# CMAP-LAP: Configurable Massively Parallel Solver for Lattice Problems

Tateiwa, Nariaki
Graduate School of Mathematics, Kyushu University

Shinano, Yuji
Applied Algorithmic Intelligence Methods (A$^2$IM), Zuse Institute Berlin (ZIB)

Yamamura, Keiichiro
Graduate School of Mathematics, Kyushu University

Yoshida, Akihiro
Graduate School of Mathematics, Kyushu University

他

# CMAP-LAP: Configurable Massively Parallel Solver for Lattice Problems

1st Nariaki Tateiwa
*Graduate School of Mathematics*
*Kyushu University*
Fukuoka, Japan
ORCID: 0000-0001-7161-6687

2nd Yuji Shinano
*Applied Algorithmic Intelligence Methods (A²IM)*
*Zuse Institute Berlin (ZIB)*
Berlin, Germany
ORCID: 0000-0002-2902-882X

3rd Keiichiro Yamamura
*Graduate School of Mathematics*
*Kyushu University*
Fukuoka, Japan
ORCID: 0000-0003-4696-2881

4th Akihiro Yoshida
*Graduate School of Mathematics*
*Kyushu University*
Fukuoka, Japan
ORCID: 0000-0002-7856-6536

5th Shizuo Kaji
*Institute of Mathematics for Industry*
*Kyushu University*
Fukuoka, Japan
ORCID: 0000-0002-7856-6536

6th Masaya Yasuda
*Department of Mathematics*
*Rikkyo University*
Tokyo, Japan
ORCID: 0000-0002-1534-5648

7th Katsuki Fujisawa
*Institute of Mathematics for Industry*
*Kyushu University*
Fukuoka, Japan
ORCID: 0000-0001-8549-641X

***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. CMAP-LAP facilitates the implementation of large-scale parallel strategies for lattice problems since all the functions are designed to be customizable and abstract. 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.**

***Index Terms*—Discrete optimization; Lattice problem; Lattice-based cryptography; Shortest vector problem; Parallel algorithms; Ubiquity Generator Framework**

## I. 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 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 [1] and have been used to construct various cryptosystems [2], including post-quantum cryptography. Therefore, developing a framework for lattice problems is an important task in both large-scale optimization and cryptanalysis (see [3] for cryptanalysis using high-performance computing). More specifically, the security of many cryptosystems is based on the hardness of an approximate variant of the SVP.

Three basic families of lattice algorithms have been developed to solve practical lattice problems: basis reduction, enumeration (ENUM), and sieve. 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 [4] 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. In contrast, MAP-SVP [5] 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 to 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 that it provides solutions to overcome these obstacles and develops a flexible framework to make various algorithms work cooperatively on a large-scale distributed computing platform. By exploiting the mathematical properties of the 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 [6] 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. The original UG framework has been successfully utilized for mixed-integer linear programming problems [7]–[10], Steiner tree problems [11]–[14], and quadratic assignment problems [15] 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 into the *Generalized Ubiquity Generator framework* (Generalized UG; UG version 1.0 RC)[1], which allows more flexibility necessary for lattice algorithms. Particular emphasis is put on the efficient and versatile message-sharing mechanics. Based on the Generalized UG, we developed the *Configurable Massively Parallel Solver for Lattice Problems (CMAP-LAP)*. This is the framework for massively parallel strategies for lattice problems. It is designed to facilitate the implementation of new parallel strategy ideas based on this framework. In this paper, we evaluate the performance of CMAP-LAP using the naive algorithm for SVP.

Main contributions of this study are summarized below:

- 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 V, and detail for solving SVP is in Section VI-A). To realize the scheme, CMAP-LAP was 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 VI).
- Each process asynchronously performs various lattice algorithms in coordination while sharing information. Processes for different algorithms are adaptively allocated, and 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 drastically improved the stability and scalability (see Sections V-A4 and V-B3).
- The high-level checkpoint-and-restart functionality is implemented to resume the execution even on different architectures and platforms of various sizes (see Section V-A5).
- 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 V-B1).

[1]The Generalized UG code will be released within the SCIP Optimization Suite [16] in 2021.

