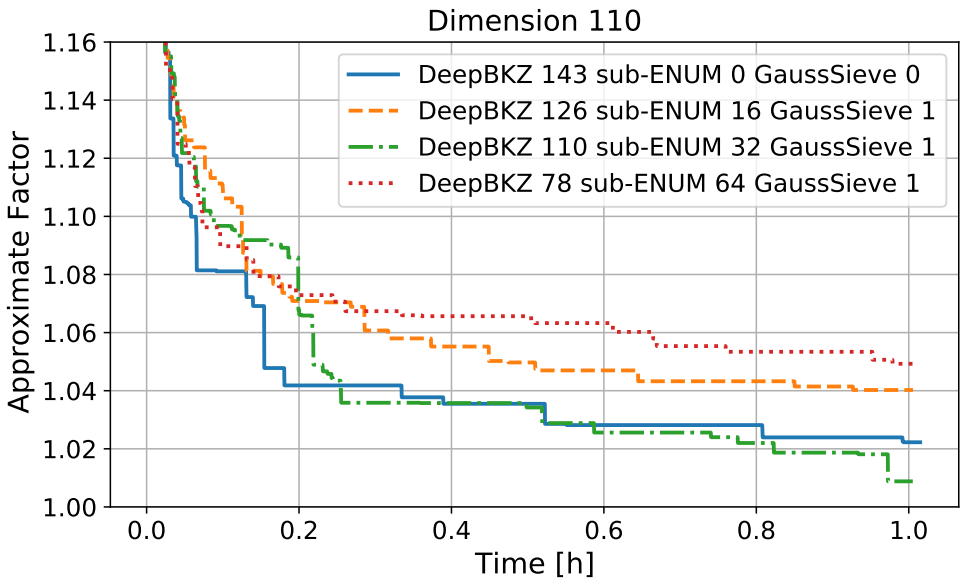

Figure 8: Same as Figure 7, but dimension is 110 and different allotment of algorithms; execution were done on the CAL A and CAL B with 144 cores.

Table 2: Iterations of DeepBKZ of each `Solvers` for 130-dimensional SVP

# of `Solvers`	2,976	6,048	12,192	24,480	49,056
averaged # of iterations	45.86	43.32	43.08	40.10	57.07

In addition, we evaluated the effect of parallelization on the transition of the approximation factor (see Figure 11). We experimented with the same SVP instances as in Section 6.2 using different numbers of `Solvers`. The size of the share-data pool was set to $100,000$. We used the CAL A and CAL B with 144 cores and ITO with $2,304$ cores for this experiment. The best (minimum) approximation factor obtained within 5 hours with 143 `Solvers` was 1.176 and 1.117 with $2,303$ `Solvers`. In terms of Gaussian Heuristics, the latter is considered to be $1.176^{130}/1.117^{130} \approx 800$ times better. It took $14,844$ seconds to reach the approximation factor of 1.176 with 143 `Solvers` while it took $2,965$ seconds with $2,303$ `Solvers`, which is a speed-up by a factor of 5.0 compared with 143 `Solvers`. Similarly, the time for the approximation factor to fall below 1.2 was $7,319$ seconds with 143 `Solvers` and $1,360$ seconds with $2,303$ `Solvers`, which is a speed-up by a factor of 5.3.

6.5 Stability with massive parallelization

We show the results of a long-time execution of CMAP-LAP.

Figure 12 shows the result of multiple executions of a 134 dimensional SVP instance. We ran the experiment 13 times using our checkpoint-and-restart functionality on the

