On the parallelization approaches for Intel MIC architecture

E. Atanassov, T. Gurov, A. Karaivanova, S. Ivanovska, M. Durchova, and D. Dimitrov

Citation: AIP Conference Proceedings **1773**, 070001 (2016); doi: 10.1063/1.4964983
View online: http://dx.doi.org/10.1063/1.4964983
View Table of Contents: http://aip.scitation.org/toc/apc/1773/1
Published by the American Institute of Physics

On the Parallelization Approaches for Intel MIC Architecture

E. Atanassov[a)], T. Gurov, A. Karaivanova, S. Ivanovska, M. Durchova and D. Dimitrov

Institute for Information and Communication Technologies, Bulgarian Academy of Sciences, Acad. G. Bonchev str., Block 25A, Sofia 1113, Bulgaria

[a)]Corresponding author: emanouil @ parallel.bas.bg

Abstract. The Intel MIC architecture is one of the main processor architectures used for the production of computational accelerators. Increasing energy and cost-efficiency of accelerators is one important option for building new HPC systems. However, the effective use of accelerators requires careful optimization on all stages of the algorithm and use of appropriate parallelization approaches. In the domain of statistical methods the quasi-Monte Carlo methods present distinct challenges when thousands of computational cores are to be involved in a computation. In this paper we describe in detail and study the performance of algorithms for generating some popular low-discrepancy sequences, aimed at devices with Intel MIC architecture. By leveraging the powerful vector instructions of the Intel MIC architecture to process many coordinates of the sequences in parallel, we obtain fast implementations that can be plugged-in in any parallel quasi-Monte Carlo computation. We present extensive numerical and timing results that demonstrate the benefit of our algorithms and their parallel efficiency. The effects of using hyperthreading are also studied. The generation routines are provided under the GPL.

INTRODUCTION

Because of the increased costs and difficulties in providing the necessary electrical power for HPC computing, there is an increased interest in deploying heterogeneous systems that include accelerators like GPU cards or Intel Xeon Phi coprocessors. However, achieving the maximum possible efficiency when using these accelerators is a non-trivial task and requires substantial effort during programming and testing. In this paper we concentrate on the Intel Multiple Integrated Cores (MIC) architecture, which is used in the Intel Xeon Phi coprocessors. In the future it is expected that such devices will also serve as CPUs, but in our system they are deployed as coprocessors in servers with regular CPUs. They have relatively large number of cores running at low frequencies and are equipped with vector units that can process several numbers at once. Both integer and floating point operations are available in vector mode, with substantially higher throughput than the sequential versions. More information about programming for these devices can be obtained from [1],[2]. The high energy and cost efficiency of using accelerators was our motivation to build the Avitohol High Performance System by using Xeon Phi 7120P accelerators. The system consists of 150 servers SL250S each with dual Xeon CPU E5-2650 v2 at 2.60GHz and dual Xeon Phi 7120P accelerator cards. In total the system has 9600 GB RAM accessible by the regular CPUs and 4800 GB RAM on the accelerator cards. The operating system on the servers is Red Hat Enterprise Linux, while the accelerators run their own special Linux OS, provided by Intel, which is part of the MPSS. The exact versions that were deployed during our tests, where 6.7 on the servers and 3.6-1 for the MPSS. It must be noted that this system reached 332th place in the Top 500 list when it started operations [3], with a theoretical peak performance of about 413 TFLOP/s, of which about 90% come from the accelerators. Because of the complex architecture of a system that has Intel MIC accelerators and the different ways to use the accelerators, there are alternative parallelization approaches that can be used. One important caveat is that if one is not using the vector instructions on the accelerators they behave like relatively slow regular CPUs. The development of programs that make use of the vector instructions is fostered by the Intel Parallel Studio XE 2016, which provides direct access to the vector instructions via compiler intrinsics. This approach is simpler than the direct coding of assembly instructions inside the C code, which is also feasible.