## II. Lattice Problems

A *lattice* of dimension $n$ is the set of integral linear combinations of $n$ linearly independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{R}^n$;

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

The symbol $\mathcal{L}(\mathbf{B})$ denotes the lattice spanned by the rows of an invertible matrix $\mathbf{B}$. The matrix 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 $\mathbf{C} = \mathbf{TB}$. Given a basis matrix $\mathbf{B}$ of $L$, the volume of $L$ is defined as $\mathrm{vol}(L) := |\det(\mathbf{B})|$, independent of the choice of basis matrices.

*Lattice problems* are algorithmic problems for lattices. Among of them, the following is of fundamental importance:

*Definition 1 (Shortest Vector Problem, SVP):* Find the shortest non-zero vector with respect to the $\ell_2$-norm in the lattice $\mathcal{L}(\mathbf{B})$, given a basis matrix $\mathbf{B}$.

SVP is a discrete optimization problem of finding the best combination of integers $x_i$'s in (1) such that $\mathbf{v} = \sum_{i=1}^{n} x_i \mathbf{b}_i$ is nonzero and the shortest in $L$, and it is NP-hard under randomized reductions [17]. (That is, there exists a probabilistic Turing-machine that reduces any problem in NP to SVP instances in polynomial-time.) The length of the shortest non-zero vector in $L$ is denoted by $\lambda_1(L)$. SVP is the problem of finding $\mathbf{s} \in L$ with $\|\mathbf{s}\| = \lambda_1(L)$. It should be emphasized that there is no known polynomial-time algorithm to check if $\|\mathbf{v}\| = \lambda_1(L)$ given $\mathbf{v} \in L$. Therefore we rely on the *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 $\mathbb{R}^n$, it leads to $\lambda_1(L) \approx \left(\frac{\mathrm{vol}(L)}{\omega_n}\right)^{1/n}$, where $\omega_n$ 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

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

Then, $\lambda_1(L) \approx \mathrm{GH}(L)$ holds for random lattices $L$ in high dimensions $n \geq 40$. (Unfortunately, the 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 below:

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

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 [2]). Most lattice problems can be reduced to SVP or the Closest Vector Problem (CVP), and hence, SVP and CVP are fundamental. As Kannan's embedding [18] 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.

## III. Related Work

We summarize the existing solvers with a particular emphasis on SVP. The Darmstadt SVP challenge [19] has been an established venue for assessing SVP algorithms. Lattice bases for dimensions $40 \leq n \leq 200$ are publicly available. More precisely, for each dimension with so-called the *seed*, a lattice basis is generated. For each generated lattice $L$, any non-zero lattice vector with length less than $1.05\mathrm{GH}(L)$ is considered as a solution. 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.

### A. Approximate-SVP solvers

We present recent works for solving the SVP challenge in high dimensions $n \geq 150$. Note that the approximation factors of most of the current records for $n \geq 150$ are over 1.02, and thus, they are not likely to be the shortest. In 2017, an SVP instance in $n = 150$ was first solved with an approximation factor 1.04192. It was reported in [20] that it took 394 days using up to 864 cores. The work is based on the random sampling [21], which samples small $x_i$'s in (1) until a short vector is found. In 2018, a number of records in $n \leq 155$ were updated using the general sieve kernel, called G6K [4]. G6K supports a variety of lattice basis reductions and sieve algorithms (Section IV). Most of the records for $n \geq 130$ and notably the current highest dimension record ($n = 180$) have been found using G6K. It was reported in [22] that the $n = 180$ record took 51.6 days on a single machine with 4 NVIDIA Turing GPUs, and its approximation factor was 1.04002.

### B. Exact-SVP solvers

We present several works solving exact-SVP based on ENUM (Section IV). ENUM is asymptotically slower than sieve, but 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 [23]–[25]. Another approach has been pursued by randomization. Applying unimodular transformation to an input basis 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 [26]. It reported the running time of solving exact-SVP over 60 cores for dimensions $n \leq 100$. In 2020, a massive parallel exact-SVP solver, called MAP-SVP, was developed in [5] using the Ubiquity Generator framework. It was a distributed asynchronous cooperative solver based on randomization and ENUM with pruning techniques. MAP-SVP found solutions for many instances of the SVP challenge in $n \leq 127$. In particular, it took 147 hours to find a solution in $n = 127$ using $100,032$ cores. The approximation factor of the solution is $0.97573$, the smallest among the current records for $n \geq 120$.

