# Scaling Lattice Sieves across Multiple Machines

Martin R. Albrecht[1] and Joe Rowell[2]

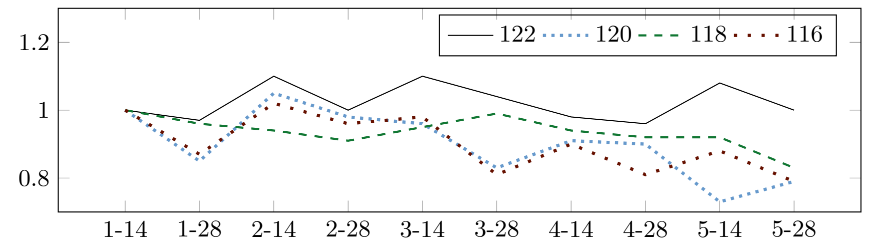
[1] King's College London and SandboxAQ, London, UK,
martin.albrecht@kcl.ac.uk,sandboxaq.com
[2] Royal Holloway, University of London, Egham, UK
joe.rowell.2015@live.rhul.ac.uk

**Abstract.** Lattice sieves are algorithms for finding short vectors in lattices. We present an implementation of two such sieves – known as "BGJ1" and "BDGL" in the literature – that scales across multiple servers (with varying success). This class of algorithms requires exponential memory which had put into question their ability to scale across sieving nodes. We discuss our architecture and optimisations and report experimental evidence of the efficiency of our approach.

## 1 Introduction

A central hard problem in post-quantum cryptography is the *Shortest Vector Problem* (SVP) on lattices: given a basis $\mathbf{B}$ of a lattice $\mathcal{L}$ find a shortest nonzero vector in $\mathcal{L}$. SVP is known to be NP-hard under randomised reductions [Ajt98], with hardness results extending up to subpolynomial approximation factors [Mic01, Kho05, HR12, Mic12]. It is generally assumed that the difficulty of SVP degrades gracefully as the approximation factor increases. Moreover, it is generally assumed that there is no probabilistic polynomial-time or even bounded-error quantum polynomial-time algorithm that solves SVP to within polynomial approximation factors. This hardness (coupled with compact and easy-to-implement constructions) has led to many cryptographic primitives basing their security on the hardness of variants of SVP, and as a result many algorithms have been considered and proposed for solving (approximate) SVP [Kan83, FP83, AKS01, NV08, GNR10, MV10a, MW15, Laa15, BDGL16, Duc18a, ADH+19, ABF+20, ABLR21], with the fastest known family of algorithms being *lattice sieves*.

Lattice sieves come in both provable [AKS01, NV08, MV10b, ADRS15] and heuristic variants [NV08, MV10b, BGJ15, BDGL16, HK17]. These variants exhibit time and memory complexity of $2^{\Theta(n)}$ and implementations of heuristic sieves currently dominate the Darmstadt SVP Hall of



Y axis shows (wall time/cores)/(wall time$_{(1\text{-}14)}$/14). A factor of 1.0 is ideal scaling. Labels on the X axis: (#nodes)-(#cores per node). All nodes were of type K in Table 1. One experiment per dimension and configuration.

Fig. 1: Parallel performance for BGJ1 sieving in dimensions 116 to 122.

Fame [LR24].[3] Briefly, lattice sieves operate by first sampling an exponentially large list of vectors $L$ and then iteratively reducing this list by computing the pairwise sums and differences of vectors from this list, keeping those that have smaller norms. For heuristic sieving algorithms – which we focus on in this work – this list typically has $2^{0.210n+o(n)}$ many vectors.

While sieving outperforms alternative approaches to finding short vectors in lattices both in practice and asymptotically, it is an open question of how it scales on realistic hardware. The central challenges here are the aforementioned exponentially-large lists. That is, while the benefits of sieving in a single machine context are well-understood [ADH+19, DSv21], these benefits may be somewhat limited to when access to the list is fast – say, when the entire list is in system memory. Indeed, some have argued [Ber16, BBC+20] that the significantly lower memory requirements of enumeration would appear to indicate that enumeration might scale better than sieving in a networked setting, where bandwidth between nodes may become a significant bottleneck. More generally, the requirement for exponential memory in lattice sieves and its implication for realistic cost estimates of these algorithms has received significant attention around the NIST PQC standardisation process [AS17, NIS23, Jaq24, Sch24].

## 1.1 Contributions

Our contributions are (a) an efficient and scalable proof-of-concept implementation of sieving algorithms using MPI and (b) an investigation into

[3] In contrast, enumeration-based algorithms [Kan83, FP83, GNR10, MW15, ABF+20, ABLR21] run in time $n^{\Theta(n)}$ and $\mathsf{poly}(n)$ memory.

the scalability of sieving algorithms over, possibly heterogeneous, larger nodes connected over Ethernet.

In more detail, we study and implement distributed variants of the BGJ1 [BGJ15, ADH$^+$19] and the BDGL sieve variant [BDGL16, DSv21] used in G6K. This is motivated by their shared good performance but rather diverging designs, allowing us to explore the problem space.

After some preliminaries, we discuss how distributed sieving algorithms should scale increasingly well as the lattice dimension increases in Section 3. This discussion informs our architecture. We stress that our design is exclusively concerned with distributing so-called "buckets" across different nodes. In particular, we do not consider distributing a single such bucket across multiple nodes. This restricts our results to a setting of somewhat powerful nodes as the memory requirements per bucket grow as $2^{0.104\,n+o(n)}$ in our implementation.[4]

Then, using the open-source G6K implementation [ADH$^+$19] as a starting point, we present and describe an open-source implementation of distributed sieving in Section 4. It might appear straight-forward to parallelise both BGJ1 and BDGL by simply processing multiple buckets of vectors in parallel, which is indeed the strategy applied in G6K in a single-server setting. However, adopting this approach in a distributed context is not efficient. The main challenge is that G6K reduces the amount of memory needed to represent a bucket as a series of smaller indices into a much larger table to maintain consistency. We are unable to use this approach in our setting: indexing into the database in this way assumes that each node has access to a full copy of the database, which is not the case in the multiple server setting. Instead, we require that each node gathers buckets in their entirety that need to be sieved, requiring additional storage. The challenge is then to keep the additional memory usage small whilst also maintaining a small wall time.

Moreover, and critically, we found that preventing the insertion of duplicate entries into the database in a distributed setting is highly non-trivial, requiring some consensus across all nodes in the cluster. Interestingly, we find that bespoke consensus techniques should be applied to each sieve, depending on the underlying properties of each algorithm. We discuss the techniques applied for this in more detail in Section 4.

Finally, in Section 5 we show that our distributed BGJ1 lattice sieving implementation achieves the desired reduction in wall time when

[4] Asymptotically, bucket sizes of $2^{o(n)}$ are achieved, but known implementations, including ours, do not use parameters justifying such an asymptotic formula.

using a standard 10Gbps network, see Figure 1 and Table 2.[5] To the best of our knowledge, this is the first time that the efficiency of any distributed lattice sieving algorithm for general lattices has been experimentally demonstrated to be performant over such a network, and that distributed variants of G6K have been studied, which was given as an open problem in [BBC$^+$20].[6] Our results for BDGL are less favourable, see Tables 4 and 5, we discuss this below. Our implementation is available at https://github.com/joerowell/G6K-Dist-Sieve.

## 1.2 Related work

Both enumeration and sieving have previously been considered in parallel contexts. *Enumeration* has been studied widely in both locally parallel [HSB$^+$10, DHPS10, KSD$^+$11, DS10, CMP$^+$16, BBK19, PSZ21] and distributed [TKH18, TSN$^+$20] settings. It is now commonly accepted that enumeration scales well across multiple cores in a single machine setting, and the widely used fplll library supports multi-core enumeration by default [dt23]. On the other hand, the largest scale distributed results are due to [TKH18] and [TSN$^+$20]. We note that both of these works present results from large, well-connected clusters and globally shared memory [TKH18, §6.2].[7] Moreover, modern sieving implementations outperform enumeration [ADH$^+$19] and the Darmstadt SVP Hall of Fame is dominated by sieving. Thus, we ignore enumeration for the rest of this work.

Existing works on scaling *sieving* can be divided into three broad categories. First, there are several works that explore shared-memory parallelism for lattice sieving [MS11, IKMT14, MBL15], and the most performant open-source library for lattice sieving, G6K [ADH$^+$19], uses multi-core parallelism.

[5] We note that the "dimension" given is the *sieving dimension* rather than the dimension of the (approx-)SVP problem that is targeted in e.g. the Darmstadt SVP Challenges. The former is smaller than the latter due to "dimensions for free" [Duc18a]. For example, [ADH$^+$19] solved a 155 dimensional SVP HoF instance, but sieved in dimension up-to 127.

[6] The TU Darmstadt SVP Hall of Fame [LR24] includes entries that hint at distributed lattice sieving, but these results do not appear in the literature and appear to use bandwidth-rich clusters.

[7] The authors do not mention exactly how much global space was used: however, they do report that around 60GB [TKH18, 6.2] of lattice vectors remained at the end of the 150-dimensional SVP challenge, and that "several hundred gigabytes" would have been used in total.

Second, a line of work has considered extending G6K to operate in non-shared memory environments; for example, Andrzejczak and Gaj [AG20] implemented the inner product computations on FPGAs, and a G6K variant utilising GPUs was presented in [DSv21]. In both cases, the main idea is to store the sieving database $L$ in system memory and to delegate all sieving operations (such as bucketing and searching for reductions) to an external device. Despite that access to the external device is typically far slower than accessing system memory, these works show a significant speed-up over CPU-only lattice sieving: intuitively, these speedups are possible because the quadratic portion of the sieve – considering all pairs of vectors – can be constrained to use only the fast, local memory of the external devices. This masks the delay of loading vectors directly from memory.

We note that, despite the apparent similarities between these works and ours, there is still a significant difference between delegating computations to an external device and sieving across different servers. The foremost reason for this is that a network bus – typically Ethernet – is significantly slower than the buses considered in both [AG20, DSv21]. As a result, mitigating the loading latency is a significantly harder task than in a single-machine context. Moreover, internal buses are typically quite fault reliant. In contrast, packet loss is a common concern in networked applications, and mitigating for this loss of data naturally leads to performance penalties. Finally, the designs of [AG20, DSv21] assume that the entire database can fit into a single system's RAM, whereas our work assumes that no single entity is able to store the entire database in local memory.

Third, there has been some experimental investigations to how sieving scales across multiple-machine clusters [BNvdP14, TSY$^+$21]. In the first of these works, the authors presented results over ideal lattices in the ring $\mathbb{Z}[x]/(x^n + 1)$, where $n$ is a power of two, allowing for the required bandwidth to be reduced by a factor of $n$. Interestingly, this reduction of a factor $n$ does not affect our scalability model by much, see Section 3.1. On the other hand, [TSY$^+$21] uses a hybrid Gauss Sieve-enumeration algorithm to solve a dimension 134 SVP instance in around 100 hours on a super computer with 103,680 cores and with around 50Gbps of network bandwidth per CPU. Our results, in contrast, were gathered using significantly fewer resources. This is partially explained by that the Gauss Sieve [MV10b] is exponentially slower than both BGJ1 and BDGL, and thus our implementation benefits from a substantial algorithmic advantage, which we leverage to obtain good parallel speedups over slower interconnects.

In [Duc18b, Kir16, KMPM19] architectures for massively parallel sieving were sketched: a ring of devices computing inner-products. These architectures illustrate the viability of sieving in an area-times-time (AT) model and promise that sieving does indeed scale reasonably well. However, since the focus of these architecture sketches are bespoke, massive circuits, their focus is still quite different from ours and closer to the FPGA/GPU sieves discussed above. We study sieving on commodity CPUs connected over commodity networks, enabling cooperative sieving across different such nodes. On the other hand, by treating large scale clusters in a similar manner to these massive circuits we may expect similar levels of scaling.

At the time of writing a series of new CPU sieving records have recently been registered in the Darmstadt SVP Hall of Fame by Zhao, Ding and Yang [LR24, ZDY24]. These entries were achieved using a low-level optimised, multi-core implementation of a BGJ sieve [BGJ15] that places particular emphasis on minimising random memory accesses. From a certain perspective we may view our implementation as following a similar principle. However, here the database is split across multiple machines rather than system memory. We also note that the scalability analysis in Section 3 is broadly unaffected by the size of the buckets that are used and the underlying parameterisation of a particular sieving algorithm. Overall, we consider the improvements in [LR24, ZDY24] as orthogonal to this work.

## 2 Preliminaries

*Notation.* We start indexing at 0. Vectors and matrices are denoted by bold lower case letters and bold capital letters respectively. Unless stated otherwise, all vectors are column vectors and matrices $\mathbf{B} = (\mathbf{b}_0, \dots \mathbf{b}_{n-1})$ are comprised of column vectors. We denote the Euclidean norm of a vector $\mathbf{b}$ as $\|\mathbf{b}\|$. The size of an object is the length of its binary representation. For any two vectors $\mathbf{v}, \mathbf{u}$, we denote the inner product of $\mathbf{v}$ and $\mathbf{u}$ as $\langle \mathbf{v}, \mathbf{u} \rangle$. We define the *sign* function $\operatorname{sgn}(n) : \mathbb{R} \mapsto \{0, 1\} = \begin{cases} 1 \text{ if } n \geq 0 \\ 0 \text{ otherwise} \end{cases}$. We denote the exclusive-or (xor) operation by the symbol $\oplus$.

