On the Parallelization Approaches for Intel MIC Architecture

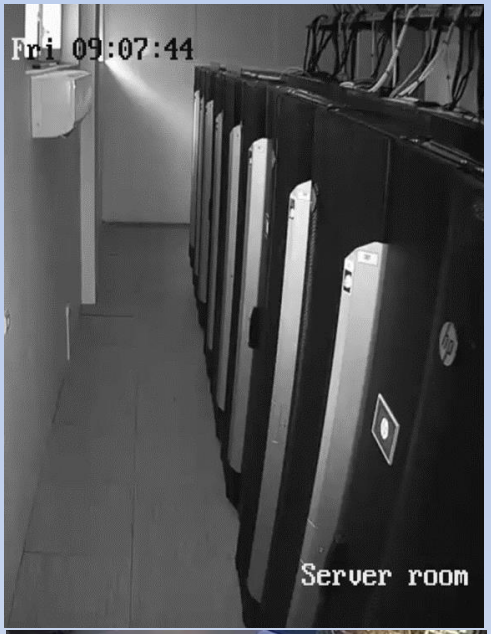
Aneta Karaivanova

IICT-BAS

(Joint work with E. Atanassov, T. Gurov, and S. Ivanovska)

The supercomputer AVITOHOL(332 in Top500 – 2015)

- The system consists of 8 racks of type HP MCS 200, paired in couples.
- Each pair provides power and cooling for up to 50 kW of equipment, cooled by water cooling.
- 150 servers, each with 2 8-core Xeon CPUs and 2 Xeon Phi 7120P
- Red Hat Enterprise Linux, Intel Cluster Studio
- **About 90% of the computational power comes from the accelerators** – one 7120P coprocessor achieves 1.25 TFlop/s in double precision.



Content

- Introduction
- Target architecture
- Parallelization strategies and case studies:
  - Generation of Halton and Sobol sequences
  - Monte Carlo for multidimensional integrals
  - Monte Carlo for matrix computations
- Conclusion

Introduction

- Intel **M**any **I**ntegrated **C**ore (**MIC**) architecture was used on coprocessor devices in the first generation of the Intel Xeon Phi line of processors.

- Main characteristics of the coprocessors:
  1. Equipped with vector units that allow processing of several integers or floats at once
  2. Running many cores at low frequencies
  3. Availability of hyperthreading

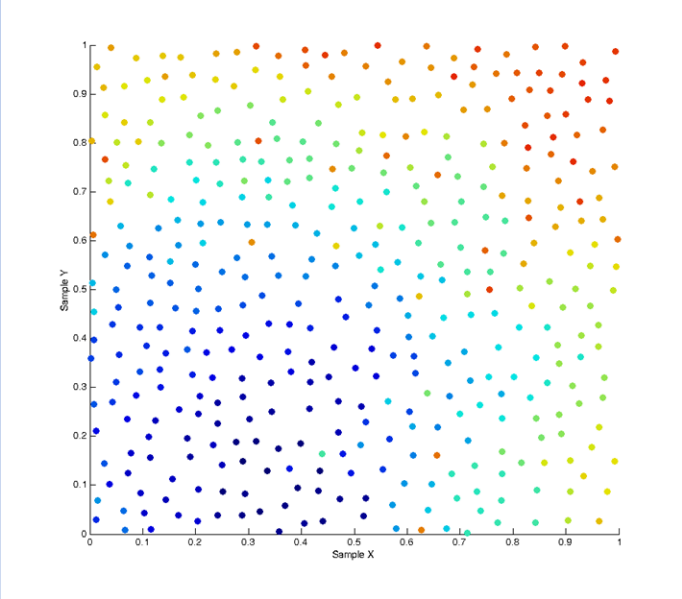
If vector instructions are used on such accelerators their full capabilities for high-performance computing are utilized.

- The Intel suite of compilers and other development tools provide:
  1. Direct access to the vector instructions, facilitating their use by the program developers
  2. Direct coding of assembly instructions inside C codes

Different ways of using Intel MIC accelerators lead to multiple effective parallelization approaches.