Application of Mathematics in Technical and Natural Sciences
AIP Conf. Proc. 1773, 070001-1–070001-9; doi: 10.1063/1.4964983
Published by AIP Publishing. 978-0-7354-1431-0/$30.00

One important type of algorithms that are run on HPC systems are Monte Carlo algorithms, which model real-world phenomena or other processes of interest. The quasi-random sequences are sequences that can be used instead of random or pseudo-random numbers in Monte Carlo algorithms, with the aim to achieve faster convergence. Because they are fully or partially deterministic, they provide better theoretical error bounds. In many algorithms they also behave well in practical use, achieving convergence rates that are substantially better than the typical $O\left(N^{-1/2}\right)$ rate of Monte Carlo methods. The star-discrepancy $D_N^\star$ is an established measure of the uniformity of a set of points, which is theoretically related to the integration error via the Koksma-Hlawka inequality (see, *e.g.*,[4]). The low-discrepancy sequences are those that achieve the presumably best rate of convergence of their discrepancy to zero, *i.e.*, $O\left(N^{-s}\log^s N\right)$. Other measures of irregularity are also used, in order to cover other function classes. In practice the observed rates of convergence are better than what one would expect from the theoretical estimates. However, when considering the use of low-discrepancy sequences instead of pseudo-random numbers in a certain Monte Carlo algorithm, one has to take into account some practical drawbacks, [5]. Monte Carlo methods are easy to parallelize and if one uses independent streams of pseudo-random numbers at the different processors, one does not need to complete all the computations in order to obtain an estimate of the result. Thus in an event of processor failure the computations allocated to that processor may be safely discarded. Usually Monte Carlo algorithms provide automatic estimation of the error, which may be used to control the length of the computations. Quasi-Monte Carlo methods do not have such features without some additional work. The technique of "scrambling" can be used in order to add on-the-fly error estimation, since it adds some randomness to the otherwise fully deterministic low-discrepancy sequences and allows for the use of the variation in the results similar to the Monte Carlo case. Various types of scrambling have been investigated, see, *e.g.*, [6], [7]. It is more interesting that sometimes the "scrambled" sequences provide better rates of convergence for certain classes of functions. The modified Halton sequences defined in [8] can also be considered as some kind of "scrambled" Halton sequences. For convenience we provide their definition:

Definition 1 *Let s be the dimension and $p_1, \ldots, p_s$ are distinct prime numbers and let the numbers $k_1, \ldots, k_s$ be admissible for them (see Definition 2.1, [8]). Choose arbitrary integers $b_j^{(1)}, \ldots, b_j^{(s)}$, so that $0 \le b_i \le p_i - 1$, for $j = 0, 1, \ldots$. The ith coordinate of the nth term of the sequence is obtained by expanding n in number system with base p_i:*

$$n = \sum_{j=0}^{\infty} a_j^{(i)} p_i^j$$

and then computing

$$\phi^{(i)}(n) = \sum_{j=0}^{\infty} c_j^{(i)} p_i^{-j-1}$$

where $c_j^{(i)} \equiv b_j^{(i)} + k_i^{j+1} a_j^{(i)} (mod p_i)$, $0 \le c_i \le p_i - 1$.

Note that in order to increase the scrambling effect we had k_i^{j+1} instead of just k_i^j as in the original definition in [8]. The theoretical advantage from the use of modified Halton sequences comes from the estimate (see Theorem 2.3, [8]), which establishes a good constant for the leading term of their discrepancy estimate. By considering the proofs one can observe that the same theoretical estimates are valid also with the above modification. Through the use of random shifting terms $b_1^{(j)}, \ldots, b_s^{(j)}$, one can have independent samplings of the scrambled sequence, thus obtaining automatic error estimates if several streams are used. The kind of computations described in formula 1 are easier to perform if consecutive terms of the sequence are being generated. This means that in practice the blocking approach to parallelization should be favoured instead of the "leap-frogging". The main idea of our algorithm for generation of the Halton sequences is that if the dimension is reasonably high the difference between consecutive terms of the sequence will arise only from the first term in the sum, since the probability of carry in base p_i will be only p_i^{-1}.