## 2.1 Lattices

Lattices are discrete additive subgroups of $\mathbb{R}^m$. A lattice $\mathcal{L}$ in $\mathbb{R}^m$ can be represented as a set of all integer linear combinations of $n \leq m$ linearly independent vectors $\mathbf{B} \coloneqq (\mathbf{b}_0, \dots \mathbf{b}_{n-1})$ in $\mathbb{R}^m$. We refer to this set of vectors as a *basis*. When $n = m$ then $\mathcal{L}$ is said to be *full-rank*. In this work, we will refer to $n$ (resp. $m$) as the *rank* (resp. *dimension*) of the lattice $\mathcal{L}$. As soon as $n \geq 2$, any lattice may be spanned by infinitely many bases; for some lattice $\mathcal{L}$, any two arbitrary bases $\mathbf{B}$ and $\mathbf{C}$ may be written as $\mathbf{B} = \mathbf{C} \cdot \mathbf{U}$, where $\mathbf{U}$ is some matrix with $|\det(\mathbf{U})| = 1$. Such a matrix $\mathbf{U}$ is referred to as a *unimodular matrix*. The *determinant* of $\mathcal{L}$, $\det(\mathcal{L}) = \sqrt{\det(\mathbf{B}^T \cdot \mathbf{B})}$ is invariant of the basis used, and thus an invariant of the lattice. If some group $\mathcal{L}' \subseteq \mathcal{L}$ is also a lattice, then we refer to $\mathcal{L}'$ as a *sublattice* of $\mathcal{L}$.

For a given basis $\mathbf{B}$ we define $\pi_i$ as a projection orthogonal to the span of $(\mathbf{b}_0, \dots \mathbf{b}_{i-1})$, and the Gram–Schmidt orthogonalisation of $\mathbf{B}$ as

$$\mathbf{B}^* = (\mathbf{b}_0^*, \dots, \mathbf{b}_{n-1}^*) = (\pi_0(\mathbf{b}_0), \dots, \pi_{n-1}(\mathbf{b}_{n-1})).$$

The *projected sublattice* $\mathcal{L}_{[\ell:r]}$ where $0 \leq \ell < r \leq n-1$ is defined as the lattice with basis $\mathbf{B}_{[\ell:r]} = (\pi_\ell(\mathbf{b}_\ell), \dots, \pi_\ell(\mathbf{b}_{r-1}))$. When working in the projected sublattice $\mathcal{L}_{[\ell:r]}$ we say we are working in the *context* $[\ell : r]$.

## 2.2 Sieving algorithms

At a high level, a sieve operates by producing some list $L$ of lattice vectors and then searching for integer linear combinations of list vectors that are short. For an appropriately sized list, iterating this procedure a polynomial number of times leads to a solution for SVP. In this work we focus on algorithms that consider pairs of vectors, so called *2-sieves*.[8] A key factor influencing the size of the list (and hence the time complexity of the sieve) is the distribution of the lattice vectors. Here, we follow the standard heuristic [NV08] that points in the list $L$ are independently and identically distributed uniformly across a thin spherical shell. Then, the key computation task in a sieving algorithm is a Near(est) Neighbour Search (NNS) on this spherical shell: find two vectors that are 'close' in

[8] There also exist heuristic sieving variants [BLS16, HK17, HKL18], known as $k$-*sieves* where the linear combination of $k > 2$ many list vectors are considered. This allows the memory requirements of the sieve to be reduced: for example, a 3-sieve presented in [BLS16] requires a database of size $2^{0.1887n+o(n)}$. However, $k$-sieves can also be parameterised along a time-memory trade-off curve, i.e. increasing the database size to $2^{0.210n+o(n)}$ in order to lower the time complexity, cf. the 3-sieve in [ADH$^+$19].

**Algorithm 1** An NV-style sieving step with a prefilter [NV08]

```
Input: Some list of vectors $L = \{\mathbf{v} \in \mathbb{R}^m\}$, a predicate function prefilter : $\mathbb{R}^m \times \mathbb{R}^m \mapsto \{0, 1\}$, a sieving radius $R$.
Output: A list of vectors $L' = \{\mathbf{v} : \|\mathbf{v}\| < R\}$
 1: $C = \varnothing$ // $C$ denotes the set of centres
 2: for $\mathbf{v} \in L$ do
 3:     if $\exists\, \mathbf{w} \in C$ : prefilter($\mathbf{v}, \mathbf{w}$) $= 1$ then
 4:         if $\|\mathbf{v} \pm \mathbf{w}\| < R$ then
 5:             add $\mathbf{v} \pm \mathbf{w}$ to $L'$
 6:         else
 7:             go to 10
 8:         end if
 9:     else
10:         add $\mathbf{v}$ to $C$
11:     end if
12: end for
13: return $L'$
```

the sense that their addition or subtraction produces a shorter vector, i.e. the angle between them is either very small $< \pi/3$ or large. Based on some geometric constants related to sphere packing [CS87], [NV08] show that $|L| = 2^{0.21n+o(n)}$, leading to a time complexity of $2^{0.42n+o(n)}$, since a naive sieve loop is quadratic in the list size.

In Algorithm 1 we reproduce a simple NV-style sieving algorithm where we tweak the default presentation of an NV-style sieve to include the use of a prefiltering operation. Whilst not strictly necessary, applying a prefilter can lead to substantial speedups in practice for CPU sieving, see below.

*Remark 1.* At first glance, the searching and reduction steps at Lines 3 and 4 would appear to be embarrassingly (or proudly) parallel: simply process all vectors $\mathbf{v}$ in the list $L$ in parallel. Yet, the list of centres may change in each iteration; as a result, simply processing all vectors in parallel misses reductions that would otherwise produce shorter vectors. In addition, it is expensive to maintain two distinct lists $L$ and $L'$. Instead, it is more efficient to modify the list $L$ directly, which in turn produces additional concurrency issues.

**Prefilters.** For prefiltering, the most performant variant used in practice is a `popcount` filter. This idea, which can be viewed as a variant of Charikar's SimHash filter [Cha02], was originally introduced for lattice sieving in [FBB+15] and was later extended in [Duc18a]: Let $z$ denote the length of the Simhash in bits. Sample $z$ sparse ternary vectors, and denote them as $\mathbf{r}_i$ for $i \in 0, \ldots\ z-1$. Let $h_i(\mathbf{v}) : \mathbb{R}^m \mapsto \mathbb{R}^m = \langle \mathbf{r}_i, \mathbf{v} \rangle$ denote a hash function. Then, the sketch function $H : \mathbb{R}^m \mapsto \mathbb{Z}_2^z$ can be defined as follows:

$$H(\mathbf{v}) = (\text{sgn}(h_0(v)), \ldots, \text{sgn}(h_{z-1}(v))).$$

Geometrically, each hash function can be thought of as a constraint on the elements of $\mathbf{v}$. So, vectors that are similar in direction will have similar sketches. Since $H$ produces bit-strings as output, prefiltering two vectors $\mathbf{v}, \mathbf{u}$ for similarity is reduced to computing the Hamming distance between their hashes $H(\mathbf{v}), H(\mathbf{u})$.

That is, given two hashes $H(\mathbf{v}), H(\mathbf{u})$, we first compute $x = H(\mathbf{v}) \oplus H(\mathbf{u})$ and then compute the Hamming weight of $x$. A low Hamming-weight vector implies that the hashes are similar: as a result, this filter can be used to quickly pre-filter vectors that are unlikely to lead to a reduction.

Note that it is typical to align $z$ to the word length of the underlying computer. In practice, the value of $z$ is typically set to 256-bits, which corresponds to 4 machine words. This leads to a filter that consists of around a dozen unvectorised x86 instructions: Ducas [Duc18a, §5.3] reports that this filter results in a speedup that is approximately half an order of magnitude over naively considering inner products between all the possible pairs of vectors in some bucket.

Such a filter will typically have some error rate, i.e. will not only filter out vectors that are not close, but also compute inner products against vectors that are too long. A simple solution is to scale the size of the database linearly to the error rate of the filter; for example, it was reported in [ADH+19] that optimal performance occurred when scaling the database by a factor of 3.2 in order to overcome the empirical 30% error rate of the `popcount` filter reported in [Duc18a]. Parameter choices for `popcount` were explored in [AGPS20].

**Bucketing.** More efficient sieving algorithms exploit the structure of the search space by *bucketing* $L$. Briefly, bucketing preprocesses the list $L$ into smaller sublists $L_0, \ldots, L_{\delta-1}$ within which the quadratic search then commences. Many works [Laa15, BGJ15, BDGL16] have gradually improved the time complexity, with the fastest known sieve [BDGL16] terminating after $2^{0.292n+o(n)}$ [BDGL16] operations on a classical computer. In Algorithm 2 we illustrate the idea of bucketing. Within a bucket, Algorithm 1 can then be run.

**Algorithm 2** A basic bucketing algorithm with one bucket.

```
Input: Some list of vectors $L = \{\mathbf{v} \in \mathbb{R}^m\}$, a predicate function $\text{prefilter}_B : \mathbb{R}^m \times \mathbb{R}^m \mapsto \{0, 1\}$ and a bucketing radius $R_B$.
Output: A bucket $B$ defined by $\mathbf{c}$, containing all vectors $\mathbf{u} : \|\mathbf{u} \pm \mathbf{c}| < R_B$.
 1: $B = \varnothing$ // bucket starts empty
 2: Choose $\mathbf{c}$ uniformly from $L$
 3: for $\mathbf{v} \in L$ do
 4:     if $\mathbf{c} \neq \mathbf{v}$ then
 5:         if $\text{prefilter}_B(\mathbf{v}, \mathbf{c}) = 1$ and $\|\mathbf{c} \pm \mathbf{v}\| < R_B$ then
 6:             add $\mathbf{v}$ to $B$
 7:         end if
 8:     end if
 9: end for
10: return $B$
```

To define a bucket, we may choose a list entry as a *centre* (this is what we illustrate in Algorithm 2) or specifically construct buckets where sorting into buckets is relatively cheap [BDGL16].

**Structured bucketing.** Bucketing can be improved by switching to a structured bucketing scheme, such as [BDGL16]. In such a scheme, we first split the lattice dimension $n$ into $t$ smaller blocks of dimensions $n_0, n_1, \ldots n_{t-1}$ that sum up to $n$. In practice, $t$ is typically chosen to be small, say at most 4. After applying a suitable orthonormal transformation to introduce some randomness, we then sample a set of random vectors $C_i \subset \mathbb{R}^{n_i}$ and produce the global set of bucket centres $C = C_0 \times \pm C_1 \times \ldots \pm C_{t-1}$. Importantly, for some vector $\mathbf{v}$ we can find the closest global bucket centre by finding the closest local bucket vector, implicitly evaluating $2^{t-1} \cdot \sum_i |C_i|$ bucket centres for a cost of around $\sum_i |C_i|$ inner products per vector. This algorithm is referred to as *list decoding.* In practice, list decoding can be made very efficient with some minor tweaks: for example, Ducas, Stevens and Van Woerden [DSv21] report that a single inner product can be computed in under 1.7 cycles on a single CPU core using `AVX2` instructions.

## 2.3 The General Sieve Kernel

The General Sieve Kernel(G6K) [ADH+19] is a lattice reduction framework that treats sieving algorithms as "stateful" entities, rather than black-box SVP oracles. That is, G6K utilises the fact that sieving in dimension $d$ produces a database $L$ of $2^{0.210\,d+o(d)}$ vectors, containing many short vectors. This approach, coupled with various low-level optimisations, has allowed the open-source implementation of G6K to break several TU Darmstadt SVP challenges [LR24].

As our implementation is based on the (CPU) version of G6K, we briefly discuss some details of G6K's operation in this section.

*Operation.* G6K can be viewed as an abstract machine that solves SVP by applying a series of transformations to some internal state. Conceptually, this internal state can be divided into two distinct portions. In the first case, G6K maintains a lattice basis $\mathbf{B} \in \mathbb{Z}^{d \times d}$ and its associated Gram–Schmidt orthogonalisation basis $\mathbf{B}^*$ and a series of positions $0 \leq \kappa \leq \ell \leq r \leq d$. These positions define the current *sieving context* $[\ell : r]$ and the current *lifting context* $[\kappa : r]$. Additionally, G6K also maintains a database $L$ of lattice vectors that live in the sieving context, and series of insertion candidates $\mathbf{c}_\kappa \in [\kappa : r], \ldots, \mathbf{c}_r \in [r : r]$. In practice, each vector $\mathbf{v} \in L$ is represented as an `Entry` that contains (amongst other data) the coefficient representation $\mathbf{w}$ i.e. $\mathbf{v} = \mathbf{B}_{[\ell:r]} \cdot \mathbf{w}$.