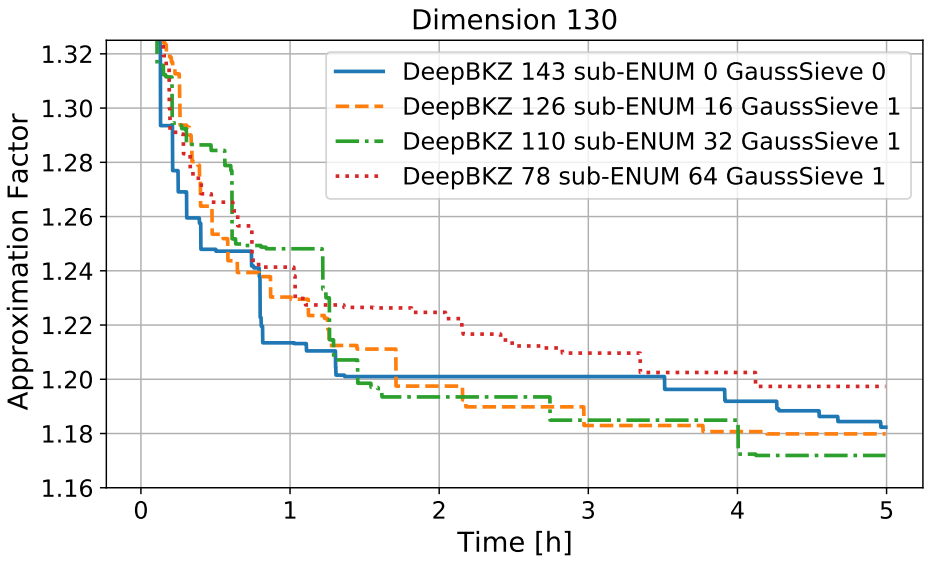

Figure 9: Same as Figure 7, but for different allotment of algorithms; execution were done on the CAL A and CAL B with 144 cores.

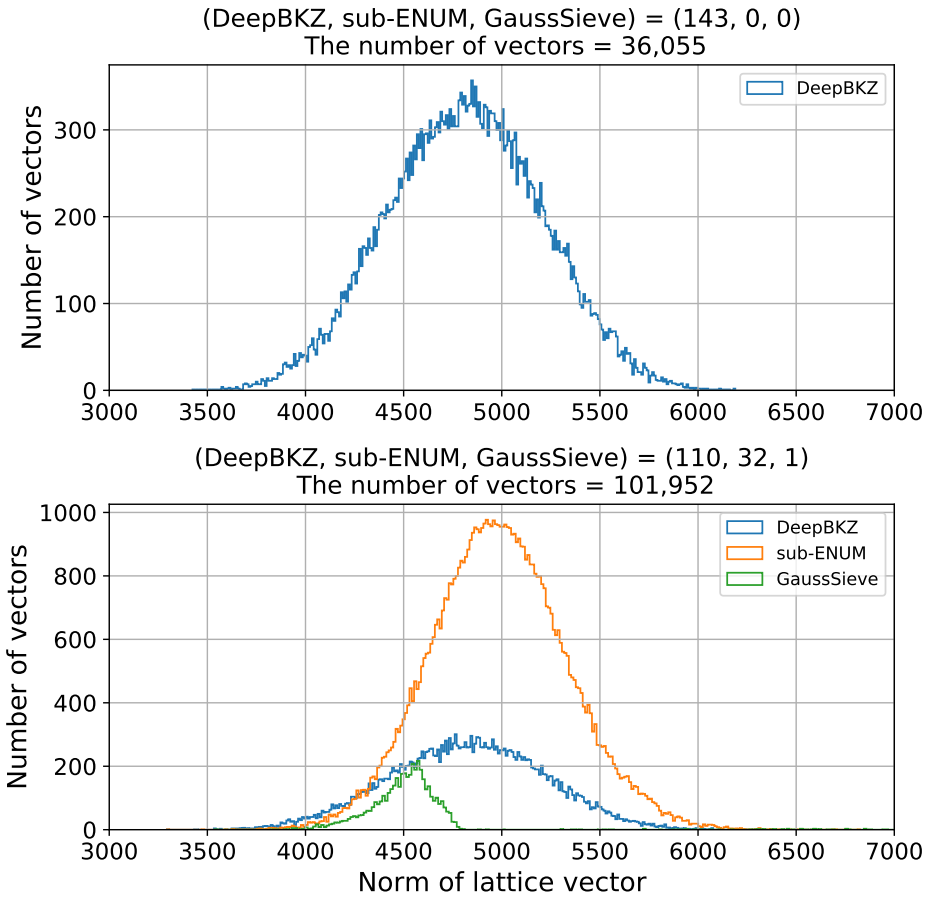

Figure 10: Distribution of the norm of vectors in the share-data pool.