Another popular family of sequences are the Sobol' sequences. Different scrambling algorithms have been proposed for them too, see, *e.g.*, [9]. Since our algorithms do not use any special properties of the so-called direction numbers we shall provide a definition that covers a much wider set of sequences and provides for some basic "scrambling." The user of the codes may choose between different tables of direction numbers, for example [10].

Definition 2 *Let s be the dimension and let $A_1, \ldots, A_s$ be infinite binary matrices,*

$$A_i = \left\{a_{jk}^i\right\}_{j=0..\infty}, a_{jk}^i \in \{0, 1\}.$$

Let

$$n = \sum_{j=0}^{\infty} c_j 2^j$$

be the representation of n in binary number system. Choose some random numbers $\beta_i \in [0, 1)$ *and represent them in binary number system:*

$$\beta_i = \sum_{j=0}^{\infty} b_j^{(i)} 2^{-j-1}.$$

Then the binary representation of the ith coordinate of the nth term of the sequence is defined as:

$$x_n^{(i)} = \sum_{j=0}^{\infty} d_j^{(i)} 2^{-j-1},$$

where

$$d_j^{(i)} = \oplus_{j=0}^{\infty} a_j^{(i)} c_j^{(i)} \oplus b_j^{(i)}.$$

It has been shown that the low-discrepancy sequences may be used successfully for problems of high dimensions, for example in financial mathematics where the constructive dimensionality (see [10] for a definition) may be a multiple of 250 (corresponding to the number of trading days in the year).

Our new algorithms for generating the modified Halton sequences and the Sobol sequences on the Xeon Phi coprocessors are described in detail in the next sections. Then we give more information about how they can be used in the various modes of computation for the Xeon Phi coprocessor (native, symmetric, offload). We provide various timing results that demonstrate the efficiency of our approach, compared to standard generation as done on a CPU. We finish with conclusions and directions for future work. Our implementation in computer code is provided under the GNU Public license at [11].

DESCRIPTION OF THE ALGORITHMS

The algorithms are based on the idea to pre-compute certain values that are needed in the computation and to organize the data in a way that data is aligned properly and the most frequent operations can be vectorized. We base our implementation on previous, portable codes. In this way we can compare our hand-tuned vectorized implementation with the results from the automatic vectorisation performed by the compiler as well as the non-vectorized version that can be obtained if this compiler feature is disabled. Since the vector size on Xeon Phi is 512 bits, a vector register can hold up to 16 32-bit integers, 16 single precision floating point numbers or 8 double precision floating point numbers, it is advantageous to process the dimensions 16 at a time. We are only targeting Xeon Phi coprocessors of this generation and the Intel compiler, since the GCC is behind in its support of the vector capabilities.

Algorithm for generation of modified Halton sequences on MIC

The algorithm is based on continuously updating certain quantities like the partial sums and the digits, making use of the fact that most of the time they do not need to be changed or can be changed in parallel, using vector operations. The main data structure used is as follows:

```
struct haltondata {
        fullarraydouble output ;
        fullarraydouble partsums[DIGITSDOUBLE];
        fullarrayint lastdigit;
        fullarrayint blastdigit;
        fullarrayint perturbedlastdigit;
        fullarrayint perturbedblastdigit;
        fullarrayint primes;
        fullarraydouble inveprimes;
        fullarrayint modifiers[DIGITSDOUBLE];
```

```
        fullarrayint digits[REMAININGDIGITSDOUBLE];
        fullarrayint perturbeddigits[DIGITSDOUBLE-2];
        fullarrayint shifters[DIGITSDOUBLE];
}  __attribute__ (( aligned(64)));
```

where

```
typedef double fullarraydouble [FULLDSIZE]
__attribute__ ((aligned(64)));
typedef float  fullarrayfloat [FULLDSIZE]
__attribute__ ((aligned(64)));
typedef int    fullarrayint [FULLDSIZE]
__attribute__ ((aligned(64)));
```

with In order to ensure correctness, it is necessary to enforce the alignment on a 512-bit boundary. Here it is a achieved with the gnu extension __attribute__((aligned (64))). The constant DIGITSDOUBLE is equal to the number of binary digits in the mantissa of a double precision number, *i.e.*, 52. The constant FULLDSIZE is equal to the dimension, rounded up to a multiple of the number of integers or floats that can be held in a vector register, *i.e.*, a multiple of 16.

