Exploring parallel enumeration algorithms to improve efficiency to solve the SVP

Helder Goncalves
heldergoncalves92@gmail.com
and Joao Magalhaes
jpfmm123@gmail.com

Artur Mariano
Technical University of Darmstadt
artur.mariano@sc.tu-darmstadt.de

Alberto Proenca
University of Minho
aproenca@di.uminho.pt

***Abstract*—Lattice-based cryptography methods is well known primitive from the asymmetric cryptographic primitives set. Although widely studied for decades they have gained new notoriety in the last years because it is believe in the field of cryptography that these methods can withstand attacks by upcoming quantum computers.**

This paper describes how we optimized the Enumeration algorithm's serial code and then parallelized it using *PThreads*. The results are shown in the results section being very promising and showing that we can obtain significant performance improvements over the said implementation.

Keywords*—*Lattice, SVP, Enumeration, Parallelization, Pthreads.

I. Introduction

Lattice-based cryptography methods is well known primitive from the asymmetric cryptographic primitives set. Although widely studied for decades they have gained new notoriety in the last years because it is believe in the field of cryptography that these methods can withstand attacks by upcoming quantum computers. A lattice in R^n is a discrete subgroup of R^n which span a real vector space R^n. Every lattice in R^n can be generated from a basis for the vector space by forming all linear combinations with integer coefficients. A lattice may be viewed as a regular tiling of a space by a primitive cell.

Lattices can be applied in many contexts but it has gained importance in cryptography due to advent of quantum computing. The known public key cryptography is easily attacked by this new generation of computers. In contrast, the lattice-based cryptography is believed to be resistant to attacks from quantum computers. In cryptography NP-hard algorithms are especially attractive due to their exponential time complexity. New cryptographic approaches can be reduced to the shortest vector problem (SVP) which is known to be NP-hard making it a prime candidate to develop said approaches.

A given basis Λ constituted by a set of totally independent linear vectors. Although a lattice can be an unbounded set of vectors, in this paper we only consider square basis, i. e. the set size is bound by the number of basis vector such that each vector corresponds to a dimension.

Experimental testing of the algorithms properties may shed some light on its vulnerability. However the experimental results must be obtained within reasonable time envelops, which implies efficient parallel implementations of the Enumeration algorithm able to tackle not only conventional architectures such as *Intel X86*.

In order to increase the development performance, ensure reliability and fulfill time constrains we resort to the external library NTL. NTL encapsulates decades of performance tuning by others and provides reliability assurances that could only by achieved at the expense of large amounts of development time.

The first step requires the computation of Gram–Schmidt coefficients which then will be used by the Enumeration algorithm. In order to improve the sequential implementation of the Enumeration algorithm we used a two step approach:

i) optimization of the sequential code and its data structures
*ii)*perform a guided parallel based on the results obtain on the previously state

As the results actual chooses the two step described in this paper allowed us to achieved linear and/or even super-linear speedups in some cases.

Section I is the introduction to the problem. In section II has the strategy used to tackle the problem. Section III is about the optimization of the sequential version of the enumeration algorithm. Section IV is about the parallelization of the enumeration algorithm. Section V is where the results of the parallelization are shown. Section VI briefly talks about the problems of adapting the code to *CUDA* and *MIC*. Sections VII and VIII are the future work and the conclusion of our work.

II. Strategy

A. Data storage

Due to the huge size of the data input, it would be expensive to store it in memory thus to avoid the problem we store only the reduced basis which is computed by the Lenstra-Lenstra-Lovász (LLL) algorithm from the NTL library.

The NTL is a high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers, and for vectors, matrices, and polynomials over the integers and over finite fields.

B. Pre-Processing stage

The enumeration algorithm requires the orthonormalization of the reduced basis, which is achieved through a Gram-Shmidt method.

It is a method for orthonormalising a set of vectors in an inner product space, most commonly the Euclidean space R^n. The Gram–Schmidt process takes a finite, linearly independent set $S = v1, ..., vk$ for $k \leqslant n$ and generates an orthogonal set $S' = u1, ..., uk$ that spans the same k-dimensional subspace of R^n as S, i.e. as prime is the set of orthogonal basis vectors of the inner product space of S.

The S prime set is stored as a triangular matrix μ and a vector containing the squared norms although this routine is a prime candidate for vectorization, since it is only executed once with a small time footprint over the entire overall execution time such an approach yields little benefit.

C. Enumeration

The lattice can be seen like a hypersphere. In such a scenario the enumeration process consists of enumerating the lattice inner points while progressively decreasing the hypersphere radius as the points are enumerated. From another perspective this model can be seen as a tree transversal, which is easier to visualize and allows easier identification of parallelization opportunities.