G6K manipulates its internal state using a series of abstract instructions. Notably, most of these instructions are independent of sieving, and solely relate to database management. On the one hand, G6K provides a series of instructions (`Extend Left`, `Extend Right` and `Shrink Left`) that change the sieving context of the database. Each of these operations are cheap to carry out in practice: the `Extend Left` operation can be achieved by applying Babai's Nearest Plane [Bab85] algorithm to each vector in the database, whereas the other instructions simply require truncating the coordinate representation of each vector. G6K also provides instructions for growing (`Grow`) and shrinking (`Shrink`) the database. In both cases, these instructions attempt to preserve the quality of the database by either attempting to sample (relatively short) vectors or by discarding the longest vectors in the database. Finally, G6K also provides a `Sieve` instruction that applies a sieving algorithm to the database, producing short vectors until some stopping condition is reached.

During the execution of the `Sieve` instruction, certain vectors are lifted (by repeatedly applying `Extend Left`) from $[\ell : r]$ to $[\kappa : r]$. If the lifted vector is shorter than a particular insertion candidate $\mathbf{c}_i$, then the lifted vector replaces $\mathbf{c}_i$ as an insertion candidate. If a particular insertion candidate $\mathbf{c}_j$ improves the basis substantially, then it may be inserted into $\mathbf{B}$ using the `Insert` instruction.

*Strategies.* We note that the aforementioned instructions can be combined to create *strategies* that dictate lattice reduction from an abstract perspective. One such strategy used in G6K is the progressive sieving strategy known as the *pump*. In this strategy, G6K starts with a small context and alternates the `Extend Left`, `Grow` and `Sieve` instructions until a particular target context is reached. Note that this strategy recycles the sieving database between contexts, allowing the sieve to start with relatively many short vectors. Once this target context has been reached, G6K applies a sequence of `Insert` and `Shrink` instructions to improve the quality of the lattice basis. Combining several of these pumps together is referred to as a *workout*, which gradually improves the quality of the basis. We note that the increase in norm added by applying `Extend Left` to a particular vector depends strongly on the quality of the basis.[9] Thus, iteratively improving the basis simultaneously reduces the amount of time needed for a particular pump and increases the possibility of finding a short lattice vector in $\mathcal{L}$.

## 2.4 Message Passing Interface (MPI)

We give a short summary of the Message Passing Interface (MPI) standard used in our implementation and experiments. The interested reader may refer to the MPI standard [For12] for more details.

*Messaging passing.* MPI can be viewed as an instantiation of the *message passing* model of concurrency. At a high-level, MPI programs are comprised of *groups* of *processes*. Each process has exclusive access to its own local memory and computational resources and may be further subdivided into a set of *threads*. In order to share data, processes communicate over shared, stateful *communicators* that act as channels.

We briefly describe how MPI handles messages. Namely, suppose that $A$ wishes to send a message $M$ to $B$ in a *point-to-point fashion*. To achieve this, $A$ supplies $M$ to an MPI procedure as a *message buffer*. At this stage, the MPI library inspects $M$ and decides on how $M$ should be sent. We remark that the scope for decision here is rather vast; for example, if $M$ is short then the MPI library may simply send $M$ to $B$ without any prior notice. On the other hand, sending a large $M$ in this way may overwhelm $B$, and thus the implementation may choose to inform $B$ in advance.

In order to reduce the complexity of sending messages, the MPI standard provides several *modes* that specify how procedures handle messages. A procedure is said to be *completed* if $A$ can re-use the message buffer without affecting the transmission of the message. Moreover, a procedure is said to be *blocking* if it does not return until after it has completed.

[9] This follows from the usage of Babai's Nearest Plane algorithm.

On the other hand, a *non-blocking* procedure may return immediately without completing i.e. $A$ may not be able to re-use the message buffer when the procedure returns. In order to determine when the buffer can be re-used, MPI allows the progress of a non-blocking procedure to be tracked via a *request* object. We note that MPI provides the ability for the programmer to explicitly choose if a particular message is sent using either a blocking or non-blocking procedure. This choice permits optimisations to be made explicitly; for example, non-blocking procedures enable several concurrent requests to be in progress at once, or for processing to be offloaded asynchronously on to a network card. On the other hand, blocking procedures may reduce memory usage in some settings as buffers can more easily be re-used by the programmer. In practice, we found that using non-blocking routines was more efficient in our use-case, and thus our implementation uses them extensively.

*Collective operations.* In addition to point-to-point communications, MPI also allows for multiple processes to exchange messages at once in a *collective* fashion. Whilst collective communications also come in blocking and non-blocking variants, collective communications can also take advantage of algorithmic improvements that are not available for point-to-point messages. For example, a broadcast from process $p_0$ across a group $P$ can be efficiently implemented by organising processes in a tree rooted at $p_0$, allowing multiple communication links to be used at once. These savings are often substantial; for example, pairwise message exchange (also known as `AlltoAll`) of $n$ messages between $p$ nodes can be optimally realised in $O(\log p)$ rounds [BHK+97], compared to $O(p^2)$ rounds using point-to-point messages. We note, however, that the "best" algorithm to use typically depends on the properties of the messages that are being transmitted and the characteristics of the underlying interconnect i.e. if multicast is supported. To handle this, MPI implementations typically select the appropriate algorithm on a case-by-case basis. However, this choice comes with an additional restriction; collective communications over a particular communicator must be called in the same order across all processes to prevent confusion. In practice, this restriction can cause programs to lose some flexibility, and care needs to be taken to use these operations safely in a fully asynchronous environment. However, in practice the performance benefits of using these operations is typically substantial, far outweighing any lost flexibility. We note that whilst MPI is rather high-level, substantial performance benefits can be realised by performing additional low-level optimisations: we describe our efforts in this regard in Section 4. Still, almost all network programming tasks such as e.g. heartbeating are handled by the MPI library and, thus, are hidden from the programmer. This simplicity allows the creation of highly complex distributed applications.

*The logP scalability model.* We analyse the scalability of distributed lattice sieving using the well-known *logP* [CKP+93] model of parallel systems. In contrast to other, simpler models, the logP model can be used to succinctly predict the cost of network activity in a topology-agnostic fashion. For brevity, we only give an introduction to this model here, and we refer the interested reader to [CKP+93] for further details.

At a high-level, the goal of the logP model is to express the costs of network activity in terms of machine cycles. In order to achieve this comparison, the logP model treats network activity as a function of four distinct parameters: the maximum latency of sending a single byte message ($\lambda$), the overhead of sending or receiving a single byte message ($\phi$), the "gap" in time between two successive messages ($g$) and the number of nodes in the network ($P$). The logP model can also be augmented to model loosely connected networks by adding two additional parameters: the maximum number of intermediate hops $H$, and the forwarding time at each hop $r$.[10] As each single byte message requires both some sending and receiving overhead, the time taken to send a single byte message in this model is $2 \cdot \phi + \lambda + H \cdot r$ cycles. We note that the logP model assumes that at most $\lceil \lambda / g \rfloor$ bytes may be in transit at any given time, and therefore care must be taken when handling potentially large messages. In order to handle larger messages generically, we assume that each message of $M$ bytes can be decomposed into at most $\sigma$ smaller chunks (i.e. $\sigma$ is the smallest integer satisfying $M \leq \sigma \cdot \lceil \lambda / g \rceil$), leading to a total cost of $\sigma \cdot (2 \cdot \phi + M/k \cdot (\lambda + g) + H \cdot r)$ cycles per message. Put differently, all cost calculations in this work incorporate network congestion.

## 3 Architecture

We are now ready to discuss our high-level architecture.

[10] Whilst $H$ could theoretically be as large as $P$ it is far more typical to see $H$ being at most$\approx \log P$, as nodes can always be re-arranged into a (potentially unbalanced) binary tree.

**Algorithm 3** An simplified bucketing algorithm for $p$ nodes.

```
Input: A global database of $N$ vectors, divided up into $N/p$ lists spread across $p$ nodes.
Output: A set of $q \cdot p$ buckets.
 1: Each $P_i$ chooses $C_i = (\mathbf{c}_0, \ldots, \mathbf{c}_{q-1})$ from its local database.
 2: for $0 \leq j < p - 1$ do
 3:     $P_i$ sends $C_{i-j \bmod p}$ to $P_{i+1 \bmod p}$ // $P_i$ receives $C_{i-j-1 \bmod p}$.
 4: end for
 5: for $j < p$ do
 6:     $P_i$ builds set $\beta_{(i,j)} \coloneqq \mathbf{bucket}(C_j)$ against their local database.
 7:     // $P_j$'s completed buckets are $\cup_i \beta_{(i,j)}$.
 8: end for
 9: for $0 \leq j < p - 1$ do
10:     $P_i$ sends $\beta_{(i,i-j \bmod p)}$ to $P_{i-j \bmod p}$ // $P_i$ receives $\beta_{(i+j \bmod p,i)}$
11: end for
12: $P_i$ sieves each bucket in $\cup_i \beta_{(i,j)}$.
```

## 3.1 Unstructured bucketing

First, we present a high-level scalability analysis of distributed sieving, focusing on sieving algorithms that use random database entries to define buckets. This analysis applies to sieving algorithms that either sieve quadratically over the entire database or use an unstructured form of bucketing (cf. Algorithms 1 and 2 respectively).[11] The goal of this analysis is to evaluate the ratio between the time spent communicating buckets $T_{comm}$, and the time spent processing them $T_{comp}$; a large ratio would imply that sieving is unsuitable for parallelisation, whereas a small ratio would imply that the computationally expensive parts of sieving can be parallelised efficiently.

For the purposes of exposition, we present a simplified sieving algorithm in Algorithm 3 on a network of nodes $P_0, \ldots, P_{p-1}$ and use this for our analysis. At a high-level Algorithm 3 works by building a total of $q \cdot p$ buckets per iteration i.e. $q$ per node before sieving them. In order to build these buckets, each node first chooses $q$ random vectors from their local database to act as bucket centres, which are then forwarded to every other node in the network. Upon receiving all centres $C_i$, each node builds a series of $q \cdot p$ local buckets against their database, which are then re-distributed in a pairwise fashion across the network.

We assume that all $p$ nodes are identically capable and all have equal access to the network. Given that all $p$ nodes are equally powerful, we split our database of size $N$ equally across all $p$ nodes, and thus each node

[11] The analysis can cover $k$-sieves by replacing $B^2$ terms in the denominator below by $B^k$.

holds approximately $N/p$ vectors. This assumption is valid since we may redistribute the database as we see fit. We also assume that each lattice vector in dimension $n$ is comprised of $n$ entries of $c$ bytes each, with $c$ being some constant that does not vary with $n$. Moreover, we assume that $c$ corresponds to a machine-friendly data-type (e.g. a single precision float) and thus assign a unit cost for both multiplication and addition of two $c$ byte numbers. We extend this and assign a cost of $2\cdot n-1$ operations to the task of computing the inner product of two lattices vectors in dimension $n$. Finally, for the sake of simplicity we assume that each built bucket contains exactly $B$ vectors; as the database is distributed evenly across all $p$ nodes, this implies that each bucket requires $B \cdot (p-1)/p$ vectors to be sent.

We now analyse Algorithm 3 by considering each stage in turn. To begin, notice that the first and second communication loops (i.e. the loops at Line 2 and Line 9, respectively) are almost identical, with both loops executing $p-1$ iterations. In fact, the only difference is the amount of data sent per iteration, with each node sending $q\cdot c\cdot n$ bytes per iteration in the first loop and $B\cdot q\cdot n\cdot c$ in the second. By letting $\sigma_1$ and $\sigma_2$ be the smallest integers satisfying $\sigma_1\cdot\lceil\lambda/g\rceil \geq q\cdot c\cdot n$ and $\sigma_2\cdot\lceil\lambda/g\rceil \geq B/p\cdot q\cdot n$, we conclude that the first loop requires $(p-1)\cdot\sigma_1\cdot(2\cdot\phi+q\cdot c\cdot n/\sigma_1\cdot(\lambda+g)+H\cdot r)$ cycles to terminate, with the second loop requiring $(p-1)\cdot\sigma_2\cdot(2\cdot\phi+B/p\cdot q\cdot n\cdot(\lambda+g)+H\cdot r)$ cycles. Thus, the communication time in Algorithm 3 is approximately

$$T_{comm} = (p-1)\cdot\Big((\sigma_1+\sigma_2)\cdot(2\phi+H\cdot r)+(\lambda+g)\cdot(B/p+1)\cdot(n\cdot c\cdot q)\Big)$$

cycles. We now consider the ratio between $T_{comm}$ and the time taken to produce and sieve all $q\cdot p$ buckets. On the one hand, building a single bucket requires a total of $N$ inner products, and thus around $N\cdot q\cdot(2\cdot n-1)$ cycles in total. Since sieving a bucket requires $B^2$ inner products, we get a total cost of $B^2\cdot p\cdot q\cdot(2\cdot n-1)$ cycles. Therefore

$$\begin{aligned}\frac{T_{comm}}{T_{comp}} &= \frac{(p-1)\cdot\Big((\sigma_1+\sigma_2)\cdot(2\phi+H\cdot r)+(\lambda+g)\cdot(B/p+1)\cdot(n\cdot c\cdot q)\Big)}{N\cdot q\cdot(2\cdot n-1)+B^2\cdot(2\cdot n-1)\cdot q\cdot p}\\ &= \frac{p-1}{p}\cdot\left(\frac{(\sigma_1+\sigma_2)\cdot(2\cdot\phi+H\cdot r)}{(N/p+B^2)\cdot(2\cdot n-1)\cdot q}+\frac{n\cdot c\cdot(B/p+1)\cdot(\lambda+g)}{(N/p+B^2)\cdot(2\cdot n-1)}\right).\end{aligned}$$

On the one hand, note that the leading $(p-1)/p$ term is bounded from above by 1 for any choice of $p$, and thus increasing $p$ with $n$ does not affect the scalability of sieving. Intuitively, this observation is consistent with the fact that $B$ is dictated solely by $n$, and thus increasing $p$ should not affect the amount of communication. On the other hand, the inner terms of the equation are both dominated by $B^2$ and $N$; even though $\sigma_2$ grows with $B$ (and hence exponentially in $n$) this increase is cancelled out by the $B^2$ term in the denominator. Moreover, as the second term only changes in terms of $B, n$ and $N$, we note that the same conclusion holds for that term, too.