The algorithms consists in updating the partial sums

$$\phi_m^{(i)}(n) = \sum_{j=m}^{\infty} c_j^{(i)} p_i^{-j-1}$$

where the partial sums with higher index m would change far less frequently than those with lower index. Obviously the final result is held in the partial sums with index 0.

If the least significant digit of n in base p_i is increased by one, most of the time it will not reach p_i and only the last partial sum will have to be updated, using the partial sum with index one. In such case only $c_0^{(i)}$ will change. In our implementation these digits are stored in the array perturbedlastdigit. To update $c_0^{(i)}$ we have to add k_i and if the result is $\geq p_i$ subtract p_i. Once the perturbed last digit is obtained, it has to be multiplied by p_i^{-1} and added to the partial sum with index 1 in order to obtain the partial sum with index 0. This part of the algorithm is the most time consuming and obvious candidate for vectorization. If the least significant digit of n in base p_i is $p_i - 1$, after advancing it we obtain p_i and we have to set it back to 0 and use the carry to update the other digits. We shall advance the next digit, update the partial sum with index 1 and deal with potential changes in the other digits. The operations on the digits before the last ones can also be vectorized, by using the masking technique. If we obtain a mask that specifies which elements of a vector have to be updated and which vectors are to be kept in place, we can issue a vector instruction to do that in an efficient way.

The vector instructions can be accessed with *asm* statements, but it is cleaner to use the intrinsics provided by the Intel compiler. The instructions that we used to work with the last digit were

```
_mm512_add_epi32, _mm512_set1_epi32, _mm512_cmpge_epi32_mask,
_mm512_mask_sub_epi32,
_mm512_cvtepi32lo_pd, _mm512_fmadd_pd,
_mm512_extload_epi64
```

where the instructions from the third row deal with floating point numbers and the last one was used because the number of ints in a register is more than the number of doubles, so we need more instructions to deal with the double precision quantities than for the integers. We also make use of the bit masks that can modify which of the results of a vector operation are actually copied to memory or where carry has occured. We point out that the generation of the original Halton sequences can be achieved by replacing the admissible numbers with ones. However, we consider that for the high dimensions that we target using the original Halton sequence is not advised, because of the correlations between consecutive dimensions (see, *e.g.*, [12]). Our observation is that because of the scrambling effect of both the admissible numbers and the numbers $b_j^{(i)}$, the modified Halton sequence is competitive with the other families of low-discrepancy sequences in high dimensional problems.

Algorithm for generation of Sobol' sequences on MIC

From the very definition of the Sobol' sequences it is obvious that they are simpler to implement on binary computers. Our generation algorithm, based on hand-tuned vectorisation, provides for efficient generation of a wide set of

sequences that includes the Sobol' sequences and provides for some scrambling so that aposteriory error estimation can be performed. The data structure that is necessary for the consecutive generation of the terms of the Sobol' sequence is simpler than that for the Halton sequences, because the main operation to be performed is the *xor* operation. Essentially we store only the direction numbers in an equivalent form and the current term of the sequence again in an equivalent form. The generation of the next term of the sequence consists of performing one vector *xor* operation and then one vector operation in double precision. The trick that we employ has been described before [13]. It is based on the fact that the representation of $(1 + x)$ in double precision according to the IEEE 754 standard will have the bits related to the exponent fixed, because it is always between 1 and 2. In the definition of the Sobol' sequence above the matrices A_i represent the direction numbers. For our vector algorithm we need to store the "twisted" direction numbers defined as cumulative exclusive or of the usual direction numbers:

$$t_i^{(j)} = \oplus d_i^{(j)},\ j = 1, \ldots, s$$

where the $\oplus$ operation is performed bitwise. They are stored in memory that is aligned to 512-bit boundary in order to speed-up the subsequent vector *xor* operations. The main loop of the generation looks like this:

```
for (i = 0; i < ((soboldim & ~1) )*2; i += 16) {
                Is32vec16 *r2 =(Is32vec16 *) & yptr [i];
                Is32vec16 *r3 =(Is32vec16 *) & twistptr[i];
                *r2=Is32vec16(_mm512_xor_epi32(*r2,*r3));
        }
    .
F64vec8 r2 = *(F64vec8*)&minusone[0];
for (i = 0; i < (soboldim & ~1); i += 8) {
        F64vec8 iparts = *(F64vec8 *)(& yrealptr[i]);
        F64vec8 res= _mm512_add_pd( iparts , r2);
           *(F64vec8*)(&result[i]) = res;
}
```

which generates 8 coordinates in double precision in each iteration. The remaining up to 8 coordinates are generated sequentially. The results of the *xor* operation are stored in our memory for later use. The results from subtracting 1. in double precision from all of them are provided to the user, assuming he or she has provided us with aligned memory space for the results. It is obvious that the generation of the Sobol sequence in this way is highly efficient, although in the conclusions we shall mention some possibilities for further improvement. Even though it is relatively straightforward, we will see later that the results from the compiler's auto-vectorization are not as good as the hand-tuned code. Our generation routines use the direction numbers from [10], but the user can easily replace them if necessary. Note that we do not make use of the property that $a_{jk}^{(i)} = 0$ when $k > j$, which gives some additional freedom. However, it is possible to increase the generation speed slightly, if appropriate packing strategy for storing the direction numbers is employed if this property is assumed.

PARALLEL COMPUTATIONS USING LOW-DISCREPANCY SEQUENCES

There are two main approaches for parallelization of quasi-Monte Carlo computations – blocking and leap-frogging. Our codes currently support only generation of the next term of the sequence using information produced during pre-processing or during the generation of the previous term of the sequence. This means that the blocking approach to parallelisation should be used, rather than the leap-frogging. If a process or thread has to deal with more than one block, one should be aware that the overhead of the preprocessing needed for initialization of the data is high and thus the block size should be quite large in order to achieve good efficiency. Therefore it is preferable if only one block per OMP thread or MPI process is used. Considering the three main ways of using the Xeon Phi accelerators – symmetric, native, offload, we can see that from point of view of parallel computations, there are two main approaches to make use of our codes – either to perform all the computations on the accelerators, or to offload the generation of the low-discrepancy sequences only and to perform the main computations on the CPU. Since the accelerators are generally more powerful than the host processors, the latter approach is interesting only if there are reasons why the computations to be done on the CPU should not be done on the accelerators. Such reasons may be, *e.g.*, the need to use a lot of RAM, the need to perform mainly integer operations, or the need to use some libraries that have not been

ported to Xeon Phi. This approach can also be useful if the number of coordinates of the sequence that are actually required for the computations vary. When using accelerators for the generation, one may afford to generate much more coordinates than are required on the average.

Our generation codes may be used equally well in all types of parallel computations. For the offloading case, if only the generation of the sequence is to be done on the Xeon Phi, it would be advisable to add a buffer so that the generation is performed independent of the host computations and terms of the sequence are picked up from that buffer as the computation goes along.

TIMING AND NUMERICAL RESULTS

We compared our vectorized implementation with the simpler one that is also usable on regular CPUs and we concluded that they are functionally the same, within the expected round-off errors. That is why the numerical results obtained with the vectorized version are similar to the results of the regular version. All our results were achieved on our servers SL250s, equipped with two Intel Xeon Phi 7120P coprocessor cards.

As a demonstration that the modified Halton sequences achieve good accuracy even at high dimensions, where the vectorisation approach provides substantial speed benefit, we present the integration errors when computing the integrals of the functions

$$f_1(x) = \prod_{i=1}^{s} \left(x_i^3 + \frac{3}{4} \right)$$

(Figure 1) and

$$f_1(x) = \prod_{i=1}^{s} |4x_i - 2|,$$

(Figure 2) over the s-dimensional unite cube $I^s = [0,1]^s$, where $s = 32$, with various numbers of terms of the modified Halton and Sobol' sequences. The approximate errors are obtained by varying the scrambling terms. For the usual Monte Carlo method we estimate the RMS error. It is drawn as a solid line (because of using log-scale).