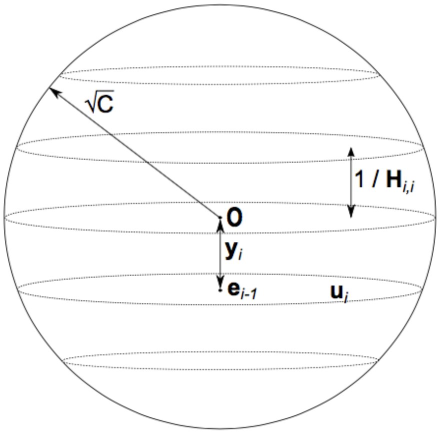


Figure 1: Hypersphere representation and its sub-layers

In Figure 1, $\sqrt{C}$ is the norm of the shortest vector found so far, representing its radius. The hypersphere can be subdivided in layers of lower dimensions where the instant layer is denominated by i. Layer i is divided in more sub-divided layers in the dimension $i - 1$.

The algorithm searches a solution in all layers by starting to walk the tree at layer n and finishing when revisits that layer. Walking the tree implies two methods *moveUp* and *moveDown*.

- **MoveDown** - Visits the layer with a lower dimension.
- **MoveUP** - Visits the layer with a higher dimension.

If at any moment the algorithm reaches dimension 0, then a smaller vector exists. In such case the current smallest vector is replaced by the newly formed.

D. Pseudo-Code

To easily understand how the sequential version of the Enumeration algorithm works, we have a draft of our implementation in pseudo-code, based on Schnorr's work [1].

```
//Start by Initializing all the variables

while( t <= End_Dimension ){
    //Compute cT[i]
    if( cT[t] < current_Norm ){
        if( t > 0 )
            moveDown();
        else
            updateVector();
    }else
        moveUp();
}
```

E. Profiling

In order to identify hotpots in the code we used gprof. Table I shows the profiler results, namely the number of calls and the percentage of overall time for each of the code functions. The profiling took 404.70 seconds longer.

Function	Time (%)	N^o of Calls
ENUM	98,5	1
moveDown	67,6	3159878635
moveUp	10	3159878580
updateVector	0,7	-

Table I: GPROF profiling results for a basis dimension of 55 and $-O3$ optimization

From table *I* it is trivial to conclude that *moveUp* and *moveDown* have an extremely high number pf function calls and most importantly *moveDown* accounts for the majority of the execution time.

III. Sequential enumeration algorithm optimization

After a careful examination of the *C* code and the compiler generated *Assembly* code several hotspots were identified.

1) Power Decomposition: The first one was the decomposition of the power function. The power function is a very heavy function and must be avoid when the exponent is always the same, and that was the case $(a + b)^2$. So, to do the decomposition we applied some math rules.

$$(a + b)^2 = a^2 + 2ab + b^2 \tag{1}$$

$$(a + b)^2 = (a + b) \times (a + b) \tag{2}$$

To solve this problem there are two possible solutions. The first one is slower than the second one. This happens because to compute the first solution it is necessary to compute at least 4 multiply operations and 2 sum operations. In contrast, to compute the second option just is necessary compute 1 sum operation and a multiply operation. This is possible because is done the sum to a local variable and then the value multiply itself.

2) Avoid Computation: A careful examination of the usage of $cT[t]$ revealed we could postpone its computation to the moment exactly prior to its use.

3) Inline: As gprof reported, the functions *moveUp* and *moveDown* are constantly called which implies all the necessary operations such as stack creation, returning, ... To avoid function call overhead, both funcions were inlined which not only removes the call overhead but also increases the compiler opportunities for automatic optimization.

Now to try to increase the performance of the current algorithm, were tried greater depth implementations.

4) Memoizing: In most cases performance improvement has a careful balancing between computations cost and memory access latency. In this particular application we exploited a memoization technique for $y[t]$ and empirically determined that it was not worth while paying the cost of memory access.

Although it seems a good practice, it got worst results because this computation is small enough to not provide a best result. That happens because computing it is faster than getting it from the memory.

5) Map-Reduce: Map-Reduce is a technique in which we map the problem into different sub problems and then reduce it all to have the final result. In function *moveDown* we saw an opportunity to vectorize a *for* cycle, taking advantages of the possibility of computing various values at the same time. In this case we had a problem because in every cycle iteration we would store the value in the same memory location. We tried to apply the Map-Reduce concept in this case by creating an array of values, dividing the iterations between those values and then sum all the values of that array, which meant we could vectorize the first cycle. Unfortunately, the last dependency was very hard to turn around. Although the increase achieved on vectorization was a great advantage, it is lost on the second cycle when the sum was done.

A. *Improvement Results*

Table III show us the results of tests made for a 55 basis dimension. After all the tests, the *Inline* version has the best speed up. Despite attempts to advanced improvements, we could not achieve better results than the *Inline*, for the reasons explained above.