## IV. Lattice Algorithms

We summarize practical algorithms solving lattice problems, mainly SVP (see [27], [28] 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 orthogonal vectors $\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*$ defined recursively by

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

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

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

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

### A. 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 $\mathbf{s} = v_1\mathbf{b}_1 + \cdots + v_n\mathbf{b}_n$ is the shortest in $L \setminus \{\mathbf{0}\}$. With the 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 squared length $\sum_{j=k}^n \left( v_j + \sum_{i=j+1}^n \mu_{ij} v_i \right)^2 \|\mathbf{b}_j^*\|^2$ for each $1 \leq k \leq n$. Given a search radius $R > 0$, an enumeration tree of depth $n$ is constructed, whose nodes at depth $n - k + 1$ correspond to the set of all vectors in $\pi_k(L)$ of a maximum length of $R$. The key observation is that if a shortest vector satisfies $\|\mathbf{s}\| \leq R$, its projections also satisfy $\|\pi_k(\mathbf{s})\|^2 \leq R^2$ for all $k$. Therefore, it is crucial to choose a good $R$ that is sufficiently small but larger than the shortest norm. One useful strategy is pruning [29] where a smaller tree is built by replacing the inequalities $\|\pi_k(\mathbf{s})\|^2 \leq R^2$ with $\|\pi_k(\mathbf{s})\|^2 \leq R_{n+1-k}^2$ with shorter radii $R_1 \leq \cdots \leq R_n = R$ at each depth determined by a pruning strategy. This method is probabilistic because it is not certain that s can be found in this pruned tree. Another strategy is parallelization. We start with a sufficiently large $R$. When one instance finds a short vector, its norm $R'$ is shared across all instances. Then we can replace $R$ with $R'$ to reduce the size of the enumeration tree. To boost reduction of the search radius, we introduce the novel *sub-ENUM* algorithm, which produces many short lattice vectors using similar idea as sub-sieving [30]. ENUM is performed in a projected lattice $\pi_k(L)$ to obtain short lattice vectors $\mathbf{v} \in L$ such that $\|\pi_k(\mathbf{v})\| \leq \tau \cdot \mathrm{GH}(\pi_k(L))$, where $\tau$ is a constant. We typically chose $\tau = \sqrt{\frac{4}{3}}$ in our implementation. Then, ENUM is again used to find a shortest vector for the $k$-dimensional lattice spanned by $\{\mathbf{b}_1, \ldots, \mathbf{b}_{k-1}, \mathbf{v}\}$.

### B. Sieve

Given a lattice $L$ of dimension $n$, sieve is a probabilistic algorithm that solves SVP exactly with a time complexity of $2^{O(n)}$, 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 radius $R$ with $\lambda_1(L) \leq R \leq O(\lambda_1(L))$. 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 $\cup_{k=1}^{n} (\pi_k(L) \cap S)$ with a total number of $2^{O(n^2)}$. 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 is 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, the shortest vector $\mathbf{s}$ can be found by computing the differences of pairs in $M$. There are various implementations of sieve algorithms that differ mainly in how to sample $M$, such as GaussSieve [31]. Similar to ENUM, the choice of $R$ is crucial to the sieve.

### C. Basis reduction

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*). The well-known reduction algorithms are LLL [32], BKZ [33], and their generalizations, DeepLLL and DeepBKZ [34]. LLL is constructed using basic row-wise matrix transformations, and $\beta$-BKZ is constructed with LLL and SVPs on the $\beta$-dimensional projected lattice spanned by $\{\pi_i(\mathbf{b}_i), \pi_i(\mathbf{b}_{i+1}), \ldots, \pi_i(\mathbf{b}_{i+\beta-1})\}$. 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, basis reduction is also performed as a preprocessing 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.

## V. Design of CMAP-LAP

It is essential for a practical solver to utilize the multiple lattice algorithms introduced in Section IV. Most of the existing

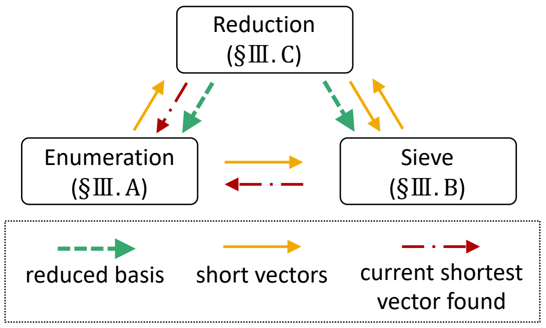

Fig. 1. Interaction among SVP algorithms with a synergy: Basis reduction generates a reduced basis, over which enumeration and sieve can find short vectors efficiently. In contrast, enumeration and sieve find short vectors so that basis reduction accelerates to find a more reduced basis.

solvers discussed in Section III 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 IV 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 Fig. 1). 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.