*Remark 2.* Our analysis also shows that the factor $n$ saving in bandwidth for ideal lattices [BNvdP14] does not largely affect how sieving scales in an asymptotic sense. However, the factor $n$ reduction in bandwidth will still permit substantial improvements in practice.

*Remark 3.* At first glance it may appear appealing to add further parallelism to the aforementioned algorithm by also sub-dividing each bucket across $p$ nodes e.g. we may simply divide $B$ into $p$ blocks and pass these blocks around all $p$ nodes. However, as the size of the buckets grows slowly compared to the size of the overall database, this approach is unlikely to be useful in practice. Indeed, while the scalability analysis is broadly similar to the analysis given above, anecdotal evidence [ADH+19, Appendix B] suggests that processing a single bucket across multiple cores leads to substantially poorer parallelism in practice.

*Saturation in practice.* We now estimate, concretely, what throughput we require of our interconnect to saturate our computational units.

As a starting point of this analysis, we recall that the main computational task associated with processing a bucket of $B$ vectors is the computation of $B^2$ inner products. Given that high-performing sieving implementations [ADH+19, DSv21] typically represent each $n$-dimensional lattice vector as an array of $n$ 32-bit floating point values, we can naively lower bound how fast a particular sieve will execute on a particular computer by studying the number of inner products that can be executed per second. For the sake of simplicity, we assume that our goal is to take a single second to process a bucket of size $B$. Then, if we have a processor that executes $F$ 32-bit inner products per second, then we would take a second to process a bucket of size $B$ when $B = \sqrt{F}/(2n)$. From the perspective of distributed lattice sieving, this model implies that maximum parallelism can be achieved by supplying exactly $\sqrt{F}/(2n)$ vectors per processor per second; put differently, we would require the transmission of approximately $P \cdot \sqrt{F}/(2n)$ lattice vectors per second in order to fully saturate $P$ processors.

Next, we use the throughput figures provided by [DSv21] for an Intel Xeon Gold 6248 CPU, which can execute up to $F = 3.2 \cdot 10^{12} \approx 2^{41.5}$ 32-bit floating-point operations per second when using hardware accelerated instructions. Thus, in order to saturate such a processor a distributed system would need to provide approximately $\sqrt{F}/(2n)$ vectors [ADH+19, §5.1] per second over the network, each costing $2n$ bytes.

Using `popcount` filters changes this calculus slightly. For simplicity, pessimistically assume only the cost `popcount` counts. Considering again a Intel Xeon 6248 Gold with 20 cores at 2500Mhz and a popcount cost of six cycles, we obtain $M = 8.3 \cdot 10^9 \approx 2^{33}$ `popcount` calls per second. To exhaust this capacity, we need to send at least $\sqrt{M}$ vectors, which again cost $2\,n$ bytes per vector.

Concretely, picking $n = 128$ and a 1Gbps LAN, we can send $2^{30}/(16 \cdot 128) = 2^{19}$ vectors per second, requiring $2 \cdot 128 \cdot 2^{2\cdot 19} > F$ floating point operations or $2^{2\cdot 19} > M$ `popcount` applications to process. Even if we consider the same GPU as [DSv21] and 16-bit floating point arithmetic, we still only require around $1.47 \cdot 10^7 \approx 2^{24}$ bytes per second over the network. In summary, for sufficiently large instances the cost of computing $B^2$ inner products outweighs the cost of sending $B$ vectors over a network.

## 3.2 Structured bucketing

We now adapt our previous analysis to consider sieves that utilise structured bucketing, such as BDGL [BDGL16]. To begin, let $t$ be an integer and suppose, as before, that each node holds approximately $N/p$ vectors in their local databases. Moreover, we assume that the random codes $\pm C_0 \times \pm C_1 \times \cdots \times \pm C_{t-1}$ are evenly shared amongst all $p$ nodes i.e. each node is responsible for $m/p = 2^{t-1} \cdot \sum_i |C_i|/p$ buckets, and that each bucket contains $N^{1/t+1}$ vectors. Note that as the centres are randomly (but deterministically, i.e. from a seed) generated, we can distribute these centres by allowing one node to distribute a random seed across the network. As the size of the seed is asymptotically negligible, we simply assume that it has a fixed size of $d \leq \lceil \lambda/g \rceil$ bytes and thus requires $(p-1) \cdot (2 \cdot \phi + d \cdot (\lambda + g) + H \cdot r)$ cycles to transmit across the network. By assuming that each node contributes exactly $N^{1/t+1}/p$ vectors to each bucket spread over $\sigma_3$ messages and by re-using the second communication loop from Algorithm 3 we conclude that structured bucketing would spend around

$$T_{comm} = (p-1) \cdot \Big((1+\sigma_3) \cdot (H \cdot r + 2 \cdot \phi) + (\lambda + g)(N^{1/t+1} \cdot n \cdot c/p + d)\Big)$$

cycles on communication. On the other hand, recall that bucketing a single vector $\mathbf{v}$ can be done efficiently by considering around $m^{1/t}$ inner products, and hence bucketing the entire database requires approximately

$N \cdot m^{1/t} \cdot (2 \cdot n - 1)$ cycles. Given that processing a single bucket requires $N^{2/t+1} \cdot (2 \cdot n - 1)$ cycles, the ratio of communication to computation is

$$\begin{aligned}
\frac{T_{comm}}{T_{comp}} &= (p-1) \cdot \left( \frac{(1+\sigma_3) \cdot (H \cdot r + 2 \cdot \phi) + (\lambda + g) \cdot (N^{1/t+1} \cdot n \cdot c/p + d)}{(2 \cdot n - 1) \cdot (N \cdot m^{1/t} + N^{2/t+1} \cdot m)} \right) \\
&= \frac{(p-1)}{(N^{t/t+1} \cdot m^{1/t} + N^{1/t+1} \cdot m) \cdot (2 \cdot n - 1)} \\
&\quad \cdot \left( \frac{(1+\sigma_3) \cdot (H \cdot r + 2 \cdot \phi)}{N^{1/t+1}} + (\lambda + g) \cdot \left( \frac{n \cdot c}{p} + \frac{d}{N^{1/t+1}} \right) \right) .
\end{aligned}$$

First, note that for any choice of $t$ the leading $p - 1$ term tends to zero even if $p = N$. Moreover, as $\sigma_3$ is strictly less than $N^{1/(t+1)/p}$ the inner terms also grow at most linearly in $n$. As a result, the entire expression is dominated by $N$, and thus the ratio $T_{comm}/T_{comp}$ decreases as $N$ increases.

## 3.3 Buckets and nodes

As mentioned above, our design is concerned with a setting where several larger or "beefier" nodes jointly sieve over a distributed database. In particular, we do not consider distributing individual buckets over multiple nodes. Here, we argue that this is compatible with existing cluster setups.

First, recall that the optimal number of buckets is dictated by the sieve we consider. Considering both BGJ1 and BDGL with one level, i.e. the sieves considered in practice so far [ADH+19, DSv21], the database of size $N = 2^{0.2075\,n+o(n)}$ is stored in $O(\sqrt{N}) \approx 2^{0.1038\,n}$ buckets each holding $O(\sqrt{N}) \approx 2^{0.1038\,n}$ vectors. In [DSv21], the maximum sieving dimension considered was $n = 150$ and used 1.5TB of RAM. Thus, we expect $\approx 2^{15.6} \approx 50,000$ buckets. To put that into perspective, the Frontier supercomputer has 9472 nodes, each with 128GB of RAM.[12] Thus, given that we have more buckets than nodes already in this dimension, our design choice is compatible with current supercomputer architectures. Moreover, it is compatible with standard academic computing infrastructures where many, possibly heterogeneous, powerful nodes are connected via Ethernet.

## 4 Design & implementation

We adapted G6K [ADH+19] to support distributed variants of both BGJ1 and the relaxed BDGL sieve presented in [DSv21]. We chose to implement

[12] https://en.wikipedia.org/w/index.php?title=Frontier_(supercomputer)&oldid=1218026460

these algorithms as they naturally permit different implementation trade-offs and design choices that may be interesting in different contexts. For example, our BGJ1 implementation handles buckets by storing them in temporary storage, whereas our BDGL implementation writes received buckets directly into each node's local sieving database.

In order to separate our changes from the existing high-level code in G6K, we implemented all networking code in G6K's C++ layer by adding a stateful MPI object to G6K's Siever class. This separation of networking code and sieving code yields several benefits. On the one hand, separating network code from sieving code allows us to conditionally enable and disable MPI at compile-time, removing any runtime overhead of maintaining an unused object. On the other hand, as all networking code is hidden behind a well-defined interface, we allow the possibility of substituting MPI with other networking libraries in future. We assume no particular topology and instead allow MPI to organise nodes.

## 4.1 High-level design decisions

We briefly describe the operation of our networking code. For simplicity, our implementation assumes that the high-level Python layer associated with G6K executes on a single node, $\rho$, with all other nodes running a simple C++ program that interfaces with G6K. As $\rho$ is the node that receives high-level instructions from the Python layer, we ordain $\rho$ as the root node of the network, making $\rho$ responsible for issuing instructions to all other nodes. Thus, $\rho$ simply broadcasts all high-level instructions from the Python layer to the rest of the nodes, with each action handled opaquely from the Python layer. In practice, we represent these instructions as two 64-bit integers, allowing the transmission of an additional parameter where appropriate.

Instructions issued by $\rho$ typically trigger some additional distributed computation. On the one hand, $\rho$ may instruct all other nodes to engage in some sieving operation, which requires a large amount of bandwidth to execute successfully. On the other hand, certain context change operations also require additional work compared to the single node variant of G6K. For example, consider the task of shrinking the global database to contain the best $N$ vectors. In a single node setting, finding the best $N$ vectors can be achieved in $O(N)$ by using G6K's internally sorted list of vectors: however, in a multiple node setting we are required to discover the best $N$ vectors globally across many lists. We note, however, that these operations are still cheap compared to sieving itself, with even the most expensive operation requiring time linear in the global database size. As shown in

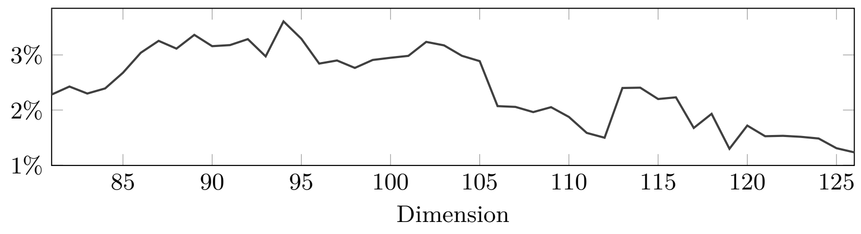


Fig. 2: Percentage of time spent executing context changes with dimension varying. The overhead time taken to execute context changes is small compared to the cost of sieving. Timings were gathered across nodes S, H, and A.

Figure 2, these costs are practically rather small compared to the cost of sieving, requiring at most 4% of execution time.

*Database division.* We divide the global sieving database amongst nodes, with more powerful nodes receiving a larger share of the global database. At a high-level, this approach ensures that the workload is divided fairly amongst the nodes in the cluster. We also note that operations such as e.g. lifting lattice vectors can be trivially parallelised across multiple machines.

*Serialisation of lattice vectors.* Recall that G6K represents each $n$-dimensional lattice vector $\mathbf{v} = \mathbf{B} \cdot \mathbf{x} \in \mathbb{Z}^n$ as an *entry* in a sieve database, storing both the (16-bit) integer coefficients $x$ and the (32-bit) vector $\mathbf{v}$ in each entry. Moreover, G6K also stores additional information about $\mathbf{v}$, such as its squared length and a unique identifier, leading to a cost of around 1 KiB of storage per lattice vector for $n = 128$. Given the constrained network bandwidth, we serialise $\mathbf{v}$ using its $\mathbf{x}$ representation, leading to a bandwidth cost of $2 \cdot n$ bytes per lattice vector. Whilst this representation incurs an additional cost of $\Theta(n^2)$ operations per lattice vector, we remark that this cost is rather small compared to sieving, especially for sufficiently large buckets. We empirically verify this claim below.