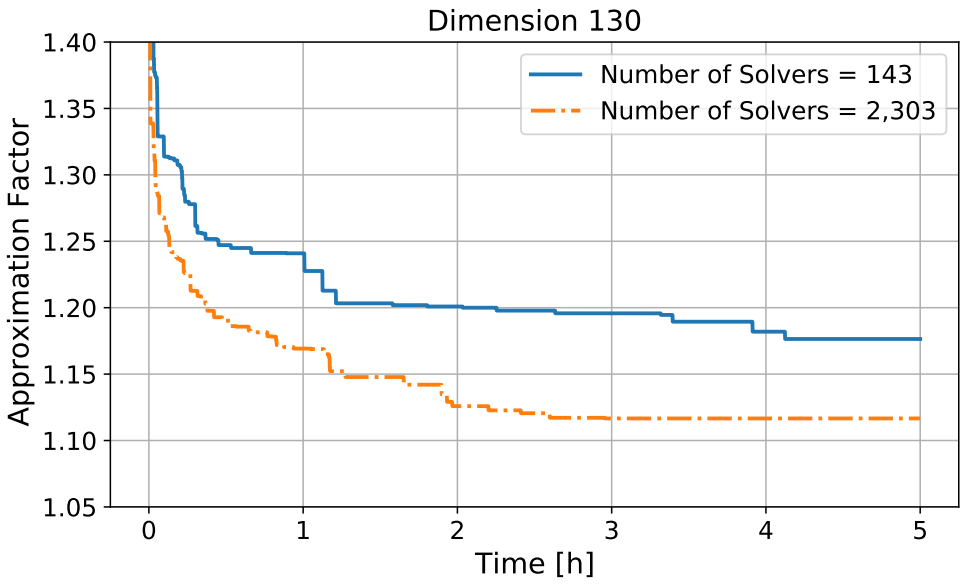

Figure 11: Same as Figure 7, but for different number of `Solver`s; execution were done on the CAL A and CAL B with 144 cores, and ITO with 2,304 cores.

Lisa supercomputer with 103, 680 cores. The first few executions were performed for short periods to test the checkpoint functionality. During the test, we observed occasional aborts due to an excessive number of MPI messages waiting to be received by the `LC`. As a workaround, the Checkpoint Writer (described in Section 5.1.5) was developed, and the upper limit of the size of ISendQueue was set based on the number of messages the `Solver` sends to the `LC` (described in Section 5.2.3). This has improved the stability and enabled a longer execution time. We have tested up to 42 hours of continuous execution. Together with checkpoint and restart, the approximation factor was improved over time.

Figure 13 shows the result of multiple executions of a 130 dimensional SVP. This time, we tested a restart from a checkpoint created on a different environment. The first 14 executions were performed on the Emmy with 12, 288 cores and the last 1 execution was restarted on the Lisa with 103, 680 cores. Although the number of cores used in the Lisa is 8.44 times more than that of the Emmy, the execution was carried over by the checkpoint functionality without any problem. The *Task*s running on the Emmy when the checkpoint was created were executed on the Lisa immediately after the restart, and new *Task*s were generated from the instance pool and assigned to extra `Solver`s available on the Lisa. It should be noted that the approximation factor was improved in the last execution after the final restart (see the purple segment in Figure 13).

The interval of the creation of checkpoint files were set to an hour. It took an average of 1, 531.75 seconds per checkpoint for the Checkpoint Writer to compress and write the pool's information in files, whose size was approximately 7.09 GB on memory. In contrast, it took only an average of 2.77 seconds for `LC` to copy the pools for the Checkpoint Writer. In this manner, the blocking time of `LC`'s message processing was greatly improved by the