Pseudorandom numbers and quasi-random numbers

- Typically Monte Carlo algorithms use pseudo-random number generators
- Quasi-random numbers are specially designed sequences with additional uniformity properties
- Quasi-Monte Carlo algorithms may achieve faster convergence when using low-discrepancy sequences



General advantages and disadvantages of pseudorandom and quasi-random numbers

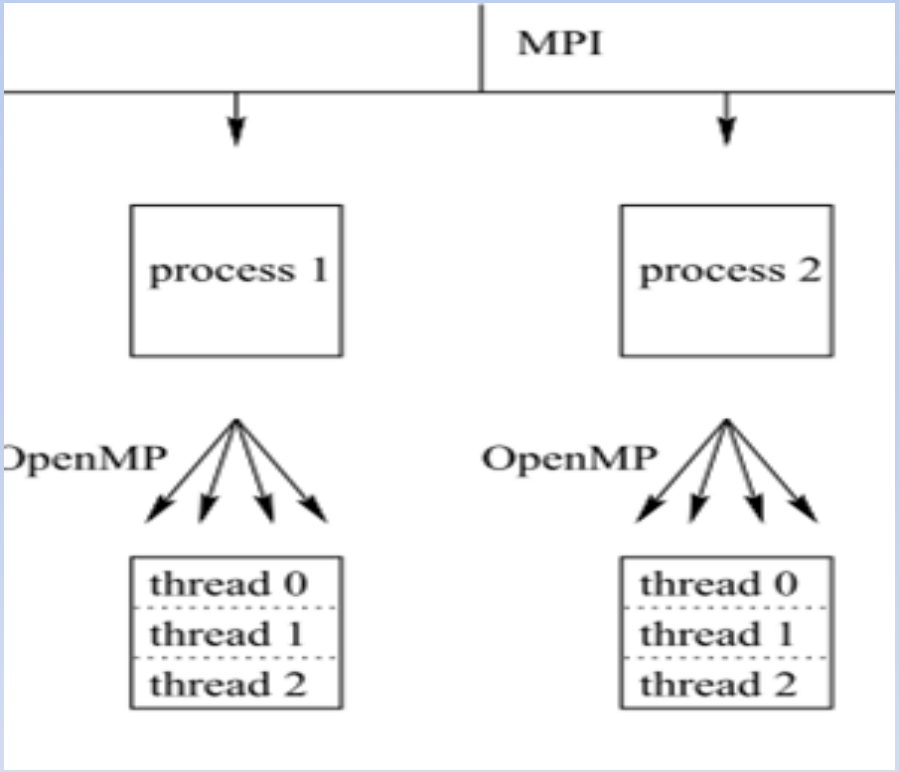
- Pseudorandom number generators are extensively studied and many competing constructions are used
- Many constructions of low-discrepancy sequences are defined, some of them suitable only for certain classes of functions
- In practice better generators are more complex and thus slower
- Many of the constructions of low-discrepancy sequences are complicated and difficult to implement
- To be used in parallel computing, the pseudo-random number generators should have additional properties

What is Intel MIC architecture

- MIC stands for Multiple Integrated Cores
- Intel designed processors that can contain 60 or more cores, with instruction set close to x86 with its 64-bit extensions, but with added vector instructions that lead to increased floating point performance
- Initially they could be used only as accelerator cards, but new generations (KNL) are stand-alone.
- Familiar instruction set and development tools, visible as a CPU on a Linux machine.
- High energy efficiency, compact, high number of independent threads of execution (4x hyper-threading).