### A. Architecture of CMAP-LAP

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. The LC has the following data pools: 1) Instance Pool, 2) Solver Pool, 3) Task Pool, and 4) Share-Data Pool. (See Fig. 2). 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.

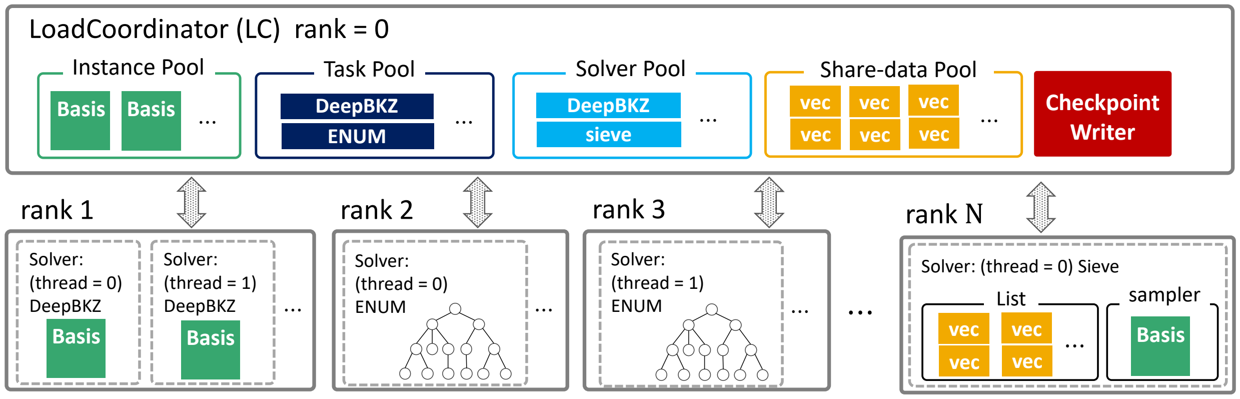
LoadCoordinator (LC) rank = 0
Instance Pool
Basis
Basis
Task Pool
DeepBKZ
ENUM
Solver Pool
DeepBKZ
sieve
Share-data Pool
vec
Checkpoint Writer
rank 1
rank 2
rank 3
rank N
Solver: (thread = 0) DeepBKZ
Solver: (thread = 1) DeepBKZ
Solver: (thread = 0) ENUM
Solver: (thread = 1) ENUM
Solver: (thread = 0) Sieve
List
sampler

Fig. 2. System overview of CMAP-LAP for SVP

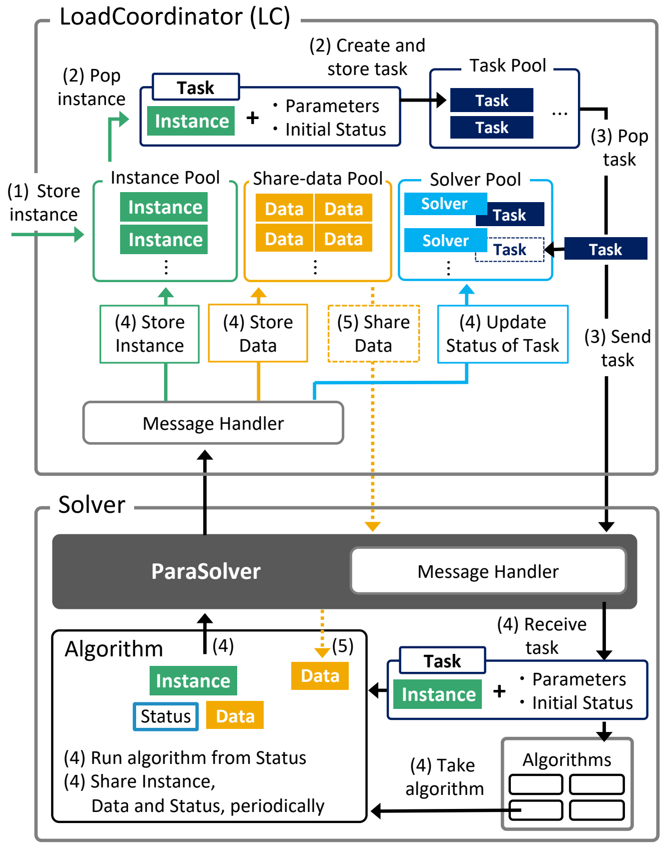

Fig. 3. Execution flow of CMAP-LAP

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 Fig. 3):

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 Fig. 1. LC stores information in the pool according to this type. In addition, Solver sends *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.
6) When a Solver finishes the assigned *Task*, it sends its final *Status* to LC and becomes idle.