Test Version	Time (seconds)	SpeedUp
Source	407,70	1,00
Inline	225,23	1,80
Save & Re-use	274,54	1,47
Map-Reduce	240,09	1,69

Table II: Improvements tests results

B. *Cache misses*

The average cache misses are an important measure of application performance. Figure 2 shows the percentage of instruction and data misses for dimension 55.

The total values can be see in table III.

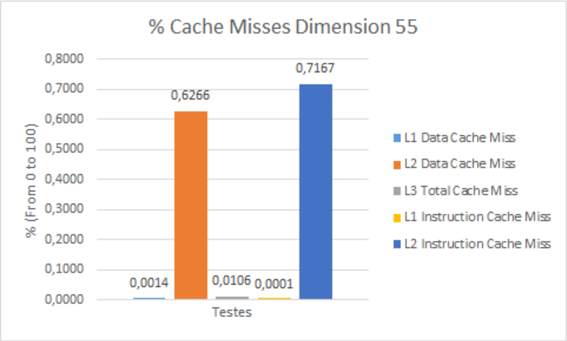


Figure 2: Cache miss percentage for the inline version, dimension 55

Counter	Value
PAPI_LD_INS	369695637512
PAPI_ST_INS	614658154544
Total instructions	984353792056
PAPI_L1_DCM	13648217
PAPI_L2_DCM	85526
PAPI_L3_TCA	131845
PAPI_L3_TCM	14
PAPI_L1_ICM	469907
PAPI_L2_ICM	3368
PAPI_FP_OPS	224287468034
Time(s)	199
FLOPS	1127072704
Operational Intensity	1257893,643

Table III: Cache miss values for the inline version, dimension 55

As we can see in Figure 2, we have a very low percentage of cache misses, with the maximum value being 0.72% for the L2 instruction cache miss. The operational intensity is measured by the number of floating point operations per second divided by the number of bytes that was transferred by the memory. In this case we have more than 200 billion floating point operations and 199 seconds, so we have about 1 billion floating operations per second. Since we only have 14 times 64 bytes transferred from memory, we have an operational intensity of more than 1 million. This definitely means that we have a compute bound problem. Since we have a compute bound problem, we do not need to restructure the data to make it more memory friendly.

IV. Paralellization

The paralellization of the enumeration algorithm is very important, since it is possible to obtain linear, or even super linear speedups up to 20 threads, as can be seen in the results section. To parallelize the algorithm, Pthreads was used since it has lower overhead thread creation than OpenMP.

A. *Before paralellization - preparation of the code*

To divide the enumeration algorithm, the code must be prepared for paralellization. For that a new structure was created, called Enum. This structure contains the following two short values:

- bound - Responsible of telling the task in which dimension it must start and stop, not going further.

- sibling - Used to tell the task what is its sibling (see subsection IV.C).

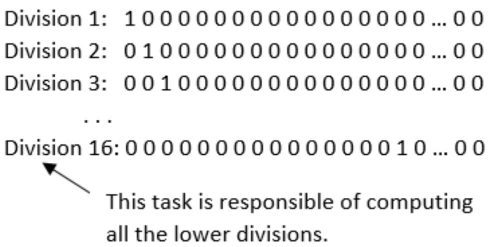
This structure also has an array of short values, called *vec*. This array can either be null or of size *VectorLength* (see subsection IV.D). The level is used to tell the task how much values the array has. Finally, this structure also has a variable called type, that defines the behaviour of the task. The possible behaviours of the task are the following:

- 0 - the task behaves like the serial algorithm.
- 1 - the task behaves like on type 0 but has a fixed bound, that can not be exceeded.
- 2 - this type is part of the siblings division and it runs until its bound plus one (see subsection IV.C). This type only occurs when the sibling is 2.
- 3 - this type is like type 2 but it must stop in its bound (see subsection IV.C).
- 4 - this type is like type 2 but, instead of running until its bound plus one, the dimension that the task stops running must be calculated (see subsection IV.D). Like type 2, this type only occurs for sibling 2.

A structure to handle all the Enum structures was created. This structure (called LEnum) is a linked list of Enum structures and its accessed by the threads when they are created or when they complete a task. When a thread requests a new task, that task is removed from LEnum. If no tasks are left in the linked list, then the thread will wait until all threads end their work to end the program.

B. First attempt - Blindly divide the dimensions

The first attempt to parallelize the code, consisted of blindly divided the dimensions. Each dimension has an associated task, in which we disregard the values of bigger dimensions and apply the algorithm starting from a given dimension (type 1 division). This division is done until a special dimension is reached, defined by the variable *MAX_DEPTH* and after that all dimensions are computed as part of the same task. In the Figure 3, the divisions made for a lattice with dimension 50 and *MAX_DEPTH* equal to 35 can be seen:



Division 1: 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0
Division 2: 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0
Division 3: 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0
. . .
Division 16: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 ... 0 0

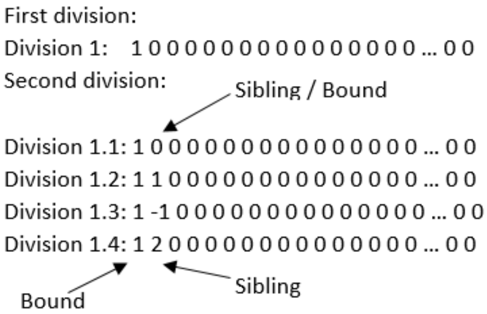
This task is responsible of computing all the lower divisions.