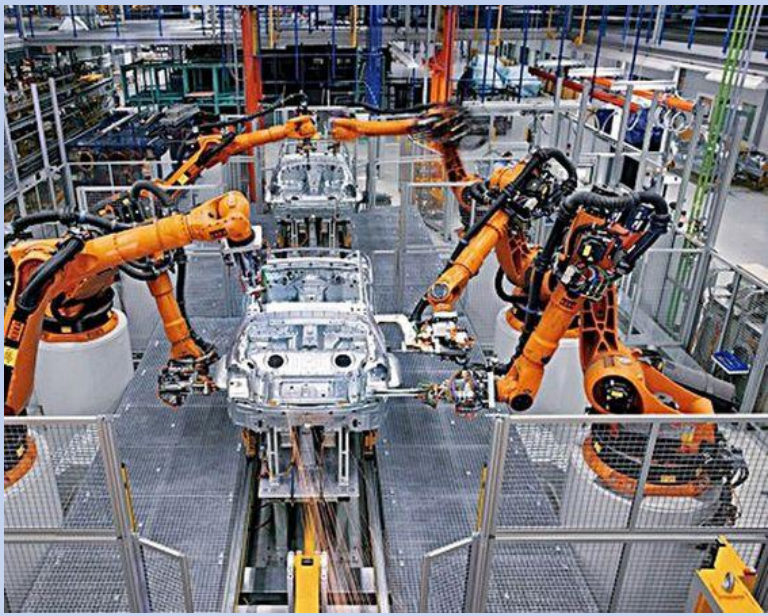
Models for parallel computing on MIC

- A single card can be used for parallel computing via OpenMP, MPI or OpenMP+MPI.
- The offload mode combines host CPU and MIC, but is not future proof. Most computationally intensive functions can be offloaded.
- Multiple cards can be used with MPI only or hybrid OpenMP+MPI.



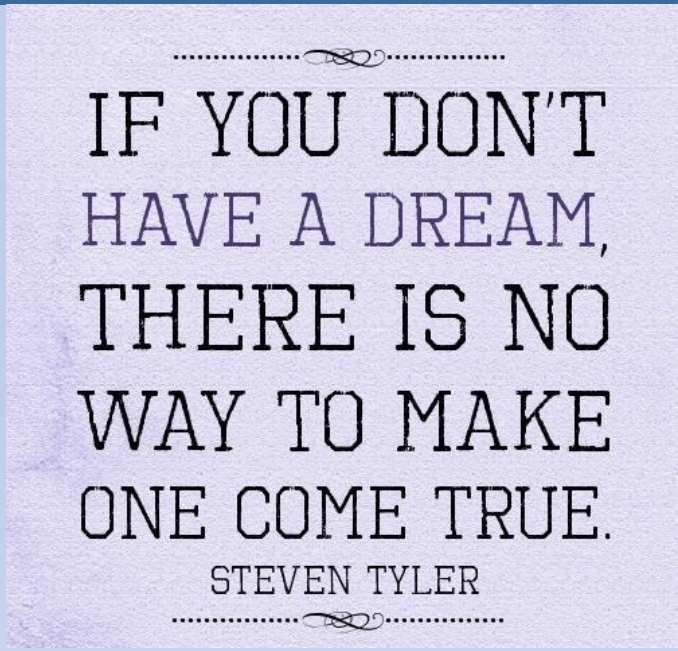
The importance of vector processing on MIC

- **Without using vector processing instructions, Intel MIC-based cards are slower than the Xeon CPUs**.
- To make full use of vector instructions, one must use more threads than the number of physical cores on the MIC
- A vector instruction can deal with 8 doubles or 16 floats at a time.
- Vector instructions can deal with integers too
- GNU compiler still can not vectorize code for MIC
- Intel compilers should be able to vectorize automatically.
- This is just a dream!



More dream scenarios

- Take parallel MC code
- Replace the pseudorandom number generator with some QRN sequences
- Enjoy increased accuracy
- Implement the definition of some low-discrepancy sequence
- Test on CPU
- Recompile
- Enjoy the benefits of automatic vectorization



Motivation for developing for Intel MIIC

AVITOHOL at IICT-BAS
150x HP ProLiant SL250s Gen8 each with
2x Intel Xeon E5-2650 v2 (8C/16T),
64 GB DDR3-1866 RAM and
2x Intel Xeopn Phi 7120P
6x HP ProLiant DL380p Gen8 nodes with
2x Intel Xeon E5-2650v2 (8C/16T),
64 GB DDR3-1866 RAM
Infiniband 56 Gb/s FDR
Storage system with 96 TB

AVITOHOL

Total Performance:
RPeak:412.3 TFlop/s
RMax: 264.2 TFlop/s
Top 500 position: 389

HPCG cluster at IICT-BAS
36 blades BL 280c(2x Intel X5560(4C/8T); 24GB DDR3);
8 management nodes HP DL 380 G6(2x Intel X5560(4C/8T); 32GB DDR3);
2 HP ProLiant SL390s G7(2x Intel E5649(6C/12T);96GB DDR3)
8x nVidia TESLA M2090 per server;
2 HP SL270s Gen8 (2x Intel Xeon E5-2650 v2(8C/16T); 128GB DDR3)
Total number of Xeon Phi 5110P coprocessors: 9
Total 132TBs of system storage