## 4.2 Database management

One particularly difficult aspect of distributed sieving is ensuring that the database stays free of duplicates. In more detail, the problem is that during sieving multiple nodes may produce the same lattice vector $\mathbf{v}$ and insert into their local database. We refer to this occurrence as a *collision.*

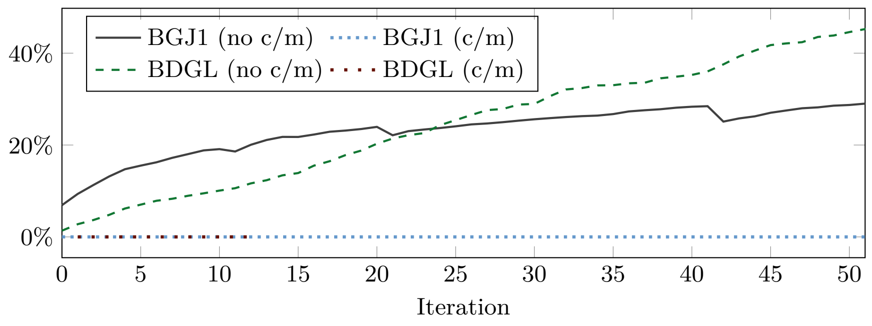


Fig. 3: Collision rate inside both distributed sieves with and without countermeasures (abbreviated as c/m). The number of duplicate entries increases steadily with the number of sieving iterations, necessitating our countermeasures. For BGJ1 the data was recorded inside a pump with $n \in \{55, \ldots, 60\}$, whereas for BDGL the data was recorded for $n = 68$ as the effect is more pronounced in higher dimensions. Each drop for BGJ1 corresponds to a change in sieving dimension.

As shown in Figure 3, we measured an increase in duplication of around 1% per sieving iteration. Given that lattice sieving algorithms typically require many iterations to terminate, the number of unique vectors in the database can thus quickly shrink.

Simply accepting this behaviour is not a viable strategy as this leads to a dramatic increase in the number of required buckets compared to single-machine sieving. This effect is to be expected: as each bucket contains a small number of unique vectors, most reductions are unlikely to yield short vectors. Moreover, these buckets likely contain multiple duplicate vectors, requiring expensive filtering to remove.

We address this issue by modifying G6K's internal hash table. Briefly, G6K maintains a hash table containing the 64-bit hash $H(\mathbf{v})$ of all vectors $\mathbf{v}$ in the sieving database. Each hash $H(\mathbf{v})$ is computed as the inner product of $\mathbf{v}$ with a global random vector in the ring $\mathbb{Z}/2^{64}\mathbb{Z}$. Notably, this hash scheme permits the computation of $H(\mathbf{v} \pm \mathbf{u}) = H(\mathbf{v}) \pm H(\mathbf{u})$, allowing duplicates to be rejected without requiring the computation of $\mathbf{v} \pm \mathbf{u}$.

For concurrency reasons, this internal hash table is actually subdivided into several individual hash tables $T_0, \ldots T_{h-1}$ that are individually synchronised across all active cores. In order to support this subdivision, G6K maps each $H(\mathbf{v})$ to the hash table indexed by $H(\mathbf{v}) \mod h$. Taking inspiration from this technique, we distribute the $T_i$ amongst all nodes in the cluster, with each node $N_j$ receiving some proper subset $M_j = T_{i_1}, T_{i_2}, \ldots$ of the set of hash tables. We then ensure consistency by requiring that each node maintains exclusive ownership over the vectors that belong to their hash tables: any insertion to a table $T_i$ must involve the owning node $N_j$ in some way. In practice, we implement $N_j$'s involvement in two separate ways.

*BGJ1.* As a first approach, we choose to tightly couple $N_j$'s internal database to $M_j$ i.e. we restrict $N_i$'s internal database to only containing vectors that live in one of the sub tables in $M_j$. In this approach, dealing with inserting $\mathbf{v}$ into the global database simply requires computing the hash of $\mathbf{v}$ before streaming $\mathbf{v}$ to the appropriate node $N_j$. From an implementation perspective, this approach comes with several trade-offs. First, this approach naturally maps to settings where produced buckets are only kept in memory for a short period of time: our implementation of the BGJ1 sieve, for example, discards of a received bucket as soon as it has been processed in order to save memory. In this setting, it is rather convenient to eagerly compute and store $\mathbf{v} = \mathbf{x} \pm \mathbf{y}$ without needing to worry about retaining $\mathbf{x}$ and $\mathbf{y}$. However, we note that this approach handles the situation where $\mathbf{v}$ is produced twice during the lifetime of the sieve rather lazily, relying on $N_i$ to handle the duplicate.

Moreover, we note that implementing this approach in a performant manner is rather challenging. On the one hand, streaming vectors one at a time requires little memory, but the latency costs for such small messages is likely to be prohibitive. Yet, naively batching vectors for insertions requires substantial extra storage: a slow node is likely to deal with insertions slowly, leading to many outstanding insertions on other nodes. In some instances, this cost is as large as the sieving database itself; our prototype implementation of this scheme, for example, required around 40GB of extra storage when sieving in dimension 113, whilst the sieving database required 34GB of storage. Whilst the relative cost of this extra storage decreases as the sieving dimension grows, the overhead of this approach is still noticeable even in large dimensions. We resolve these issues by handling insertions whenever a particular batch is finished, which prevents the lists of pending vectors from growing too large. We demonstrate the efficiency of our approach in Figure 3.

*BDGL.* As a second approach, we choose to decouple $N_j$'s internal database from $M_j$ i.e we allow $N_j$ to insert vectors that do not belong to $M_j$ into its local database. In this setting, we only require that $N_j$ tracks which insertions and removals have been made to $M_j$ during the lifetime of the sieve, without requiring that $N_j$ holds these insertions locally. Put differently, this approach allows $N_j$ to act as a membership oracle for $M_j$, rather than as a storage node for $M_i$. In contrast to the previous approach, this approach allows us to handle duplicated insertions globally: we may simply stream the hash $H(\mathbf{v} \pm \mathbf{u})$ to $N_j$, allowing $N_j$ to reject any vectors that are already present. In practice, we found this approach preferable in situations where buckets are retained for longer than in the BGJ1 case, as we no longer need to serialise new vectors across the network. We thus used this approach for our BDGL implementation. We demonstrate the efficiency of our approach in Figure 3, too.

### 4.3 BGJ1

At a high-level, our implementation of the BGJ1 sieve is almost identical to the approach described in Section 3.1, albeit with a few differences. For example, we do not insist that the sieving database is evenly divided across all nodes on the network, as mentioned above.

From an operation perspective, our implementation of the BGJ1 sieve runs in an iterative fashion (similarly to [DSv21]). For simplicity we describe this stage from the perspective of a single node, but note that this process is repeated in parallel across the entire cluster. Namely, suppose that some node $s$ wishes to produce a bucket defined as all lattice vectors in the database close to $\mathbf{c}$. To build this bucket, $s$ broadcasts $\mathbf{c}$ to all other nodes on the network, receiving in response the *number* of vectors $\ell$ that are close to $\mathbf{c}$ in the global database. In practice, we simply run the BGJ1 bucketing routine against $\mathbf{c}$ on each node and sum the count. At this stage, $s$ allocates enough storage to hold the $\ell$ vectors and reads the vectors that are close to $\mathbf{c}$ from the global database (over the network). Here, we store each received bucket in temporary storage that is separate from the main database: we discuss this in more detail in Section 4.3.

With the bucket $B$ produced, $s$ sieves over the bucket and inserts newly produced vectors in the global database. It turns out that database insertions require some additional care, see Section 4.2. Finally, $s$ simply repeats this process until the global database contains enough short vectors for the sieve to terminate.

This scheme can be easily parallelised via a series of modifications. The simplest of these modifications is to allow every node in the cluster to request buckets simultaneously, rather than sequentially. This transformation is trivial, as each bucketing iteration is independent. In practice, realising this functionality requires the use of additional synchronisation and multiple MPI communicators, which comes with negligible additional overhead. Moreover, this approach also allows us to utilise optimised `AlltoAll` implementations reducing the communication complexity for distributing $k$ buckets in parallel from $O(k^2)$ to around $O(\log k)$ [BHK+97].

We then further increase the throughput of bucketing by allowing each node to instead request *batches* of multiple buckets and for multiple such batches to be processed simultaneously. Intuitively, the presence of multiple batches establishes a pipeline of work for each node, reducing the amount of time that each node spends in an idle state. Moreover, as each batch and bucket can be processed independently, each node can use multiple threads for better local parallelism.

We note, however, that increasing both the number and size of each batch introduces a trade-off between local CPU utilisation and sieving iterations. This trade-off appears because the sieving algorithms used inside of G6K gradually improve the database quality as buckets are produced. Thus, if too many buckets are processed on, say, the first iteration, then the database is likely to only be slightly improved. In this vein, we allow nodes to vary the number of centres they issue depending on the size of their database. In practice, this choice reduces the number of sieving iterations compared to using the same number of centres per node. We remark that handling many buckets in parallel increases the memory requirements of each node, as many extra vectors needs to be stored for each bucket. However, in practice this extra overhead appears to be small compared to the size of the sieving database (see Figure 5), and we found that utilising multiple batches substantially improved CPU utilisation from around 40% to around 100% in dimensions as low as 75. In order to improve flexibility, we allow the size and number of batches to be controlled via a user-supplied parameter.

*Memory usage.* Our implementation uses additional memory compared to G6K, which might seem counterproductive given that distributed sieving is meant to go beyond the memory limits on a single server. We thus discuss these additional small overheads.

The extra memory use in our implementation can broadly be split into two categories. On the one hand, each node is required to store some additional state related to networking and job management compared to G6K. We find, however, that this cost is very small, requiring a maximum of around 5MB in our tests. We thus ignore these overheads and focus on the memory requirements introduced by sieving.

There are two potential memory inefficiencies that arise from how our implementation handles buckets. Recall that each node stores their received buckets in temporary storage, rather than in their local database. At first glance, this decision may seem surprising, as storing these buckets separately requires extra storage. In order to explain this decision, we recall that each database vector $\mathbf{v}$ may belong to several buckets that are in flight at once, rather than just one. Given that this is the case, storing $\mathbf{v}$ directly in each node's sieving database would either introduce duplicates globally, or require an intricate system for managing potentially overwritten vectors. In both cases, we found that the appropriate countermeasures were simply too slow to be performant, leading to an appreciable slowdown. Put differently, in practice we found it to be faster to simply store incoming buckets outside of the main database, at the cost of using slightly more memory.

We now turn our attention to minimising the overheads associated with this style of bucketing. Recall that, when a batch is processed, each node first learns the number of vectors that they will receive, followed by the vectors themselves. To restrict the memory usage of processing buckets, we represent each produced bucket as a series of database indices i.e. if $\mathbf{v} = db\,[i]$ belongs to a particular bucket, we simply store $i$. This reduces the cost of storing partially built buckets to around 4 bytes per vector. We note that this cost is rather low: for example, a bucket built with G6K's default BGJ1 parameterisation of approximately $3.2 \cdot 2^{0.10375\,n}$ vectors per bucket would require around 128KB of additional storage in dimension $n = 128$ in this representation. Of course, this optimisation only applies for the initial bucketing procedure and some conversion is needed before actual sieving occurs. We discuss a low-level optimisation to this process below.

Serialising the (vectors for the) buckets themselves is substantially more expensive. At a high-level, we serialise each bucket $\beta$ by copying the $\mathbf{x}$ coefficient representation of each vector in $\beta$ into a single C++ `std::vector`, which we then send across the network. Then, whenever a thread comes to process the bucket, we unpack the temporary vector into a thread-local set of entries. Given that entries are much larger than the coefficient representation, this leads to a large saving over naively storing the vectors as entries. In practice, this always saves storage over storing buckets in their `Entry` representation, as we never have fewer than one bucket per thread in a batch. Note that as a vector $\mathbf{v}$ belongs to a bucket with exponentially low probability we do not expect there to be much redundant traffic when serialising multiple buckets in this way compared to, say, a more clever system. This claim is empirically verified in Figure 4.

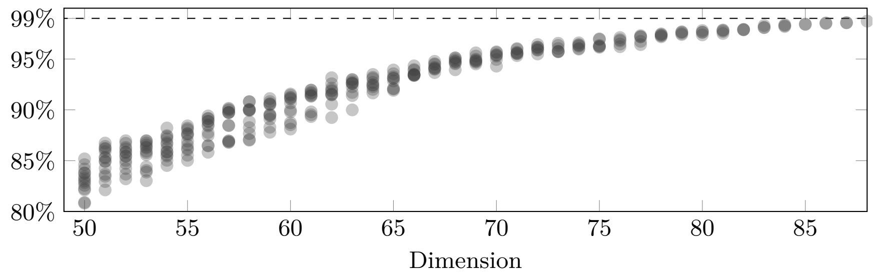


Fig. 4: Ratio of unique vectors and total vectors sent inside a BGJ1 distributed sieve. This chart shows that even when sending multiple buckets the number of unique vectors dominates the number of vectors that are sent.