Figure 3: First parallelization division

As can be seen, there will be 16 total divisions. Each task is in charge of a given dimension and the task in charge of dimension 35 computes all the lower dimensions. It was done this way because lower dimensions have much less computation than bigger ones.

C. Second attempt - Division by siblings

After testing the first paralellization attempt, it was discovered that the biggest dimensions would take a lot of time to compute and the code would become sequential at some point. For example, if for a lattice with 50 dimensions and even if the first task to be run was the one associated with dimension 50, it would be the last to finish making the program scale poorly. To improve the parallelization, it was decided to divide the biggest dimensions by siblings, limited by the variable *divRange*. These siblings improve the performance of the program a lot, by further dividing the previous tasks into four each. The siblings 0, 1 and -1 behave the same way (type 3), but sibling 2 behaves in a special way (type 2), in which the bound of the task is equal to the others plus one. The way it works is really simple, imagine if we had a previous task for dimension 50. If we use siblings, we are forcing the value associated with dimension 50 to be 1, and the value for dimension 49 is set with the value of the sibling.



First division:
Division 1: 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0
Second division:
Division 1.1: 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0
Division 1.2: 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0
Division 1.3: 1 -1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0
Division 1.4: 1 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ... 0 0

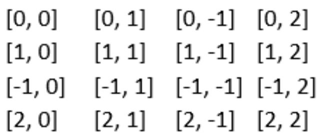
Figure 4: Second parallelization division

D. Final parallelization

After the second parallelized algorithm was tested, it was noticed that the tasks in charge of sibling 0 were still taking too much time to complete. To resolve this problem, further division of the tasks was obtained by creating vectors that force a given number of values (which varies with the variable *VectorLength*) to be fixed. After applying this vector, siblings were use to divide the tasks even more. To avoid wasting time in the generation of these vectors, they are all generated once and put into a matrix. After this matrix is generated, an iteration is made from the number of dimensions until the variable *CreateRange*, creating a task for each sibling of each vector of each dimension. After this creation, the matrix containing the vectors is free'd. With the division done this way, there were much more divisions than before and the tasks were more fine-grained. A new type of behaviour similar to type 2 (in which tasks would run until the dimension defined by its bound plus one) was used to make the algorithm run all the possible combinations. In this new type (type 4) the dimension where the task stops is calculated by the number of sequential array positions that have the number 2.

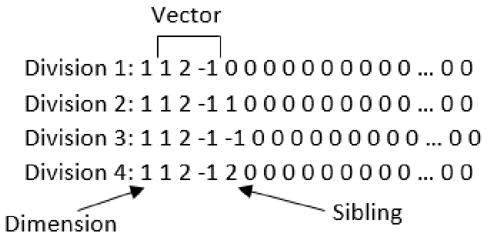
Figure 5 has a small example of vector generation for *VectorLength* equal to 2:

For each of these vectors, there are four divisions, corresponding with the four siblings being used, as can be seen in Figure 6, for vector [1, 2, -1]:



[0, 0]	[0, 1]	[0, -1]	[0, 2]
[1, 0]	[1, 1]	[1, -1]	[1, 2]
[-1, 0]	[-1, 1]	[-1, -1]	[-1, 2]
[2, 0]	[2, 1]	[2, -1]	[2, 2]

Figure 5: Vector generation



Division 1: 1 1 2 -1 0 0 0 0 0 0 0 0 0 0 ... 0 0

Division 2: 1 1 2 -1 1 0 0 0 0 0 0 0 0 0 ... 0 0

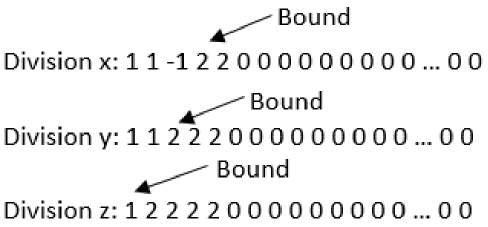
Division 3: 1 1 2 -1 -1 0 0 0 0 0 0 0 0 0 ... 0 0

Division 4: 1 1 2 -1 2 0 0 0 0 0 0 0 0 0 ... 0 0

Figure 6: Final parallelization

The vectors actually have one more position, which is filled with the sibling and used to compute it more easily.

The difference of behaviour for type 4 can be shown in Figure 7.