LC always checks for messages from Solver. Messages received by the LC are processed 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.

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

*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 IV-A 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.

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

*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.

*4) Share-Data Pool:* Share-data pool stores information that is shared across multiple Solvers. In the case of CMAP-LAP, a typical type of information sent from Solvers is the lattice vector of a small norm. LC checks if the sent vector is already in the pool. If it is not in the pool, an entry (Data, $S$, priority) is created in the pool, where Data is the sent vector. $S$ 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 the Solvers whose Solver-Ids are not in $S$ and adds their Solver-Id to $S$. In this way, information is shared among all Solvers efficiently while controlling the total amount of communication. The interval at which Solvers and LC push information can be tuned depending on the configuration of the machine.

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 $S$ is dominant and should be carefully estimated in case of massive parallelization. Moreover, the cost of Data retrieval increases when the pool size and the number of Solvers are large. In this case, the limit of the pool size and the frequency of data sharing are suppressed.

*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 entire system. Lattice problems are hard and often require millions of core hours. Thus, it is critical to record the progress and resume after an 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 saved in checkpoint files using zlib [35], 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 Solvers 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 Solvers 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 Solvers are available.

The technically important point is that the message processing from Solvers 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.

## B. Implementation Technicalities

*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 Fig. 3).

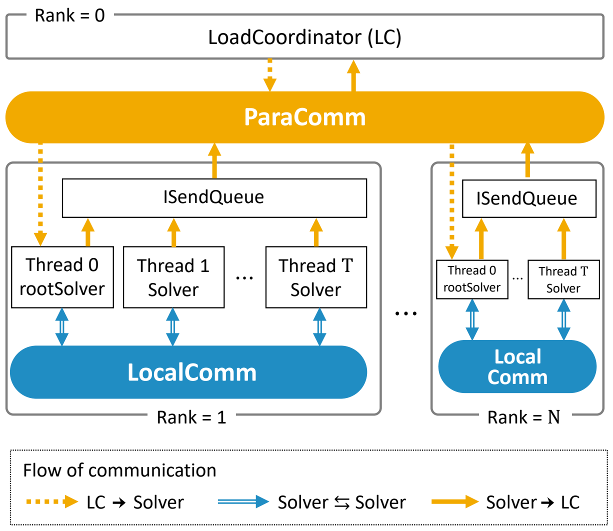

Fig. 4. Communicators between and within MPI processes: ParaComm and LocalComm.

*2) Hybrid Parallelization:* CMAP-LAP uses hybrid parallelization, which combines MPI with C++11 thread communication. LC and Solver have two kinds of communicators:

one is *ParaComm*, which wraps MPI 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. 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.

*3) MPI_ISend Communication:* 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 receipt of messages. Therefore, 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. By examining the size of each ISendQueue, we can determine the number of unreceived LC messages. Therefore, we set an upper limit on the size of ISendQueue and the messages exceeding the limit are destroyed instead of being sent, thereby preventing many messages from accumulating in LC.

## VI. Numerical Experiments

In this section, we evaluate the performance of CMAP-LAP on 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 I.

### A. 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. Then, LC generates DeepBKZ *Task*s for the bases in the instance pool. The reduced bases are sent from Solvers 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

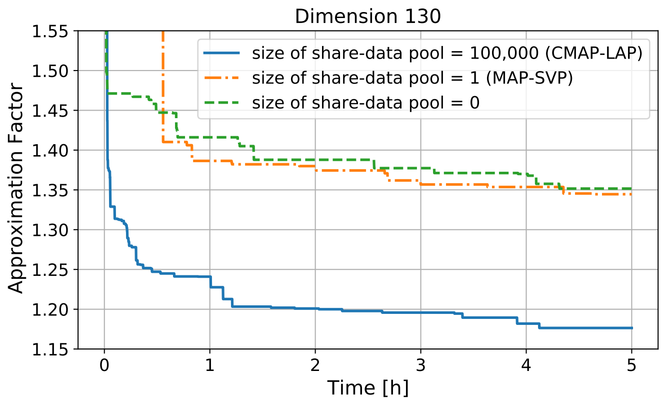

Fig. 5. Transition of the approximation factors for different share-data pool sizes; execution were done on the CAL A and CAL B with 144 cores. The solid blue lines in Fig. 5, 6 and 8 represent the same experimental results.