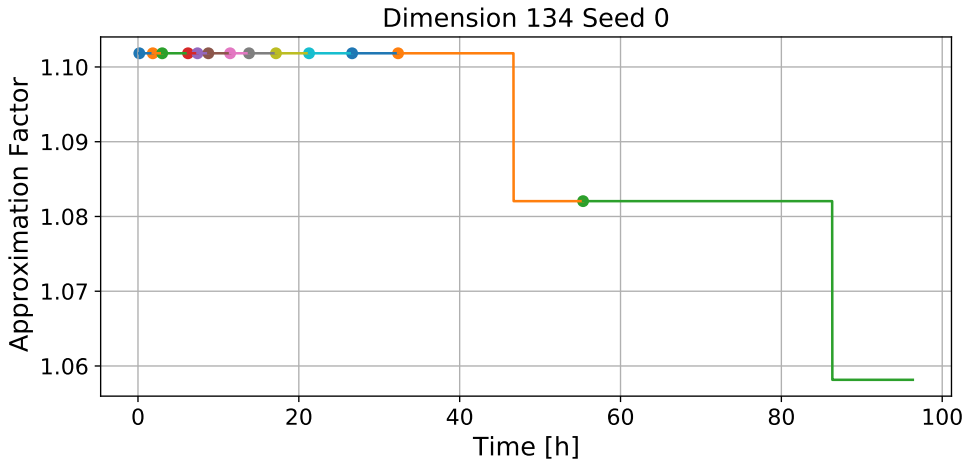

Figure 12: Transition of the approximation factor of a 134-dimensional SVP for long-time execution on the Lisa with 103,680 cores. Each dot represents the beginning of restart from checkpoint.

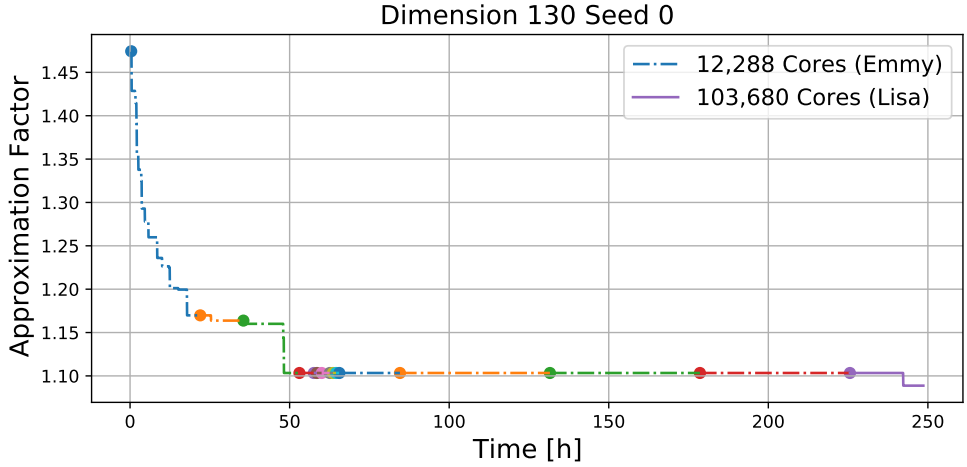

Figure 13: Transition of the approximation factor of a 130-dimensional SVP for long-time execution on the Emmy with 12,280 cores and Lisa with 103,680 cores.

Checkpoint Writer. The averaged time required to read a checkpoint is only 64.75 seconds, and since the checkpoint contains *Task* information, execution can be restarted without prepossessing.

7 Conclusion and Future Work

Lattice problems are a type of discrete optimization problem that is difficult to solve, even for a quantum computer. There is little research on solving this problem in large-scale distributed systems. In addition, the difficulty of solving the lattice problems supports

the security of major cryptographic systems in post-quantum cryptography. Therefore, investigating the potential of large-scale parallel computation of the lattice problems is important in the field of optimization and cryptanalysis.

This paper proposes a novel large-scale framework, CMAP-LAP, for lattice problems. CMAP-LAP offers a multi-algorithm paradigm in which multiple types of lattice algorithms run in parallel while sharing information to improve the performance of the entire system. To realize this paradigm, we have developed four key components. Our communication interface class enables hybrid parallel processing, independent of the solver's internal algorithms. This makes it easy to incorporate existing solvers, those run not only on shared-memory systems but also on distributed-memory systems [20]. The efficient collection and distribution of short lattice vectors by the management process facilitate information exchange among heterogeneous solvers. This is based on the fact that each lattice algorithm generates short lattice vectors as by-products, which can be utilized by other algorithms if shared. Furthermore, the management process generates new tasks from the collected information and assigns them to the solvers in order of the estimated likelihood of finding a solution. The periodic collection of all solvers's progress by the management process allows the grasp of the overall system status. This is used to adjust the assignment of tasks to solvers. In addition, a powerful checkpoint functionality is implemented, which is essential for long execution times. The management of memory and communication delays is carefully realized, which are essential for the stability of large-scale parallel execution. Several numerical experiments demonstrated the stability, scalability, and checkpointing of CMAP-LAP and showed performance improvement through information sharing and heterogeneous execution of multiple algorithms.