TOTAL PERFORMANCE:
RPEAK:22.94 TFlop/s

NCSA IBM Blue Gene/P
8192 PowerPC 450 processors
4TBs of system memory
12TBs of system stotage
IBM proprietary interconnect with
2.5 μs latency and 10GBps bandwidth

TOTAL PERFORMANCE:
RPEAK:27.85 TFlop/s
RMAX:23.45 TFlop/s

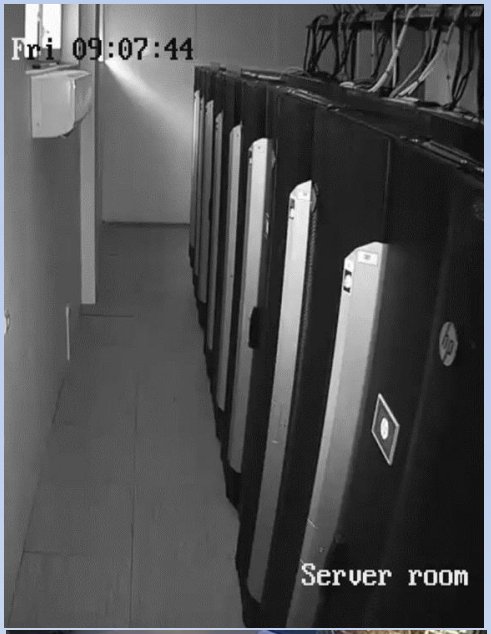
PHYSON at Sofia University
53 Intel Xeon x86_64 processors
524Gibs of system memory
6.5TBs of system storage
2x nVidia Tesla M2090 graphics processors

TOTAL PERFORMANCE:
RPEAK:3.57 TFlop/s
RMAX: 3.22 TFlop/s

MADARA at IIOCCP-BAS
54 Primergy RX200 S5 servers with
2 Intel Xeon E5520(4C/8T) each
and a total of 800GB DDR3 1066MHz
20Gb/s DDR Infiniband
108TB System Storage by Fujitsu FibreCat SX100

The supercomputer AVITOHOL

- The system consists of 8 racks of type HP MCS 200, paired in couples.
- Each pair provides power and cooling for up to 50 kW of equipment, cooled by water cooling.
- 150 servers, each with 2 8-core Xeon CPUs and 2 Xeon Phi 7120P
- Red Hat Enterprise Linux, Intel Cluster Studio
- **About 90% of the computational power comes from the accelerators** – one 7120P coprocessor achieves 1.25 TFlop/s in double precision.



Parallel pseudo-random number generators

- A sequential pseudo-random number generator can be used in parallel if one of the “blocking” or “leap-frogging” approaches can be implemented.
- Care should be taken to ensure lack or correlation between different streams
- Certain pseudo-random number generators can not be used with too many parallel processors.
- Standard packages – SPRNG, Intel MKL
- A very popular generator is the Mersenne Twister (large period).
- Generation is suggested to be done in buffers instead of point by point.

Parallel QRN generation

- Pre-processing part, with possibility to adding some randomness at initialization time.
- Memory necessary to keep state and facilitate move to next point
- Should be able to generate from any starting point
- Ideally should be able to skip (too complicated in most cases).
- Blocking and leap-frogging approaches to parallelization of QRNs are the most used.

Generating the Sobol and Halton sequences on MIC

- Using experience from previous architecture-specific codes.
- Some of them were quite involved and did use vector capabilities of CPUs
- Start from non-vector code and hope the compiler will figure out how to vectorize.
- Determine which parts are most critical and amenable to vectorization.
- Observation – when high number of threads are used, the memory that keeps state becomes a problem.
- Conclusion – use OpenMP+MPI instead of pure MPI.