Division x: 1 1 -1 2 2 0 0 0 0 0 0 0 0 0 ... 0 0

Division y: 1 1 2 2 2 0 0 0 0 0 0 0 0 0 ... 0 0

Division z: 1 2 2 2 2 0 0 0 0 0 0 0 0 0 ... 0 0

Figure 7: Variation of the bound with different vectors

As can be seen, the bound of type 4 is equal to the last position of the *vec* array with value 2 (from right to left), except when all the values of the array are 2, in which case the bound is equal to the dimension.

After all types of divisions were defined, a combination of them was used to try to maximize the performance of the algorithm by improving the way the division is made. In Figure 8, the way divisions are defined and for what dimensions can be seen:

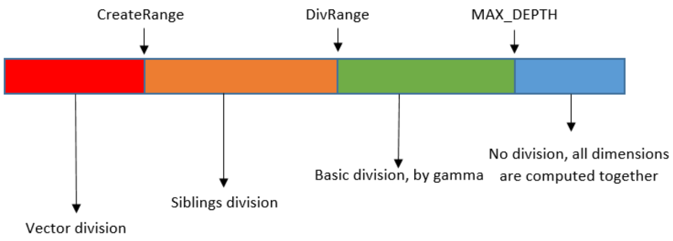
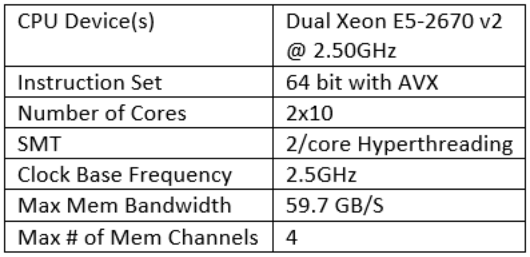


Figure 8: Final divisions

V. Results

A. TestBed

The algorithm was tested on nodes 652-1 and 652-2 of the search6 cluster. Every test took the entire node, so the performance would not be affected by another program or by context switches. The specification for this two nodes can be seen in Figure 8.



CPU Device(s)	Dual Xeon E5-2670 v2 @ 2.50GHz
Instruction Set	64 bit with AVX
Number of Cores	2x10
SMT	2/core Hyperthreading
Clock Base Frequency	2.5GHz
Max Mem Bandwidth	59.7 GB/S
Max # of Mem Channels	4

Figure 9: Specifications for nodes 652-1 and 652-2 of the Search6 cluster

B. Division variables

For this test, the variables of the division were the following:

- *CreateRange*: this variable was set as equal to the number of processors running the algorithm. For example, for 16 processors running for a lattice with dimension 50, then all dimensions from 50 to 35 would be divided with vectors.
- *DivRange*: set to 60% of the lattice dimension.
- *MAX_DEPTH*: set to 40% of the lattice dimension.
- *VectorLengh*: In this test, a vector length of 4 (including the position of the vector reserved to the sibling) was used.

These numbers are not optimal, by far. We used them by trial and error for a specific lattice (with dimension 50) and then we applied them to all the lattices in this test.

C. Test results

In appendix A, the results for this test can be seen. Tests were made for seeds 0 and 15 for lattices with dimensions 50, 55, 58, 60 and 65.

With dimension 50 the speedups are constants and almost linear as it increases the number of threads what is good. On the next dimensions (55, 58 and 60) the parameters are almost perfect, especially on basis 55, having linear speedups. But are on basis 60 (seed 15) a super-linear speedup is achieved with 24 times best for 20 threads. However, the seed 0 speedup is worst. This difference between seeds is explained by the way how our implementation crosses the tree. The solution of the problem also can be on dimension 20 as can be on dimension 50. Now, if the final solution is found at the beginning of computation, the algorithm will have less sub-trees to visit than if it is found at the end, what will make it more slow.

As it was said earlier, the biggest problem of our implementation is that the numbers of divisions created are not optimal for each one of the different basis. This is clear for the speedup of base 65, specially for seed 0, that only improves for two threads as expected and after that the speedup is really bad. This happens because there are few divisions for the complexity of the computation of this dimension. A curious problem in the seed 0 was the fact of it actually takes more time to find the solution with 20 threads than it takes with 16. That happens because when it execute with 20 threads are created a bigger set of siblings with vectors and increase memory used what causes memory access issues.

The way that the parallelization is made, the speedup really depends of the seed that is used. This is because when there is a task that is created with the vector division, first it is checked if the combination of values is possible for that lattice and only then computed if the task can provide a better solution so, if there is a seed that has a lot of tasks that do not need to be computed, augmenting the number of tasks can have a lower impact on speed than in the case where there is a need compute much more tasks.

VI. CUDA and MIC

To help improve the performance of this algorithm, the usage of CUDA and MIC systems would probably help a lot but the code would need some major changes for that to be possible.

In the CUDA case this code lacks opportunities for vectorization, so the algorithm would have to be restructured to make the CUDA implementation possible.