Even with this rather naive scheme, we can see that the added memory requirements are minor compared to the size of the sieving database as the sieving dimension increases. Indeed, suppose that there are at most $b$ batches in flight at once, each containing $m$ buckets (each of size at most $B$). Then, as each $n$-dimensional vector $\mathbf{v}$ is represented in both its coefficient representation (requiring $2 \cdot n$ bytes) and its `Entry` representation (requiring around 1KB of storage for $n = 128$) we conclude that a node with $t$ threads will require approximately $B \cdot (b \cdot m \cdot n \cdot 2 + t \cdot 1000)$ bytes of additional storage. Finally, note that other than $B$ and $n$, all factors in this expression are runtime-choices that may be adjusted to suit the memory capacity of the target cluster. This, combined with that $B$ is roughly $2^{0.105\,n}$, means that this cost quickly becomes rather small compared to the storage needed to store the $2^{0.210\,n}$ entries that make up the global sieving database.

We further reduce this by allowing all $b$ batches to share $q \leq b$ buffers of temporary storage for serialisation, with buffers being re-used once a particular batch has been processed. We prevent deadlocks by enforcing that each node processes batches in sequential order: each node first processes batch 0, then batch 1, and so on. As this ordering is consistent globally, we require no expensive global synchronisation to enforce this ordering across nodes. In practice, this approach allows us to use relatively little extra memory compared to G6K, especially in the relevant dimensions. We show this effect in Figure 5. Additionally, we aggressively free memory as soon as it is no longer in use, re-allocating as needed.

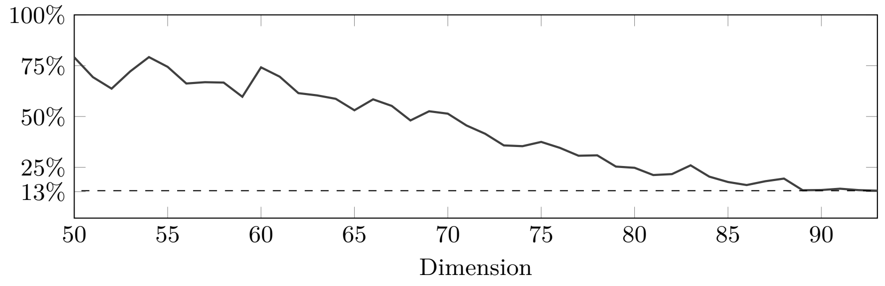


Fig. 5: Overhead memory rate: ratio of extra memory used and memory used for the sieving database inside a BGJ1 distributed sieve. The data here was recorded across a pump with $n \in \{50, \ldots, 93\}$.

*Low-level optimisation.* Our implementation makes use of several low-level optimisations to improve performance, of which we highlight a few here. Firstly, we store all received buckets contiguously in memory, i.e. in one memory region, substantially improving memory access patterns. Intuitively, this optimisation comes "for free" with distributed sieving: as each bucket needs to be received from multiple nodes, we may arrange them in memory in an optimal order for sieving. We stress that this optimisation is not free compared to the original version of G6K, as this choice requires extra memory compared to e.g. storing the vectors directly in the database. However, this optimisation enables several further optimisations: for example, as the location of these vectors in memory is no longer entirely random, we are able to reduce the amount of storage needed for G6K's compressed lists by around 50%. The combined effect of these optimisations means that our implementation performs nearly identically to the original version of G6K on a single machine, see Appendix C. For larger dimensions and multiple machines see Table 3.

### 4.4 BDGL

We describe our BDGL implementation in Appendix B.

## 5 Experimental results

All of our experiments use MPICH 4.1.1 with full optimisations enabled. In the case of our BGJ1 experiments we begin distributed sieving in dimension 90 with 8 bucketing batches and 4 auxiliary buffers. Each batch contains one bucket per thread per node e.g node K received 14 or 28 buckets per batch depending on the experiment we ran. We give experimental results in Tables 2 to 4 and 6b in Appendix A. We report our experimental results for BDGL in Appendix B. The nodes referred to in these tables are listed in Table 1. Each experiment was executed exactly once i.e. the timings given here were the result of exactly one experiment.

Table 1: Details of the machines used for experiments.

| N | CPUs | F | C | RAM | N | CPUs | F | C | RAM |
|---|---|---|---|---|---|---|---|---|---|
| H | 2x Xeon Gold 6252 | 2.1GHz | 96 | 768GiB | A | 2x Xeon E5-2690v4 | 2.6GHz | 28 | 256GiB |
| S | 2x Xeon Gold 6138 | 2.0GHz | 40 | 384GiB | K | 2x Xeon Gold 6142 | 2.6GHz | 32 | 192GiB |
| D | 1x Xeon Gold 6138 | 2.0GHz | 20 | 32GiB | | | | | |

Column “N” gives the node label, “F” gives the base frequency, “C” the number of physical cores. Experiments had hyper-threading disabled and “Turbo” frequency enabled.

## Acknowledgements

We thank Léo Ducas, Marc Stevens and Wessel Van Woerden for sharing an advanced copy of their BDGL bucketing implementation with us. We also thank the King’s College London e-Research group for allowing us to use CREATE [Lon24] for the bulk of the experiments in this work. This work was supported in part by EPSRC grants EP/S020330/1, EP/S02087X/1, EP/P009301/1, EP/Y02432X/1 and by European Union Horizon 2020 Research and Innovation Program Grant 780701.

## References

ABF$^+$20. Martin R. Albrecht, Shi Bai, Pierre-Alain Fouque, Paul Kirchner, Damien Stehlé, and Weiqiang Wen. Faster enumeration-based lattice reduction: Root hermite factor $k^{1/(2k)}$ time $k^{k/8+o(k)}$. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 186–212. Springer, Heidelberg, August 2020. 1, 3

ABLR21. Martin R. Albrecht, Shi Bai, Jianwei Li, and Joe Rowell. Lattice reduction with approximate enumeration oracles - practical algorithms and concrete performance. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part II*, volume 12826 of *LNCS*, pages 732–759, Virtual Event, August 2021. Springer, Heidelberg. 1, 3

AD18. Michel Abdalla and Ricardo Dahab, editors. *PKC 2018, Part I*, volume 10769 of *LNCS*. Springer, Heidelberg, March 2018. 5

ADH$^+$19. Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 717–746. Springer, Heidelberg, May 2019. 1, 1.1, 1.2, 5, 8, 2.2, 2.3, 3, 3.1, 3.3, 4, 4, C, 3

ADRS15. Divesh Aggarwal, Daniel Dadush, Oded Regev, and Noah Stephens-Davidowitz. Solving the shortest vector problem in $2^n$ time using discrete Gaussian sampling: Extended abstract. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 733–742. ACM Press, June 2015. 1

AG20. Michal Andrzejczak and Kris Gaj. A multiplatform parallel approach for lattice sieving algorithms. In *Algorithms and Architectures for Parallel Processing*, pages 661–680, Cham, 2020. Springer International Publishing. 1.2

AGPS20. Martin R. Albrecht, Vlad Gheorghiu, Eamonn W. Postlethwaite, and John M. Schanck. Estimating quantum speedups for lattice sieves. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 583–613. Springer, Heidelberg, December 2020. 2.2

Ajt98. Miklós Ajtai. The shortest vector problem in L2 is NP-hard for randomized reductions (extended abstract). In *30th ACM STOC*, pages 10–19. ACM Press, May 1998. 1

AKS01. Miklós Ajtai, Ravi Kumar, and D. Sivakumar. A sieve algorithm for the shortest lattice vector problem. In *33rd ACM STOC*, pages 601–610. ACM Press, July 2001. 1

AS17. Jacob Alperin-Sheriff. NIST's PQC Standardization: Suggested avenues for lattice-based research. Talk, slides available at http://crypto-events.di.ens.fr/LATCA/program/alperin-sheriff.pdf, May 2017. 1

Bab85. László Babai. On lovász' lattice reduction and the nearest lattice point problem (shortened version). In Kurt Mehlhorn, editor, *STACS '86*, volume 82 of *Lecture Notes in Computer Science*, pages 13–20. Springer, Heidelberg, 1985. 2.3

BBC$^+$20. Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions. 1, 1.1

BBK19. Michael Burger, Christian Bischof, and Juliane Krämer. p3Enum: A new parameterizable and shared-memory parallelized shortest vector problem solver. In *Computational Science – ICCS 2019*, pages 535–542, 2019. 1.2

BDGL16. Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In Robert Krauthgamer, editor, *27th SODA*, pages 10–24. ACM-SIAM, January 2016. 1, 1.1, 2.2, 2.2, 3.2

Ber16. Daniel Bernstein. Re: Inaccurate security claims in NTRUprime. Cryptanalytic algorithms mailing list, May 2016. https://groups.google.com/g/cryptanalytic-algorithms/c/BoSRL0uHIjM/m/eB4G-dscCAAJ. 1

BGJ15. Anja Becker, Nicolas Gama, and Antoine Joux. Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search. Cryptology ePrint Archive, Report 2015/522, 2015. https://eprint.iacr.org/2015/522. 1, 1.1, 1.2, 2.2

BHK+97. J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997. 2.4, 4.3

BLS16. Shi Bai, Thijs Laarhoven, and Damien Stehle. Tuple lattice sieving. *LMS Journal of Computation and Mathematics*, 19(A), 2016. 8

BNvdP14. Joppe W. Bos, Michael Naehrig, and Joop van de Pol. Sieving for shortest vectors in ideal lattices: a practical perspective. Cryptology ePrint Archive, Report 2014/880, 2014. https://eprint.iacr.org/2014/880. 1.2, 2

Cha02. Moses Charikar. Similarity estimation techniques from rounding algorithms. In *34th ACM STOC*, pages 380–388. ACM Press, May 2002. 2.2

CKP+93. David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. *SIGPLAN Not.*, 28(7):1–12, jul 1993. 2.4

CMP+16. Fábio Correia, Artur Mariano, Alberto Proença, Christian Bischof, and Erik Agrell. Parallel improved schnorr-euchner enumeration SE++ for the CVP and SVP. In *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 596–603, 2016. 1.2

CS87. J. H. Conway and N. J. A. Sloane. *Sphere-packings, Lattices, and Groups*. Springer, 1987. 2.2

DHPS10. Jérémie Detrey, Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Accelerating lattice reduction with FPGAs. In Michel Abdalla and Paulo S. L. M. Barreto, editors, *LATINCRYPT 2010*, volume 6212 of *LNCS*, pages 124–143. Springer, Heidelberg, August 2010. 1.2

DS10. Özgür Dagdelen and Michael Schneider. Parallel enumeration of shortest lattice vectors. In *Euro-Par 2010 - Parallel Processing*, pages 211–222, 2010. 1.2

DSv21. Léo Ducas, Marc Stevens, and Wessel P. J. van Woerden. Advanced lattice sieving on GPUs, with tensor cores. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 249–279. Springer, Heidelberg, October 2021. 1, 1.1, 1.2, 2.2, 3.1, 3.3, 4, 4.3, 4, B, B, B

dt23. The FPLLL development team. fplll, a lattice reduction library, Version: 5.4.4. Available at https://github.com/fplll/fplll, 2023. 1.2

Duc18a. Léo Ducas. Shortest vector from lattice sieving: A few dimensions for free. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 125–145. Springer, Heidelberg, April / May 2018. 1, 5, 2.2

Duc18b. Léo Ducas. Shortest vector from lattice sieving: a few dimensions for free. Presentation at EUROCRYPT 2018, April 2018. https://eurocrypt.iacr.org/2018/Slides/Monday/TrackB/01-01.pdf. 1.2

FBB+15. Robert Fitzpatrick, Christian H. Bischof, Johannes Buchmann, Özgür Dagdelen, Florian Göpfert, Artur Mariano, and Bo-Yin Yang. Tuning GaussSieve for speed. In Diego F. Aranha and Alfred Menezes, editors, *LATINCRYPT 2014*, volume 8895 of *LNCS*, pages 288–305. Springer, Heidelberg, September 2015. 2.2

For12. Message Passing Interface Forum. **MPI**: A message-passing interface standard, 2012. 2.4

FP83. Ulrich Fincke and Michael Pohst. A procedure for determining algebraic integers of given norm. In J. A. van Hulzen, editor, *EUROCAL*, volume 162 of *LNCS*, pages 194–202. Springer, 1983. 1, 3

GNR10. Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 257–278. Springer, Heidelberg, May / June 2010. 1, 3

HK17. Gottfried Herold and Elena Kirshanova. Improved algorithms for the approximate $k$-list problem in euclidean norm. In Serge Fehr, editor, *PKC 2017, Part I*, volume 10174 of *LNCS*, pages 16–40. Springer, Heidelberg, March 2017. 1, 8

HKL18. Gottfried Herold, Elena Kirshanova, and Thijs Laarhoven. Speed-ups and time-memory trade-offs for tuple lattice sieving. In Abdalla and Dahab [AD18], pages 407–436. 8

HR12. Ishay Haviv and Oded Regev. Tensor-based hardness of the shortest vector problem to within almost polynomial factors. *Theory of Computing*, 8(1):513–531, 2012. Preliminary version in *Proceedings of STOC '07*. 1