sent from Solvers to LC, which are inserted into the share-data pool. At regular intervals, Solvers request LC to send short vectors from the share-data pool. Each algorithm has no constraint on the timing of the communication and can reflect it in its own algorithm anytime. Therefore, if there is a delay in communication, there is no need to stop the algorithm and wait for the share-data. 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. We calculated 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.

Because 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 IV. 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 number of lattice vectors having smaller approximation factors decreases 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 $1.20$ compared with $1.30$.

### B. 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, Solvers share multiple short lat-

TABLE I
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) |

*Operating systems and versions*: Lisa and Emmy [CentOS Linux release 7.7.1908], ITO [Red Hat Enterprise Linux Server release 7.3.1611], CAL A and CAL B [CentOS Linux release 7.9.2009]. *Compilers and versions*: Lisa and Emmy [intel19.0.5, impi2019.5], ITO [icc 19.1.1.217, openmpi3.1.3], CAL A, CAL B [icc 19.1.3.304, openmpi4.0.5]. *Libraries and versions*: NTL v11.3.3, Eigen v3.3.7, gsl v2.6, scipoptsuite v6.0.2, OpenBLAS, fplll v5.2.1

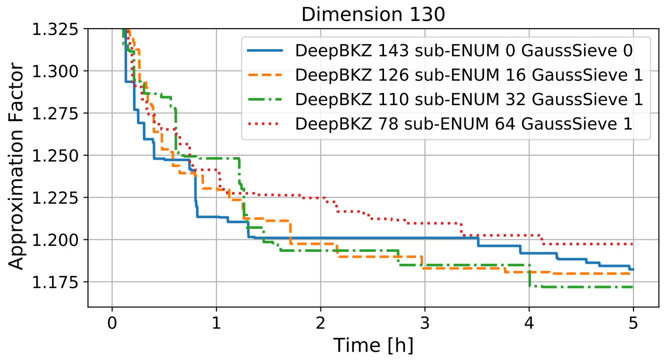

Fig. 6. Same as Fig. 5, but for different allotment of algorithms; execution were done on the CAL A and CAL B with 144 cores.

tice vectors via the share-data pool in LC. The amount of information shared among Solvers can be controlled by the size of the share-data pool. Fig. 5 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 Solvers 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 Solvers. 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.

## C. 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 Solvers 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*. Fig. 6 shows the results for a 130-dimensional SVP with four different configurations of the *Task* assignment. We ran the experiment on the CAL 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-

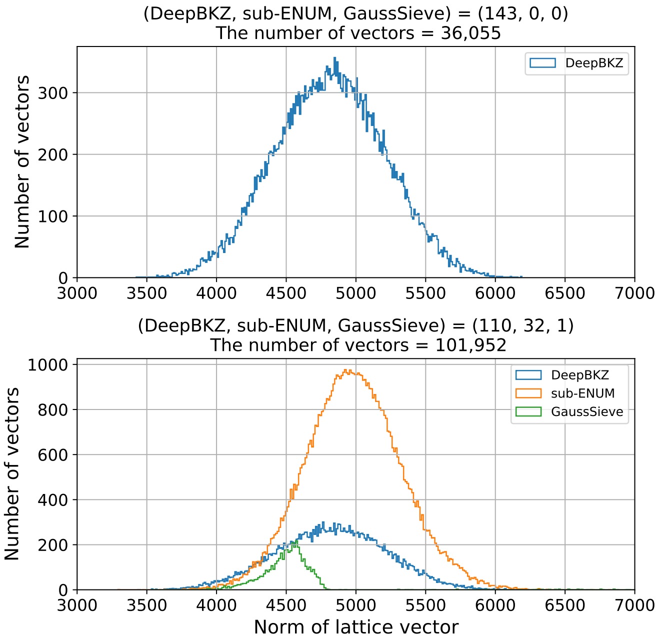

Fig. 7. Distribution of the norm of vectors in the share-data pool.

ENUM, GaussSieve) = $(110, 32, 1)$. To investigate the reason, we examine the distribution of vector norms in the share-data pool for two configurations (see Fig. 7). 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.