In the MIC case, the implementation would be really difficult since the parallelization is implemented in Pthreads and the systems like Xeon Phi are not compatible with it. So, to implement this code in Xeon Phi the way the parallelization was made would have to chenge to OpenMP. When finally there was all the different ways to divide the work defined and tests started, there was no time to rewrite the code to OpenMP to implement it on Xeon Phi.

VII. Future work

After the work that was done in our project, there is great opportunity to further improve the performance of the algorithm.

For example, instead of the division by vectors being made just for the number of dimensions equal to the number of threads, there must be a mathematical formula that will optimize the time the algorithm takes to run by varying the number of dimensions divided by vectors. This is true not only for variable *CreateRange* but also for variables *DivRange*, *MAX_DEPTH* and even *VectorLength*.

Another way to further improve the algorithm is by applying a sieving optimization like extreme pruning to discard some of the branches, making the algorithm much faster. For that there was a need to resort to CUDA do compute the bounding function of the lattice and then compare the norm of each vector with the result of the bounded function instead of the current lowest vector norm.

VIII. Conclusion

In this project we managed to parallelize the enumeration algorithm in a way that linear or even super linear speedups can be achieved, specially in the case of base 60 seed 0 in which we had a speedup of 25 times for 20 threads. In spite of this results, a lot of work can be done to further improve the algorithm, as discussed in the future work section.

The main problem we had in this project was understanding how the code works. This took a really long time and it was only managed to do it with help of our advisors. The way the auxiliary variables had to be created / altered to enable the parallelization was something that made us loose a lot of time too, since for every new division type, the way creation and assigning of values to the auxiliary variables was made changed.

Acknowledgments

We would like to thank our advisors for all the help they gave us and for always being available for all the questions we had. We would also like to thank Fábio Correia for all the help he gave us in understanding how the code works and what problems with the optimization attack first since they were the most important.

References

[1] M. Euchner C. P. Schnorr. "Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems." In: *Mathematical Programming* 66.1-3 (1994), pp. 181–199. DOI: http://link.springer.com/article/10.1007/BF01581144.

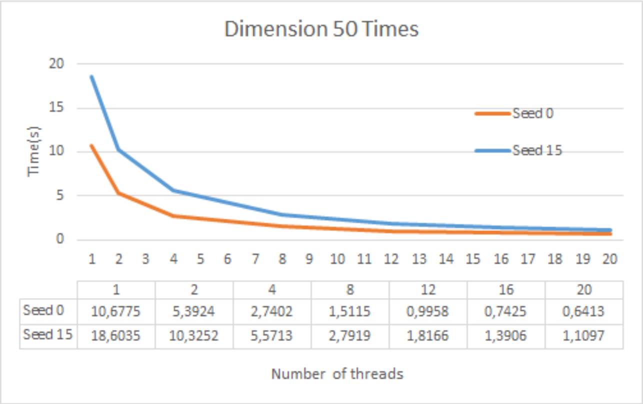
[2] Fabio Correia. "Assessing the Hardness of SVP Algorithms on Multi-core CPUs." In: (2014). DOI: http://link.springer.com/chapter/10.1007/11792086_18.

[3] Michael Schneider Özgür Dagdelen. "Parallel Enumeration of Shortest Lattice Vectors." In: *Algorithmic Number Theory* 6272 (2010), pp. 211–222. DOI: http://link.springer.com/chapter/10.1007/978-3-642-15291-7_21.

[4] Damien Stehlé Phong Q. Nguyen. "LLL on the Average." In: *Algorithmic Number Theory* 4076 (2006), pp. 238–256. DOI: http://link.springer.com/chapter/10.1007/11792086_18.

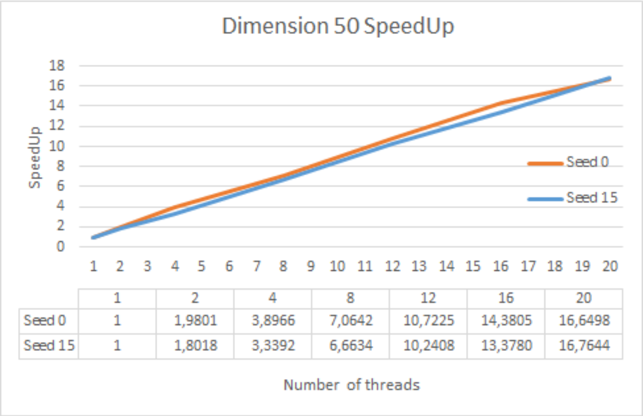
APPENDIX A
TEST RESULTS

Results and speedups for seeds 0 and 15 for dimensions 50, 55, 58, 60 and 65. The division variables are: *CreateRange* set to dimension - number of threads; *DivRange* set to 60 per cent of the dimension; *MAX_DEPTH* set to 40 per cent of the dimension and *VectorLength* set to 4 (including the reserved index for the sibling).