HSB$^+$10. Jens Hermans, Michael Schneider, Johannes Buchmann, Frederik Vercauteren, and Bart Preneel. Parallel shortest lattice vector enumeration on graphics cards. In Daniel J. Bernstein and Tanja Lange, editors, *AFRICACRYPT 10*, volume 6055 of *LNCS*, pages 52–68. Springer, Heidelberg, May 2010. 1.2

IKMT14. Tsukasa Ishiguro, Shinsaku Kiyomoto, Yutaka Miyake, and Tsuyoshi Takagi. Parallel gauss sieve algorithm: Solving the SVP challenge over a 128-dimensional ideal lattice. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 411–428. Springer, Heidelberg, March 2014. 1.2

Jaq24. Samuel Jaques. Memory adds no cost to lattice sieving for computers in 3 or more spatial dimensions. Cryptology ePrint Archive, Paper 2024/080, 2024. https://eprint.iacr.org/2024/080. 1

Kan83. Ravi Kannan. Improved algorithms for integer programming and related lattice problems. In *15th ACM STOC*, pages 193–206. ACM Press, April 1983. 1, 3

Kho05. Subhash Khot. Hardness of approximating the shortest vector problem in lattices. *Journal of the ACM*, 52(5):789–808, 2005. Preliminary version in *Proceedings of FOCS '04*. 1

Kir16. Paul Kirchner. Re: Inaccurate security claims in NTRUprime. Cryptanalytic algorithms mailing list, May 2016. https://groups.google.com/g/cryptanalytic-algorithms/c/BoSRL0uHIjM/m/wAkZQlwRAgAJ. 1.2

KMPM19. Elena Kirshanova, Erik Mårtensson, Eamonn W. Postlethwaite, and Subhayan Roy Moulik. Quantum algorithms for the approximate k-list problem and their application to lattice sieving. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part I*, volume 11921 of *LNCS*, pages 521–551. Springer, Heidelberg, December 2019. 1.2

KSD$^+$11. 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 - - how many dollars you need to break SVP challenges -. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 176–191. Springer, Heidelberg, September / October 2011. 1.2

Laa15. Thijs Laarhoven. Sieving for shortest vectors in lattices using angular locality-sensitive hashing. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part I*, volume 9215 of *LNCS*, pages 3–22. Springer, Heidelberg, August 2015. 1, 2.2

Lon24. King's College London. King's computational research, engineering and technology environment (create), 2024. Retrieved May 6, 2024 from https://doi.org/10.18742/rnvf-m076. 5

LR24. R. Lindner and M. Ruckert. TU Darmstadt lattice challenge. Available at http://www.latticechallenge.org/, 2024. 1, 6, 1.2, 2.3

MBL15. Artur Mariano, Christian Bischof, and Thijs Laarhoven. Parallel (probable) lock-free hash sieve: A practical sieving algorithm for the SVP. In *2015 44th International Conference on Parallel Processing*, pages 590–599, 2015. 1.2

MG13. Zoltan Majo and Thomas R. Gross. (mis)understanding the numa memory system performance of multithreaded workloads. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 2013. C

Mic01. Daniele Micciancio. The shortest vector in a lattice is hard to approximate to within some constant. *SIAM Journal on Computing*, 30(6):2008–2035, 2001. Preliminary version in *Proceedings of FOCS '98*. 1

Mic12. Daniele Micciancio. Inapproximability of the shortest vector problem: Toward a deterministic reduction. *Theory of Computing*, 8(22):487–512, 2012. 1

MS11. Benjamin Milde and Michael Schneider. A parallel implementation of gausssieve for the shortest vector problem in lattices. In *Parallel Computing Technologies*, pages 452–458, 2011. 1.2

MV10a. Daniele Micciancio and Panagiotis Voulgaris. A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations. In Leonard J. Schulman, editor, *42nd ACM STOC*, pages 351–358. ACM Press, June 2010. 1

MV10b. Daniele Micciancio and Panagiotis Voulgaris. Faster exponential time algorithms for the shortest vector problem. In Moses Charika, editor, *21st SODA*, pages 1468–1480. ACM-SIAM, January 2010. 1, 1.2

MW15. Daniele Micciancio and Michael Walter. Fast lattice point enumeration with minimal overhead. In Piotr Indyk, editor, *26th SODA*, pages 276–294. ACM-SIAM, January 2015. 1, 3

NIS23. NIST. FAQ on Kyber512. https://csrc.nist.gov/csrc/media/Projects/post-quantum-cryptography/documents/faq/Kyber-512-FAQ.pdf, December 2023. 1

NV08. Phong Q. Nguyen and Thomas Vidick. Sieve algorithms for the shortest vector problem are practical. *J. of Mathematical Cryptology*, 2(2), 2008. 1, 2.2, 1

PSZ21. Simon Pohmann, Marc Stevens, and Jens Zumbrägel. Lattice enumeration on GPUs for fplll. Cryptology ePrint Archive, Paper 2021/430, 2021. https://eprint.iacr.org/2021/430. 1.2

Sch24. John Schanck. An Update on Lattice Cryptanalysis vol. 2. Invited talk delivered at RWPQC'24, March 2024. https://na.eventscloud.com/website/65452/presentations-and-video-/. 1

TKH18. Tadanori Teruya, Kenji Kashiwabara, and Goichiro Hanaoka. Fast lattice basis reduction suitable for massive parallelization and its application to the shortest vector problem. In Abdalla and Dahab [AD18], pages 437–460. 1.2, 7

TSN+20. 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, 2020. 1.2

TSY+21. Nariaki Tateiwa, Yuji Shinano, Keiichiro Yamamura, Akihiro Yoshida, Shizuo Kaji, Masaya Yasuda, and Katsuki Fujisawa. CMAP-LAP: Configurable massively parallel solver for lattice problems. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 42–52, 2021. 1.2

ZDY24. Ziyu Zhao, Jintai Ding, and Bo-Yin Yang. Bgj15 revisited: Sieving with streamed memory access. Cryptology ePrint Archive, Paper 2024/739, 2024. https://eprint.iacr.org/2024/739. 1.2

## A Additional Benchmarks

In Table 2 we give our main benchmarks on a homogeneous set of up to five servers. This data is also plotted in Figure 1. In Table 6b we compare our implementation with the original version G6K to establish that it has comparable performance in a single machine *multi-CPU* setting. In Table 6a we compare our implementation with the original version of G6K in a *single-CPU* setting to measure the overhead of using MPI in such an environment. In Table 3 we give benchmarks using a heterogeneous network of three servers connected via 1Gbps Ethernet only.

*Remark 4.* We compare against CPU G6K [ADH+19] rather than the GPU variant [DSv21]. This is to measure the impact of distributed computing rather than racing against a more performant GPU implementation. A natural open problem is to utilise our distributed implementation to collaboratively sieve on many GPU-augmented servers.

Given that comparing heterogeneous experiments as in Table 3 is rather delicate, we explicate our methodology here. First, using the most recent version of G6K[13], we ran our BGJ1 experiments on node H to establish the expected wall-time on a single machine. Then, we repeated the BGJ1 experiments using nodes H, S and A, recording the wall-time. Note that all experiments used 35 of the 40 cores available on node S due to system instability. We then normalised the single node and distributed wall times by the number of cores used multiplied by the clock speed. That is, we compute "parallel efficiency" as:

$$\text{“parallel efficiency”} := \frac{\text{clock speed}_H \cdot \#\text{cores}_H \cdot \text{wall time}_{\text{single}}}{\sum_{c \in \{H,S,A\}} \text{clock speed}_c \cdot \#\text{cores}_c \cdot \text{wall time}_{\text{dist}}}.$$

[13] Commit 959fd8f

Under this metric, a performance score of 1.0 is ideal, anything above should be considered a measurement error and values $\leq 1.0$ signify less than ideal scaling. Put differently, under this loose metric, our BGJ1 implementation achieves the desired linear speed-up also in a heterogeneous distributed setting we considered. We stress, though, that this approach can at best give a rough indication of what performance to expect as it ignores factors such as available instruction sets, RAM speeds, "turbo boost" etc. We consider our homogeneous benchmarks in Table 2 a more reliable indicator. Yet, given that many academic teams may have a heterogeneous "cluster" of servers we also report these heterogeneous timings here.

In the homogeneous case, this comparison straight-forwardly simplifies to wall time divided by the number of cores.

## B BDGL

Our BDGL implementation is again rather similar to the theoretical model from Section 3.2. As before, we assume no particular topology and place no restrictions of communications between nodes. We leave it to future work to determine if more intricate topologies can achieve greater parallelism.

From a practical perspective, our implementation of BDGL works as follows. Similarly to the BDGL implementation found in G6K, our BDGL sieve follows a round-based strategy, with sieving broken up into distinct iterations. At the beginning of a sieving iteration, a single node chooses a set of codes to act as bucket centres, which are then distributed across the rest of the network. Notably, this distributes buckets according to the power of each node, with more powerful nodes receiving more buckets to sieve. Each node then locally carries out list-decoding over their database, producing a series of local buckets. Once all nodes have completed this step, all nodes iteratively request buckets to process in a similar manner to our BGJ1 implementation. Upon receiving their buckets, each node begins to sieve, producing new vectors for insertion. Given that these new vectors are unlikely to be unique across the network, we follow a strategy similar to the one followed by the BDGL implementation already present in G6K. In particular, each node $N_i$ starts sieving with an empty list $R_i$ that is used to store potential reductions: any time a new candidate vector $\mathbf{v} = \mathbf{x} \pm \mathbf{y}$ is found by $N_i$, an entry is added to $R_i$ containing $\mathbf{v}$ and its unique identifier. This choice introduces a trade-off between sieving iterations and memory, as storing more potential insertions increases memory usage. For the sake of our prototype implementation, we do not restrict how many entries are added to $R_i$, but we do ensure that $R_i$ is always free of duplicates. Once all nodes have finished sieving, each $N_i$ executes a membership query on each $r_k \in R_i$ by first mapping $r_k$ to the correct hash table slot $M_j$ and then querying $N_j$. If $N_j$ indicates that $r_k$ is already in the global database (or if $N_j$ has already been queried with $r_k$ in this round), then $N_i$ removes $r_k$ from $R_i$. With this completed, all surviving entries in $R_i$ are processed and inserted into $N_i$'s local database. Given that inserting some vector $\mathbf{v}$ into $N_i$'s local database requires removing some other vector $\mathbf{u}$, we again map $\mathbf{u}$'s hash to its hash table slot $M_j$ and forward this hash $N_j$ for removal from the hash table. Once these insertions have finished, all nodes check whether the database is sufficiently reduced to terminate and continue if not. In practice, this leads to our BDGL implementation performing less efficiently than our BGJ1 implementation, similarly to the results presented in [DSv21].

*Low-level optimisation.* As a side contribution, we realise the BDGL-style bucketing using a modified version of the `AVX2` bucketer provided in [DSv21]. In contrast to relying directly on `AVX2` intrinsics, our implementation instead uses GCC's vector extensions, allowing us to run the bucketer on any machine supported by GCC. At the time of writing, this bucketer has already been merged into G6K; however, as it may be of standalone interest we also provide this bucketer as a separate program.

*Experimental results.* We give experimental results for our BDGL implementation in both Table 4 and Table 5. In the language of Section 2.2, our experiments were conducted with the default G6K parameters of $t = 2$ i.e. for a database of $N$ lattice vectors we expect each bucket to contain approximately $N^{1/3}$ vectors. We took this choice to highlight the effects of asymptotically smaller bucket sizes on the performance of our distributed implementation. With that said, we observe that our BDGL implementation performs slightly worse than our BGJ1 implementation in terms of wall time, as illustrated in both Table 4 and Table 5. On the other hand, the amount of used CPU time is actually slightly better for BDGL than BGJ1, indicating that the small bucket sizes in low dimensions prevent the masking of network latency. This is to be expected: a similar conclusion was reached in [DSv21], where an estimated crossover for BDGL and a triple sieve variant on GPUs was stated to be around dimension 130. Given that the serialisation costs between nodes is higher than the cost of serialising vectors to a GPU, we expect that the crossover in our setting would be substantially higher than dimension 130. However, in such low dimensions our BGJ1 implementation also uses more memory than our BDGL implementation.

## C Comparison with the original version of G6K

As mentioned in Section 1.2, the original version of G6K contains several multi-threaded implementations of lattice sieves. In more detail, the parallel sieves in G6K utilise task parallelism, using $T$ threads to process $T$ independent tasks at once. For instance, in the case of the BGJ1 sieve, G6K uses $T$ threads to build and sieve $T$ buckets in parallel, with each thread working broadly independently. Moreover, the sieving implementations in G6K are carefully crafted to avoid common pitfalls in multi-threaded programming, such as lock contention and false sharing. However, we remark that the original version of G6K is not designed to handle certain parallelism-based performance bottlenecks. Indeed, the task-based parallelism in G6K allows each thread to access the entirety of the system memory without restriction i.e. it assumes a uniform memory space. In a single CPU setting, this assumption holds; each thread runs on the same physical CPU, and thus access to system memory has broadly the same cost across all threads. However, modern multi-processor machines are typically designed with a *non-uniform memory architecture* (NUMA) i.e. each physical CPU has access to its own local system memory. In this setting, a thread $t_i$ running on, say, CPU 0 can access CPU 0's local memory fairly cheaply. However, if $t_i$ needs to access memory that is attached to, say, CPU 1, then it must do so using a dedicated bus. This access can be substantially more expensive than accessing local memory, with some works reporting that cross-CPU memory accesses can be nearly twice as expensive as local memory accesses in terms of the number of required cycles [MG13]. Yet, we stress that the exact increase in cost depends on the access pattern of the underlying program; for instance, a program that primarily makes sequential memory accesses can typically take advantage of hardware prefetching to mitigate these issues. In our context, although the bulk of the memory accesses in G6K are sequential in nature (cf. [ADH+19, §5.3]), we note that accessing the underlying sieving database is done in an unordered fashion, and thus we would expect some NUMA-related effects to appear when G6K is deployed on a multi-processor system.