CMAP-LAP has the following limitations. 1) In the experiments in this paper, we used simple lattice algorithms such as the naive GaussSieve for testing purposes of the framework. The system can be made more powerful by incorporating state-of-the-art solvers such as G6K. 2) The memory requirements of the management process can be high in massively parallel environments with over a million cores. A distributed management of memory should be developed for further parallelization. 3) The system has been tested only with SVP. It is readily applicable to other lattice problems, and we will evaluate the system for them.

References

[1] Miklós Ajtai. Generating hard instances of lattice problems. In *Symposium on Theory of Computing (STOC 1996)*, pages 99–108. ACM, 1996.

[2] Martin Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In *Advances in Cryptology–EUROCRYPT 2019*, volume 11477 of *Lecture Notes in Computer Science*, pages 717–746. Springer, 2019.

[3] László Babai. On Lovász' lattice reduction and the nearest lattice point problem. *Combinatorica*, 6(1):1–13, 1986.

[4] Michael Burger, Christian Bischof, and Juliane Krämer. p3Enum: A new parameterizable and shared-memory parallelized shortest vector problem solver. In *Computational Science–ICCS 2019*, volume 11540 of *Lecture Notes in Computer Science*, pages 535–542. Springer, 2019.

[5] Jin-Yi Cai. The complexity of some lattice problems. In Wieb Bosma, editor, *Algorithmic Number Theory*, pages 1–32, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[6] Hao Chen. A measure version of Gaussian heuristic. *IACR Cryptology ePrint Archive: Report 2016/439*, 2016.

[7] Öz6Kür Dagdelen and Michael Schneider. Parallel enumeration of shortest lattice vectors. In *Euro-Par 2010–Parallel Processing*, volume 6272 of *Lecture Notes in Computer Science*, pages 211–222. Springer, 2010.

[8] Peter Deutsch and Jean-Loup Gailly. Zlib compressed data format specification version 3.3. Technical report, RFC 1950, May, 1996.

[9] Léo Ducas. Shortest vector from lattice sieving: A few dimensions for free. In *Adavances in Cryptology–EUROCRYPT 2018*, volume 10820 of *Lecture Notes in Computer Science*, pages 125–145. Springer, 2018.

[10] Léo Ducas, Marc Stevens, and Wessel van Woerden. Advanced lattice sieving on GPUs, with tensor cores. *IACR ePrint 2021/141*, 2021.

[11] Koichi Fujii, Naoki Ito, Sunyoung Kim, Masakazu Kojima, Yuji Shinano, and Kim-Chuan Toh. Solving challenging large scale qaps. Technical Report 21-02, ZIB, Takustr. 7, 14195 Berlin, 2021.

[12] Nicolas Gama, Phong Q Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In *Advances in Cryptology–EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 257–278. Springer, 2010.

[13] Gerald Gamrath, Thorsten Koch, Stephen Maher, Daniel Rehfeldt, and Yuji Shinano. SCIP-Jack—a solver for STP and variants with parallelization extensions. *Mathematical Programming Computation*, 9(2):231–296, 2017.

[14] Jens Hermans, Michael Schneider, Johannes Buchmann, Frederik Vercauteren, and Bart Preneel. Parallel shortest lattice vector enumeration on graphics cards. In *Progress in Cryptology–AFRICACRYPT 2010*, volume 6055 of *Lecture Notes in Computer Science*, pages 52–68. Springer, 2010.

[15] Antoine Joux. A tutorial on high performance computing applied to cryptanalysis (invited talk). In *Advances in Cryptology–EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 1–7. Springer, 2012.

[16] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of operations research*, 12(3):415–440, 1987.

[17] Po-Chun Kuo, Michael Schneider, Özgür Dagdelen, Jan Reichelt, Johannes Buchmann, Chen-Mou Cheng, and Bo-Yin Yang. Extreme enumeration on GPU and in clouds. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, volume 6917 of *Lecture Notes in Computer Science*, pages 176–191. Springer, 2011.

[18] Arjen Klaas Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[19] Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In *Symposium on Discrete Algorithms (SODA 2010)*, pages 1468–1480. ACM-SIAM, 2010.

[20] Lluís-Miquel Munguía, Geoffrey Oxberry, Deepak Rajan, and Yuji Shinano. Parallel pips-sbb: multi-level parallelism for stochastic mixed-integer programs. *Computational Optimization and Applications*, 73(2):575–601, Jun 2019.