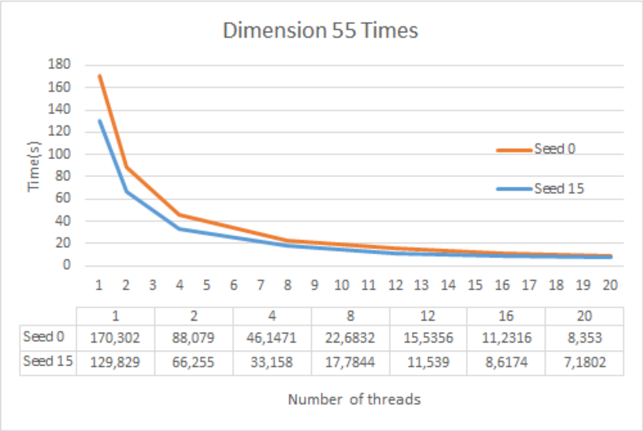
	1	2	4	8	12	16	20
Seed 0	10,6775	5,3924	2,7402	1,5115	0,9958	0,7425	0,6413
Seed 15	18,6035	10,3252	5,5713	2,7919	1,8166	1,3906	1,1097

Figure 10: Dimension 50 Times



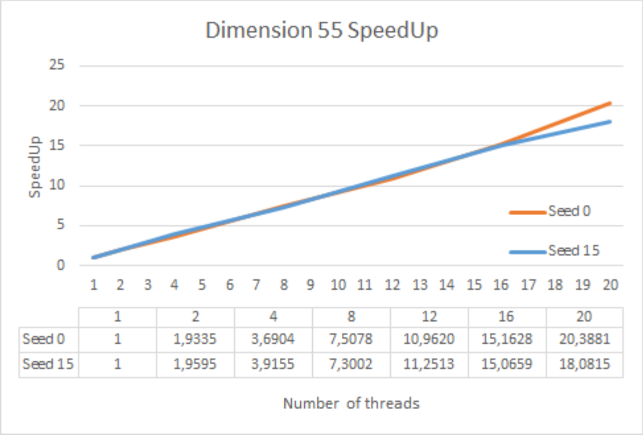
	1	2	4	8	12	16	20
Seed 0	1	1,9801	3,8966	7,0642	10,7225	14,3805	16,6498
Seed 15	1	1,8018	3,3392	6,6634	10,2408	13,3780	16,7644

Figure 11: Dimension 50 Speedup



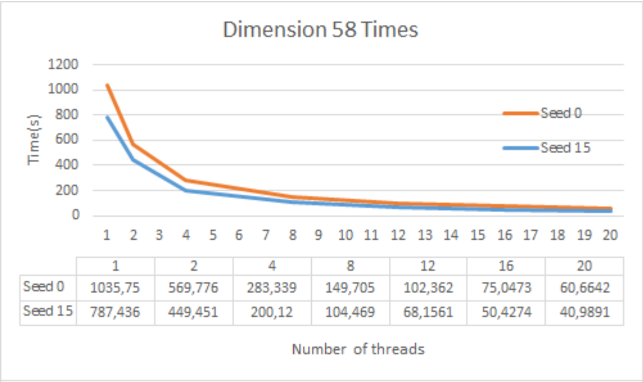
	1	2	4	8	12	16	20
Seed 0	170,302	88,079	46,1471	22,6832	15,5356	11,2316	8,353
Seed 15	129,829	66,255	33,158	17,7844	11,539	8,6174	7,1802

Figure 12: Dimension 55 Times



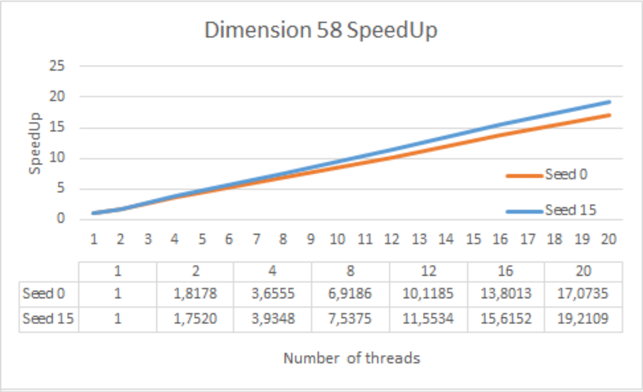
	1	2	4	8	12	16	20
Seed 0	1	1,9335	3,6904	7,5078	10,9620	15,1628	20,3881
Seed 15	1	1,9595	3,9155	7,3002	11,2513	15,0659	18,0815