## D. Scalability

In addition, we evaluated the effect of parallelization on the transition of the approximation factor (see Fig. 8). We experimented with the same SVP instances as in Section VI-B 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 $576$ and $2,304$ cores for this experiment. The best (minimum) approximation factor obtained within $5$ hours was $1.176$, $1.133$ and $1.117$

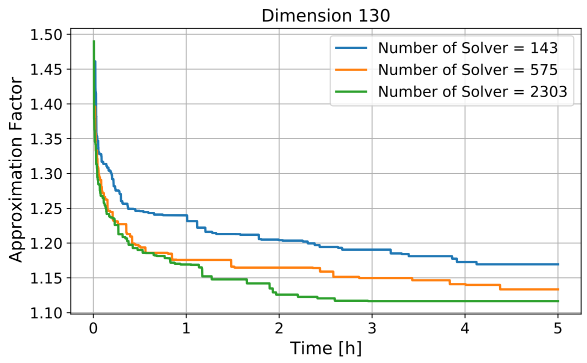

Fig. 8. Same as Fig. 5, but for different number of Solvers; execution were done on the CAL A and CAL B with 144 cores, and ITO with 576 and 2,304 cores.

with 143, 575 and 2,303 Solvers, respectively. In terms of Gaussian Heuristics, the 2,303 Solvers execution is considered to be $1.176^{130}/1.117^{130} \approx 800$ times better than 143 Solvers execution. 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.

In these experiments, the average ratio of idle time to the total execution time of the Solver processes was 0.00314, 0.00342 and 0.00359 for the number of Solvers is 143, 575 and 2,303, respectively. The idle time included the waiting time for the communication of the vector, *Status*, and *Task* with LC. This indicates the high CPU utilization. We also measured the idle time of LC . It should be noted that certain amount of idle time is desirable for LC so that more messages from Solver can be handled with no delay. The average ratio of the idle time to the total execution time of the LC process in these experiments was 0.9463, 0.9533 and 0.9122 for the number of Solver 143, 575 and 2,303, respectively, indicating that the LC process is also highly efficient.

### E. *Stability with massive parallelization*

We show the results of a long-time execution of CMAP-LAP. Fig. 9 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 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 V-A5) 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 V-B3). This has improved the stability and enabled a longer execution time. We have tested up to 42 hours of continuous execution.

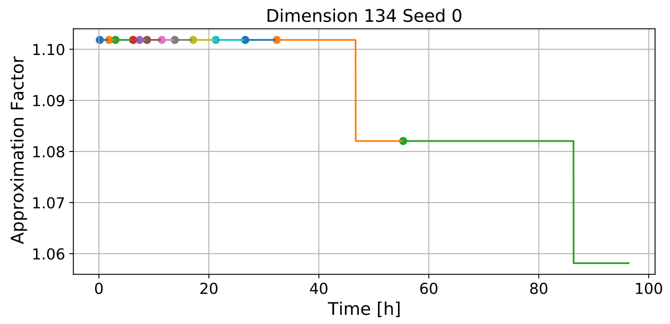

Fig. 9. Transition of the approximation factor of a 134-dimensional SVP for long-time execution on the Lisa with 103,680 cores. Each dot represents the beginning of restart from checkpoint.

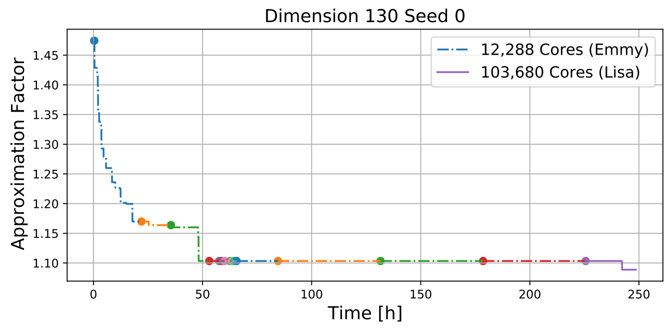

Fig. 10. Transition of the approximation factor of a 130-dimensional SVP for long-time execution on the Emmy with 12,280 cores and Lisa with 103,680 cores.

Together with checkpoint and restart, the approximation factor was improved over time.

Fig. 10 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 Solvers 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 Fig. 10).

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 with a size of 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 Checkpoint Writer.