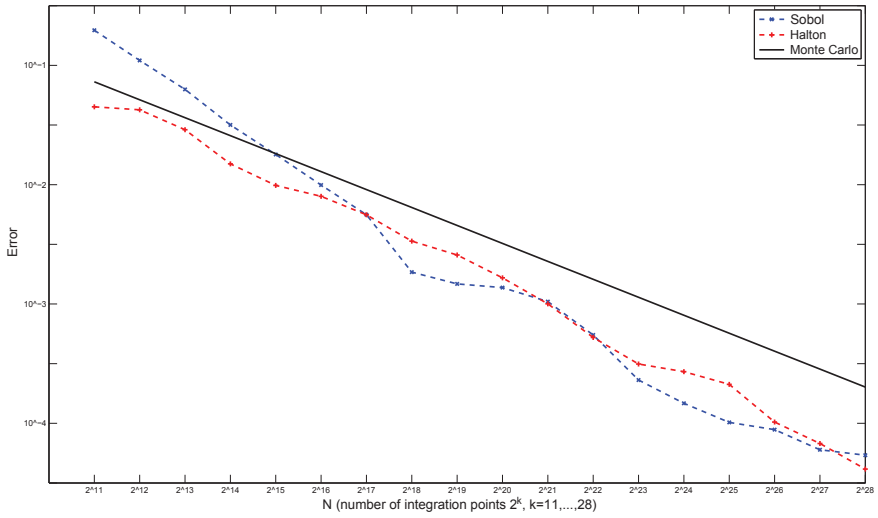

FIGURE 1. Integration errors for the test function f_1

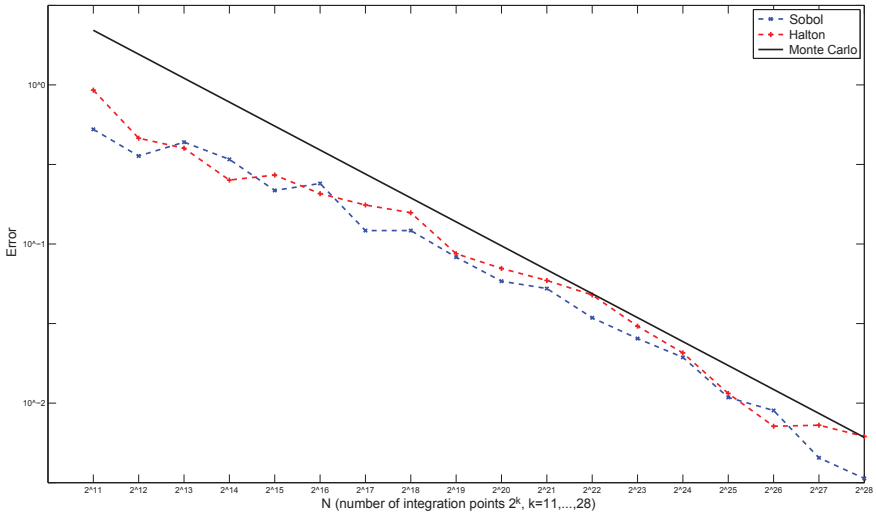

FIGURE 2. Integration errors for the test function f_2

We can see how the low-discrepancy sequences of Halton and Sobol' sequences behave in a similar way and most of the time achieve smaller errors than the Monte Carlo method, even though the dimension is so high that the theoretical estimates do not guarantee such results.

We are more interested in comparing speeds of sequential and vectorized version. We point out that the "sequential version" is still subject to vectorization by the compiler. Our tests when the automatic vectorization is disabled (option '-no-vec') showed significant difference, especially for the Sobol sequences. In the next table we present timing results of both versions, running on single core on the Xeon Phi. A trivial computation is performed with the generated coordinates, in order to prevent the compiler from optimising-out the whole generation code.