[21] Phong Q Nguyen. Hermite's constant and lattice algorithms. In *The LLL Algorithm*, pages 19–69. Springer, 2009.

[22] Chris Peikert. A decade of lattice cryptography. *Foundations and Trends in Theoretical Computer Science*, 10(4):283–424, 2016.

[23] Ted K. Ralphs, Yuji Shinano, Timo Berthold, and Thorsten Koch. Parallel solvers for mixed integer linear optimization. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 283–336. Springer International Publishing, 2018.

[24] Daniel Rehfeldt, Yuji Shinano, and Thorsten Koch. Scip-jack: An exact high performance solver for steiner tree problems in graphs and related problems. In Hans Georg Bock, Willi Jäger, Ekaterina Kostina, and Hoang Xuan Phu, editors, *Modeling, Simulation and Optimization of Complex Processes HPSC 2018*, pages 201–223, Cham, 2021. Springer International Publishing.

[25] Michael Schneider, Nicolas Gama, P Baumann, and L Nobach. SVP challenge (2010). *URL: http://latticechallenge.org/svp-challenge*.

[26] Claus Peter Schnorr. Lattice reduction by random sampling and birthday methods. In *Symposium on Theoretical Aspects of Computer Science (STACS 2003)*, volume 2607 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2003.

[27] Claus-Peter Schnorr and Martin Euchner. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical programming*, 66:181–199, 1994.

[28] SCIP Optimization Suite. `https://scipopt.org/#scipoptsuite`.

[29] Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, and Thorsten Koch. ParaSCIP – a parallel extension of SCIP. In Christian Bischof, Heinz-Gerd Hegering, Wolfgang E. Nagel, and Gabriel Wittum, editors, *Competence in High Performance Computing 2010*, pages 135–148. Springer, 2012.

[30] Yuji Shinano, Tobias Achterberg, Timo Berthold, Stefan Heinz, Thorsten Koch, and Michael 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)*, pages 770–779, Los Alamitos, CA, USA, 2016. IEEE Computer Society.

[31] Yuji Shinano, Timo Berthold, and Stefan Heinz. Paraxpress: an experimental extension of the fico xpress-optimizer to solve hard mips on supercomputers. *Optimization Methods and Software*, 33(3):530–539, 2018.

[32] Yuji Shinano, Stefan Heinz, Stefan Vigerske, and Michael Winkler. Fiberscip—a shared memory parallelization of scip. *INFORMS Journal on Computing*, 30(1):11–30, 2018.

[33] Yuji Shinano, Daniel Rehfeldt, and Tristan Gally. An easy way to build parallel state-of-the-art combinatorial optimization problem solvers: A computational study on solving steiner tree problems and mixed integer semidefinite programs by using ug[scip-*,*]-libraries. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 530–541, 2019.

[34] Yuji Shinano, Daniel Rehfeldt, and Thorsten 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*, volume 11494, pages 529–539, 2019.

[35] Nariaki Tateiwa, Yuji Shinano, Satoshi Nakamura, Akihiro Yoshida, Shizuo Kaji, Masaya Yasuda, and Katsuki Fujisawa. Massive parallelization for finding shortest lattice vectors based on ubiquity generator framework. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2020.

[36] Tadanori Teruya, Kenji Kashiwabara, and Goichiro Hanaoka. Fast lattice basis reduction suitable for massive parallelization and its application to the shortest vector problem. In *Public Key Cryptography (PKC 2018)*, volume 10769 of *Lecture Notes in Computer Science*, pages 437–460. Springer, 2018.

[37] UG: Ubiquity Generator framework. `http://ug.zib.de/`.

[38] Junpei Yamaguchi and Masaya Yasuda. Explicit formula for Gram-Schmidt vectors in LLL with deep insertions and its applications. In *Number-Theoretic Methods in Cryptology (NuTMiC 2017)*, volume 10737 of *Lecture Notes in Computer Science*, pages 142–160. Springer, 2017.

[39] Masaya Yasuda. A survey of solving SVP algorithms and recent strategies for solving the SVP challenge. In *International Symposium on Mathematics, Quantum Theory, and Cryptography*, pages 189–207. Springer, 2021.

[40] Masaya Yasuda, Satoshi Nakamura, and Junpei Yamaguchi. Analysis of DeepBKZ reduction for finding short lattice vectors. *Designs, Codes and Cryptography*, 88:2077–2100, 2020.