We now consider our own implementation. On the one hand, our implementation focuses on a multi-processor setting by default. Indeed, we note that the can avoid all NUMA-related performance issues by simply binding each process to a single physical CPU and disallowing explicit cross-CPU memory access. Yet, this manual separation of memory comes at a cost, as our implementation requires each process to explicitly engage in the exchange of data between CPUs. Moreover, our implementation serialises transfers lattice vectors by representing them in terms of their coefficient representation (see Section 4.1) and thus our implementation requires that some computation is carried out for each lattice vector. In other words, it is not clear *a priori* whether our implementation would outperform the original version of G6K in a single machine setting.

In order to quantify these overheads, we conducted two sets of experiments that compare our code to the original version of G6K. For both sets of experiments, we repeat each experiment three times and report the average. The results for each set of experiments can be found in Table 6b and Table 6a respectively. For both sets of experiments, we use 8 bucket batches with 4 auxiliary buffers for our distributed code.

- The first of these experiments is intended to capture the differences (if any) in wall-time between the original version of G6K and our code when controlling for NUMA effects. In these experiments we take $d \in \{116, 118, 120, 122\}$ and use node H (cf. Table 1) with 96 cores. We note that these cores are spread across two physical CPUs, and thus NUMA effects are likely to be visible in these experiments. For each $d$, we download the SVP challenge lattice in dimension $d$ (with seed $= 0$). Then, we use the BGJ1 implementation in the original version of G6K and our code respectively, recording the wall time for each experiment. In the case of the original version of G6K, we instantiate a single process with 96 cores. On the other hand, for our code we start two processes and bind each process to a single physical CPU i.e. each process is instantiated with 48 cores. We begin distributed sieving in dimension 90 and use the original version of G6K for all lower dimensions.
- The second of these experiments is intended to capture the overhead associated with using MPI. In these experiments we take $d \in \{90, 95, 100, 105\}$ and use node D (cf. Table 1) with 20 cores. Unlike node H, these cores are confined to a single physical CPU, and thus these experiments are not susceptible to any NUMA effects. These experiments follow the same format as described above i.e we execute a full sieve in dimension $d$ for both our code and the original version of G6K and record the results. For our distributed code, we bind two processes to the same physical CPU with 10 cores allocated to each

process and begin distributed sieving at dimension 80. By contrast, for the original version of G6K we assign all 20 cores to a single process.

We now discuss these results. As can be gleaned from both Table 6b and Table 6a, the wall times for the original version of G6K and our code are broadly the same. On the one hand, we note that the CPU-time of our implementation is consistently lower than that of the original version of G6K when running experiments in a NUMA aware context. These results indicate, at least in our particular setup, that there is a small benefit from running experiments in a NUMA aware context. Additionally, we observe that both the CPU and wall time are broadly similar in the context of a single CPU machine, too. Put differently, it appears that any overhead added by MPI in a single-machine setting is small relative to the other costs associated with sieving.

## D On the impact of pipelining

Our experiments in Section 5 critically depend on the number of batches that are in flight at once. Thus, in this section we provide experimental results that justify our choice of 8 batches and 4 auxiliary buffers. In order to establish a baseline, we use a single node of type K (cf. Table 1) with 14 threads and the original version of G6K to run a full sieve on the dimension 100 SVP Challenge lattice (with seed = 0). Then, we repeat this experiment with our code across 2 nodes of type K and 14 threads, varying the number of batches and auxiliary buffers. In the case of the two node experiments, we begin distributed sieving in dimension 90. We repeat each experiment three times and record the average wall and CPU time and compute the "parallel efficiency" relative to the baseline. The results for each experiment are tabulated in Table 7.

Before discussing these experiments, we remark that it would be unwise to extrapolate based on the data in Table 7. Indeed, we view these data points as indicative of how varying the pipelining parameters affects sieving for our particular configuration of nodes. Moreover, the sieving dimension in these experiments is rather small, and thus we caution the reader that the exact impact of pipelining may change as the dimension varies. However, in the context of these caveats there are several conclusions that we can make. First, notice that using no pipelining actually makes our distributed implementation *slower* than the baseline, despite using twice as many CPU cores. Notably, this decrease in performance is accompanied by an increase in CPU time, which we attribute to the time that each node spends waiting for network activity. Simply put, removing pipelining from our implementation appears to make parallelism so expensive as to remove any benefits that are granted by using more CPU cores. Yet, it is clear that this slowdown vanishes as the number of concurrent batches are increased; indeed, we see using any form of pipelining decreases the wall time relative to the baseline. This reduction broadly continues as the number of concurrent batches are increased, but the relative benefit diminishes as the number of batches increases. However, we observe that the CPU time does not actually decrease in the same manner as the wall time. First, we observe that the worst-case scenario for the CPU time is the setting where no pipelining at all is used, and thus we conclude that using some form of pipelining reduces the CPU time in a broad sense. On the other hand, the CPU time does not always decrease with the number of batches, with the minimum occurring with three batches and three buffers. Although we have no theoretical explanation for this behaviour, we speculate that the minor variance in CPU time is actually an implementation artefact that relates to consistency required to safely use collective operations in MPI, and thus we do not consider this behaviour to be indicative of a deeper pattern.

Table 2: Performance evaluation for BGJ1 in a homogeneous setting.

| Dim | 1-14 | 1-28 | 2-14 | 2-28 | 3-14 | 3-28 | 4-14 | 4-28 | 5-14 | 5-28 |
|---|---|---|---|---|---|---|---|---|---|---|
| 122 | 133h | 67.9h | 60.5h | 33.1h | 40.1h | 21.2h | 33.9h | 17.2h | 24.6h | 13.2h |
| | 1.00 | 0.97 | 1.10 | 1.00 | 1.10 | 1.04 | 0.98 | 0.96 | 1.08 | 1.00 |
| 120 | 72.0h | 42.2h | 34.4h | 18.3h | 24.9h | 14.4h | 19.8h | 10.0h | 19.7h | 9.15h |
| | 1.00 | 0.85 | 1.05 | 0.98 | 0.96 | 0.83 | 0.91 | 0.90 | 0.73 | 0.79 |
| 118 | 45.4h | 23.7h | 24.1h | 12.4h | 15.9h | 7.7h | 12.1h | 6.1h | 9.9h | 5.4h |
| | 1.00 | 0.96 | 0.94 | 0.91 | 0.95 | 0.99 | 0.94 | 0.92 | 0.92 | 0.83 |
| 116 | 25.2h | 14.5h | 11.9h | 6.6h | 8.6h | 5.2h | 7.0h | 3.9h | 5.7h | 3.2h |
| | 1.00 | 0.97 | 1.02 | 0.96 | 0.98 | 0.81 | 0.90 | 0.81 | 0.88 | 0.79 |
| 114 | 17.6h | 9.47h | 10.9h | 4.5h | 6.22h | 2.79h | 4.92h | 2.68h | 3.78h | 1.9h |
| | 1.00 | 0.93 | 0.81 | 0.98 | 0.94 | 1.05 | 0.89 | 0.82 | 0.93 | 0.91 |

BGJ1 sieving using identical machines (node K in Table 1) over a 10Gbps network. "Dim" gives the sieving dimension, "(N-C)" indicates "N" nodes and "C" cores per node. The first row for each dimension gives wall times, the second gives the normalised speed-up relative to the baseline of one machine and 14 core. Here, 1.00 is ideal.

Table 3: Heterogeneous performance evaluation for BGJ1.

| Dimension | Wall time | CPU time | Memory usage (GiB) | | | | "Parallel |
|---|---|---|---|---|---|---|---|
| | | | H | S | A | total | efficiency" |
| 128 | 102 hours | 583 days | 194 | 70 | 56 | 320 | 0.973 |
| 127 | 82 hours | 380 days | 184 | 67 | 54 | 305 | 0.892 |
| 124 | 34 hours | 202 days | 117 | 43 | 34 | 194 | 0.958 |

BGJ1 sieving using 3 machines (nodes S, H, and A in Table 1) over a 1Gbps network with a total of 159 cores. "Dimension" gives the sieving dimension. "Parallel efficiency" roughly compares with running the most recent version of G6K [ADH+19] on node H. A value of 1.0 is ideal under this metric, see Section 5. Memory on individual nodes is estimated, total memory was measured.

Table 4: Heterogeneous performance evaluation for BGJ1 and BDGL.

| | Wall time | | CPU time | | Total Memory | |
|---|---|---|---|---|---|---|
| Dimension | BDGL | BGJ1 | BDGL | BGJ1 | BDGL | BGJ1 |
| 105 | 2.13h | 1.91h | 6.91h | 21.2h | 11GiB | 15GiB |
| 100 | 0.70h | 0.68h | 2.23h | 6.64h | 5GiB | 8GiB |

Results for BDGL/BGJ1 sieving using 3 machines over a 1Gbps network with a total of 60 cores (20 per machine). "Dimension" gives sieving dimensions, Timing were gathered on nodes S, H, and A, see Table 1.

Table 5: Performance evaluation for BDGL in a homogeneous setting.

| Dim | 1-14 | 1-28 | 2-14 | 2-28 | 3-14 | 3-28 | 4-28 |
|---|---|---|---|---|---|---|---|
| 114 | 4.4h | 3.2h | 4.9h | 6.4h | 4.5h | 6.0h | 3.1h |
| 112 | 2.9h | 2.1h | 3.4h | 3.7h | 3.1h | 3.4h | 2.9h |
| 110 | 2.0h | 1.4h | 2.0h | 2.3h | 1.8h | 2.5h | 2.0h |

BDGL sieving using identical machines (node K in Table 1) over a 10Gbps network. "Dim" gives the sieving dimension, "(N-C)" indicates "N" nodes and "C" cores per node. For each dimension we give wall times.

Table 6: Performance evaluation for BGJ1 on a single machine.

(a) Single-CPU setting

| Dim. | Conf. | Wall time | CPU time |
|---|---|---|---|
| 105 | G | 1.57 hours | 31.3 hours |
| 105 | D | 1.63 hours | 31.3 hours |
| 100 | G | 23.3 minutes | 7.7 hours |
| 100 | D | 27.1 minutes | 8.9 hours |
| 95 | G | 6.81 minutes | 2.25 hours |
| 95 | D | 7.71 minutes | 2.49 hours |
| 90 | G | 1.6 minutes | 0.5 hours |
| 90 | D | 2.3 minutes | 0.72 hours |

Using node D (cf. Table 1). All experiments used 20 cores and all timings are the average of three runs. Configuration "G" refers to timings gathered using G6K, whereas Configuration "D" refers to timings gathered using our code.

(b) Multi-CPU setting

| Dim. | Conf. | Wall time | CPU time |
|---|---|---|---|
| 122 | N | 31.6 hours | 117 days |
| 122 | G | 31.6 hours | 124 days |
| 120 | N | 18.7 hours | 68.2 days |
| 120 | G | 18.8 hours | 73.7 days |
| 118 | N | 11.4 hours | 42.8 days |
| 118 | G | 11.4 hours | 44.6 days |
| 116 | N | 6.1 hours | 22.4 days |
| 116 | G | 6.5 hours | 25.3 days |

Using node H (cf. Table 1). All experiments used 96 cores and all timings are the average of three runs. Configuration "G" refers to timings gathered using G6K, whereas Configuration "N" refers to timings gathered using our code.

BGJ1 sieving for our code and G6K on a single machine.

Table 7: Performance evaluation for different numbers of batches in BGJ1.

| Conf. | Wall time | CPU time | “Parallel efficiency” |
|---|---|---|---|
| $(5,5)$ | 14.6 minutes | 6.1 hours | 0.87 |
| $(4,8)$ | 15.0 minutes | 6.4 hours | 0.84 |
| $(4,4)$ | 15.1 minutes | 6.3 hours | 0.84 |
| $(3,3)$ | 15.1 minutes | 6.0 hours | 0.84 |
| $(2,2)$ | 16.7 minutes | 6.1 hours | 0.76 |
| $(1,1)$ | 28.4 minutes | 7.1 hours | 0.45 |
| Baseline | 25.4 minutes | 5.9 hours | 1.00 |

BGJ1 sieving using node K (cf. Table 1) for our code with varying numbers of batches and auxiliary buffers. For each entry $(N, M)$, $N$ refers to the number of auxiliary buffers and $M$ refers to the number of batches that are in flight at any given time. “Baseline” refers to the time taken to run the dimension 100 full sieve using the original version of G6K.