TABLE 1. Comparison of speeds of generation in millions of coordinates per seconds, 10 000 000 points in 256 dimensions

Sequence	Time (s)	Speed	Speedup
Sobol Auto	26.9	95.3	
Halton Vectorized	5.6	455.7	4.8
Halton Auto	315.5	8.1	
Halton Vectorized	22	116.4	14.3

From this table we conclude that the vectorized codes significantly outperform in terms of speed. That is why when we move to test the parallel performance we only consider the vectorized codes. When the user is using accelerators for computation, it is natural to assume that he or she has full control of the server and thus the most important case is when all the physical cores of the Intel Xeon Phi card are used. In Table 2 we show the parallel perforance when using number of MPI processes up to the number of physical cores in the system (61).

Although the parallel efficiency drops when more than 16 processes are used, it is partly because the overall times become too small and startup costs become more important. In practice we would expect that the user shall make use of all the available cores with a parallel application. In the case of Intel MIC architecture there is one importance resource – the hyperthreading allows for up to 4 independent threads of execution per hardware core. The advantage of hyperthreading consists in the possibility of using optimally the available internal units in the CPU, since different threads may be using different units at a given moment in time. We point out that to make maximum use of the vector processing units it is actually necessary to use hyperthreading. That is why we tested our generation codes with

TABLE 2. Comparison of speeds of generation when using MPI, 100 000 000 points of the Sobol sequence and Halton sequence in 256 dimensions

Sequence	**Cores**	**1**	**16**	**32**	**61**
Sobol	Time (s)	54.5	3.4	1.9	1.1
	Speedup		16.0	28.2	51.8
Halton	Time (s)	219.1	14.3	7.2	4.3
	Speedup		15.3	30.3	51.1

different levels of hyperthreading, by employing multiples of up to 4 the number of physical cores. In Table 3 we show timing results achieved on one card. In general it is not guaranteed that any speedup from hyperthreading will be obtained. In our tests we see that when one card was used hyperthreading always improved the results and that the maximum improvement for the Sobol' sequence was obtained with 122 threads, while for the Halton sequence 244 times were marginally better. Since the generation of the low-discrepancy sequence form only one part of the quasi-Monte Carlo application we can not give a general advice except that it seems that this part gains from using hyperthreading and this possibility is worth exploring in each particular case.

TABLE 3. Effects of hyperthreading when using 61, 122, 183 or 244 MPI processes go generate 10^9 points in 256 dimensions

Cards/Hyperthreading		No	2×	3×	4×
Sequence	**Processors**	**61**	**122**	**183**	**244**
Sobol	Time (s)	9.81	7.12	8.46	9.40
	Speedup		**1.38**	1.16	1.04
Halton	Time(s)	38.95	33.42	31.86	31.67
	Speedup		1.17	1.22	**1.23**

In a more general setting, when many Intel Xeon Phi accelerators are used, the situation becomes more complicated. In Table 4 we show timing results when multiple cards are used and varying the amount of hyperthreading. In bold-face we show the best results for any fixed number of accelerator cards. Since we use MPI, the total number of processes becomes rather high and this may be the reason why the advantage of hyperthreading decreases. However, since the other parts of a quasi-Monte Carlo application may still benefit from hyperthreading, it is important to observe how much of a disadvantage may be presented by the generation part of the algorithm. Most probably this difference between the single and multiple card cases stems from some internals of the MPI implementation, which could be overcome if OpenMP or other threading model is used to decrease the total number of MPI processes and the startup costs. Overall we should have in mind that the ability to generate billions of points in 256 dimensions in less than one second for the Sobol' sequences and in about 3 seconds for the Halton sequence opens up possibilities to use quasi-Monte Carlo algorithms that use-up high numbers of coordinates, for example by applying acceptance-rejection methods.