Figure 13: Dimension 55 Speedups



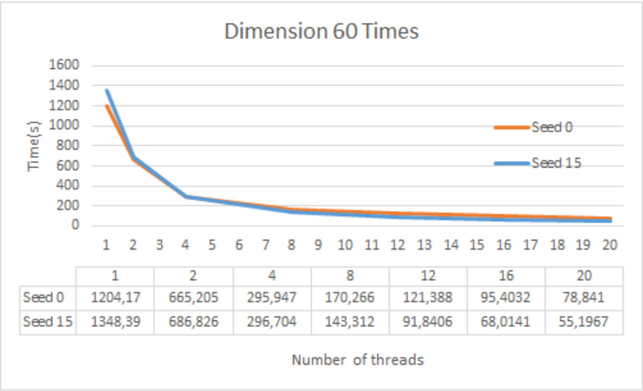
	1	2	4	8	12	16	20
Seed 0	1035,75	569,776	283,339	149,705	102,362	75,0473	60,6642
Seed 15	787,436	449,451	200,12	104,469	68,1561	50,4274	40,9891

Figure 14: Dimension 58 Times



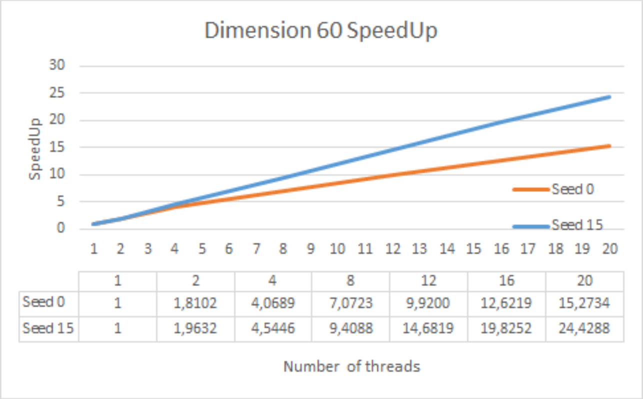
	1	2	4	8	12	16	20
Seed 0	1	1,8178	3,6555	6,9186	10,1185	13,8013	17,0735
Seed 15	1	1,7520	3,9348	7,5375	11,5534	15,6152	19,2109

Figure 15: Dimension 58 Speedups



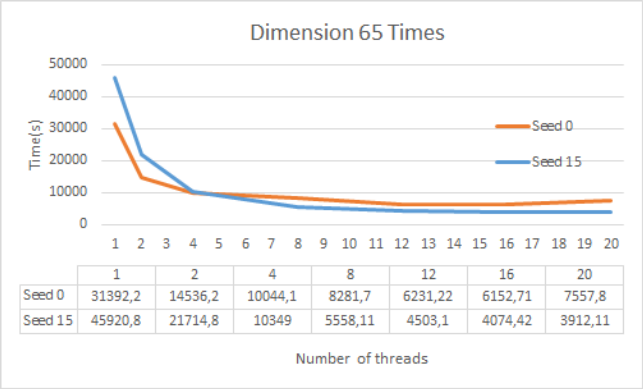
	1	2	4	8	12	16	20
Seed 0	1204,17	665,205	295,947	170,266	121,388	95,4032	78,841
Seed 15	1348,39	686,826	296,704	143,312	91,8406	68,0141	55,1967

Figure 16: Dimension 60 Times



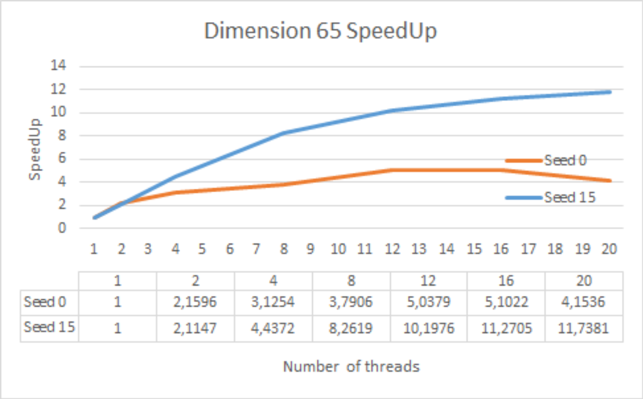
	1	2	4	8	12	16	20
Seed 0	1	1,8102	4,0689	7,0723	9,9200	12,6219	15,2734
Seed 15	1	1,9632	4,5446	9,4088	14,6819	19,8252	24,4288

Figure 17: Dimension 60 Speedups



	1	2	4	8	12	16	20
Seed 0	31392,2	14536,2	10044,1	8281,7	6231,22	6152,71	7557,8
Seed 15	45920,8	21714,8	10349	5558,11	4503,1	4074,42	3912,11

Figure 18: Dimension 65 Times



	1	2	4	8	12	16	20
Seed 0	1	2,1596	3,1254	3,7906	5,0379	5,1022	4,1536
Seed 15	1	2,1147	4,4372	8,2619	10,1976	11,2705	11,7381

Figure 19: Dimension 65 Speedups