## VII. Conclusion and Future Work

This paper proposes a novel large-scale framework, CMAP-LAP, for lattice problems. Lattice problems are a type of discrete optimization problem that is difficult to solve, even for a quantum computer. 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 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 [36]. 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 the order of the estimated likelihood of finding a solution. 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 is 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 conducted in this study, 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. Thus, 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 intend to evaluate the system on them.

## 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 Number JP20H04142, Japan, the Research Campus MODAL funded by the German Federal Ministry of Education and Research (fund number 05M14ZAM). This work was also supported the National High Performance Computing Center at the Zuse Institute Berlin (NHR@ZIB). We are grateful to the supercomputer staff, especially Matthias Läuter and Tobias Watermann.

## References

[1] J.-Y. Cai, "The complexity of some lattice problems," in *Algorithmic Number Theory*, W. Bosma, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1–32.

[2] C. Peikert, "A decade of lattice cryptography," *Foundations and Trends in Theoretical Computer Science*, vol. 10, no. 4, pp. 283–424, 2016. [Online]. Available: http://dx.doi.org/10.1561/0400000074

[3] 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.

[4] 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.

[5] N. Tateiwa, Y. Shinano, S. Nakamura, A. Yoshida, S. Kaji, M. Yasuda, and K. 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*. IEEE, 2020, pp. 1–15.

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

[7] Y. Shinano, T. Achterberg, T. Berthold, S. Heinz, and T. Koch, "ParaSCIP – a parallel extension of SCIP," in *Competence in High Performance Computing 2010*, C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, Eds. Springer, 2012, pp. 135–148.

[8] 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

[9] 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.

[10] Y. Shinano, T. Berthold, and S. Heinz, "Paraxpress: an experimental extension of the fico xpress-optimizer to solve hard mips on supercomputers," *Optimization Methods and Software*, vol. 33, no. 3, pp. 530–539, 2018. [Online]. Available: https://doi.org/10.1080/10556788.2018.1428602

[11] 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.

[12] 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.

[13] Y. Shinano, D. Rehfeldt, and T. 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)*, 2019, pp. 530–541.

[14] D. Rehfeldt, Y. Shinano, and T. Koch, "Scip-jack: An exact high performance solver for steiner tree problems in graphs and related problems," in *Modeling, Simulation and Optimization of Complex Processes HPSC 2018*, H. G. Bock, W. Jäger, E. Kostina, and H. X. Phu, Eds. Cham: Springer International Publishing, 2021, pp. 201–223.

[15] K. Fujii, N. Ito, S. Kim, M. Kojima, Y. Shinano, and K.-C. Toh, "Solving challenging large scale qaps," ZIB, Takustr. 7, 14195 Berlin, Tech. Rep. 21-02, 2021.

[16] "SCIP Optimization Suite," https://scipopt.org/#scipoptsuite.

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

[18] R. Kannan, "Minkowski's convex body theorem and integer programming," *Mathematics of operations research*, vol. 12, no. 3, pp. 415–440, 1987.

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

[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] C. P. Schnorr, "Lattice reduction by random sampling and birthday methods," in *Symposium on Theoretical Aspects of Computer Science (STACS 2003)*, ser. Lecture Notes in Computer Science, vol. 2607. Springer, 2003, pp. 145–156.

[22] L. Ducas, M. Stevens, and W. van Woerden, "Advanced lattice sieving on gpus, with tensor cores," *IACR ePrint 2021/141*, 2021.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] M. Yasuda, "A survey of solving SVP algorithms and recent strategies for solving the SVP challenge," in *International Symposium on Mathematics, Quantum Theory, and Cryptography*. Springer, 2021, pp. 189–207.

[29] 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.

[30] 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.

[31] D. Micciancio and P. Voulgaris, "Faster exponential time algorithms for the shortest vector problem," in *Symposium on Discrete Algorithms (SODA 2010)*. ACM-SIAM, 2010, pp. 1468–1480.

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

[33] 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.

[34] 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.

[35] P. Deutsch and J.-L. Gailly, "Zlib compressed data format specification version 3.3," RFC 1950, May, Tech. Rep., 1996.

[36] L.-M. Munguía, G. Oxberry, D. Rajan, and Y. Shinano, "Parallel pips-sbb: multi-level parallelism for stochastic mixed-integer programs," *Computational Optimization and Applications*, vol. 73, no. 2, pp. 575–601, Jun 2019. [Online]. Available: https://doi.org/10.1007/s10589-019-00074-0