TABLE 4. Parallel efficiency using multiple MIC Cards, Times for generating 10^9 points in 256 dimensions, using different number of Xeon Phi cards, varying the usage of hyperthreading

Sequence	**Cards/Hyperthreading**	**1**	**2**	**4**	**8**	**16**
Sobol	No	9.81	5.49	2.62	1.43	**0.90**
	2×	**7.12**	**4.06**	**2.20**	**1.38**	1.18
	3×	8.46	4.11	2.44	1.69	1.63
	4×	9.4	5.66	3.64	3.50	4.97
Halton	No	38.95	20.16	11.76	5.87	**3.06**
	2×	33.42	**16.40**	**8.98**	**5.13**	3.44
	3×	31.86	17.18	9.36	5.39	3.78
	4×	**31.67**	19.35	12.31	9.28	7.64

CONCLUSIONS AND DIRECTIONS FOR FUTURE WORK

We have presented vectorized versions of generation routines for the modified Halton sequences and the Sobol' sequences, which achieve excellent efficiency using single core or multiple cores of the Xeon Phi coprocessors. The code is available under the GNU Public License. There are some paths for further improvement, for example it should be possible to use 16-bit short integers for the digits and prime numbers and benefit from the smaller data sizes for the Halton sequences. For the Sobol' sequences it would be possible to use compressed direction numbers for the least significant digits and unpack them on-the-fly. The generation codes for the Sobol' sequences allow both the blocking and leap-frogging approach for parallelisation, provided leap-frogging is done by power-of-two steps. Only the blocking approach is supported for the Halton sequence currently. In both cases it would be interesting to implement leap-frogging by any step, although the codes will become even more difficult to read. In the future we plan to use the codes in practical problems and evaluate these possibilities in order to implement those of them that could provide substantial benefit. Our investigation of the parallel performance of the generation routines revealed that there is substantial benefit to be obtained by using hyperthreading, most probably because in both cases we interleave floating point and integer operations. It seems that using at least 122 threads is necessary for optimum performance.

ACKNOWLEDGEMENTS

This work was supported by the National Science Fund of Bulgaria under Grant #DFNI-I02/8 and by the European Commission under H2020 project VI-SEEM (Contract Number 675121).

REFERENCES

[1] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High-Performance Programming* (Morgan Kaufmann, Boston, 2013).

[2] Intel xeon phi coprocessor instruction set architecture reference manual, `https://software.intel.com/sites/default/files/forum/278102/327364001en.pdf`.

[3] Supercomputer Sites, `http://top500.org/`.

[4] R. E. Caflisch (1998) *Acta Numerica* **7**, 1–49.

[5] H. Niederreiter, *Random Number Generation and Quasi-Monte Carlo Methods* (Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992).

[6] A. B. Owen, "Randomly permuted (t,m,s)-nets and (t, s)-sequences," in *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, Proceedings of a Conference at the University of Nevada, Las Vegas, Nevada, USA, June 23–25, 1994, edited by H. Niederreiter and P. J.-S. Shiue (Springer, New York, 1995), pp. 299–317.

[7] E. Atanassov, A. Karaivanova, and S. Ivanovska, "Tuning the generation of Sobol sequence with Owen scrambling," in *Proceedings of the 7th International Conference on Large-Scale Scientific Computing, LSSC'09*, (Springer-Verlag, 2010), pp. 459–466.

[8] E. I. Atanassov, "On the discrepancy of the Halton sequences," in *Math. Balkanica*, New Series, Vol. 18, Fasc. 1–2 (2004), pp. 15–32.

[9] A. B. Owen (1998) *J. Complex.* **14**, 466–489.

[10] S. Joe and F. Y. Kuo (2008) *SIAM J. Scientific Computing* **30**, 2635–2654.

[11] E. Atanassov, *Generation Codes for Sobol and Halton Sequences on Intel MIC:* `http://parallel.bas.bg/\%7Eemanouil/sobolhaltonmic/sobolhaltonmic.tar.gz`, (2016).

[12] H. Chi, "Scrambled quasirandom sequences and their applications," Ph.D. thesis, Florida State University, Tallahassee, FL, USA, 2004.

[13] E. I. Atanassov, "A new efficient algorithm for generating the scrambled Sobol' sequence," in *Numerical Methods and Applications: 5th International Conference, NMA 2002 Borovets, Bulgaria, August 20–24, 2002*, Revised Papers, (Springer, Berlin Heidelberg, 2003), pp. 83–90.