Generating the Sobol sequence on MIC with OpenMP+MPI

- Split the state in two parts – constant and mutable. The constant part can be shared between threads.
- Make use of the fact that lower triangular matrices are used for direction numbers, thus less memory is needed
- The internal loop goes like this:

```
reg1.integer=*(__m512i  *) & yptr [2*i];
reg2.integer=*(__m512i  *) & twistptr[2*i];
reg1.integer=_mm512_xor_epi64(reg2.integer,reg1.integer);
*(__m512i*)& yptr [2*i]=reg1.integer;
*(__m512d*)(&result[i]) = _mm512_add_pd( reg1.floating, constantminusone);
```

Generating the Halton sequence on MIC

- Halton sequence is much more complicated.
- Based on number systems with distinct prime bases.
- It is rarely used without scrambling (which makes it even more complicated).
- Because of the many logical conditions – not easy to vectorize
- Lots of integer arithmetic operations.
- Significantly slower than Sobol.

Timing results

- Speed benefit is observed best at high dimensionalities (multiples by 16 or at least 8).
- Sobol is significantly faster than Halton
- Consider 320 dimensions, 10 000 000 numbers
- Intel Xeon Phi 7120P accelerator
- Sobol:
  - Scalar code, auto vectorization – 67s
  - Scalar code, no auto vectorization – 123s
  - Vectorized code by hand: **13** seconds
- Halton:
  - Scalar code, auto vectorization - 780 seconds
  - Scalar code, no auto vectorization – 787 seconds
  - Vectorized code by hand: **57** seconds
- Pseudorandom (Mersenne twister from MKL): **19.75**

Test integrals

- The functions to be integrated are defined as follows:

$$F_1 = \prod_{i=1}^{d} \left(x_i^3 + \frac{3}{4} \right)$$

$$F_2 = \prod_{i=1}^{d} |4x_i - 2|$$

$$F_3 = \prod_{i=1}^{d} \frac{\pi}{2} \sin \pi x_i,$$

$$F_4 = (1 + {}^1/_d)^d \prod_{i=1}^{d} (x_i)^{{}^1/_d}$$

over the d– dimensional unite cube $I^d = [0, 1]^d$, where d = 32, 64.

Results using multiple threads and cards

Effects of hyperthreading for Halton sequence, Xeon Phi cards

Cards	1		2	
Threads|Dimension	32	64	32	64
61	45.82	62.82	25.89	33.83
122	33.78	40.22	16.32	20.85
183	28.46	37.99	14.87	20.01
244	27.91	36.15	14.49	19.24

Scalability results comparison

Table: Scalability results using 61 cores, within different number of Xeon Phi cards

Generator	dimension	Cards	1	2	4	8	16	32
Sobol	d=32	Time (s)	5.60	2.90	1.55	0.83	0.52	0.39
		Speedup		1.9	3.6	6.7	10.7	14.3
	d=64	Time (s)	7.25	3.71	1.93	1.07	0.62	0.44
		Speedup		1.9	3.7	6.8	11.6	16.4
Halton	d=32	Time (s)	15.3	7.62	3.97	2.12	1.24	0.73
		Speedup		2.0	3.8	7.2	12.3	20.9
	d=64	Time (s)	22.25	10.93	5.72	2.96	1.61	1.02
		Speedup		20.9	3.9	7.5	13.8	21.8
MT2203	d=32	Time (s)	26.99	13.64	7.16	3.57	1.97	1.16
		Speedup		1.9	3.8	7.6	13.7	23.2
	d=64	Time (s)	26.94	14.03	7.23	3.71	2.00	1.14
		Speedup		1.9	3.7	7.2	13.4	23.5

Numerical results

Table: 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

Numerical results

Table: Comparison of speeds of generation when using MPI and OpenMP, 100 000 000 points of the Sobol sequence and Halton sequence in 256 dimensions

Sequence MPI	Cores	1	16	32	64
Sobol	Time (s)	54.5	3.5	1.9	1.1
	Speedup		15.5	28.6	51.8
Halton	Time (s)	219.1	14.3	7.2	4.3
	Speedup		15.3	30.3	51.1
Sequence OpenMP	**Cores**	**1**	**16**	**32**	**64**
Sobol	Time (s)	54.5	3.4	2.1	1.0
	Speedup		16.0	25.9	54.5
Pseudorandom	Time (s)	146.3	9.2	5.2	3.3
	Speedup		15.9	28.1	44.3

Numerical results (D=64)

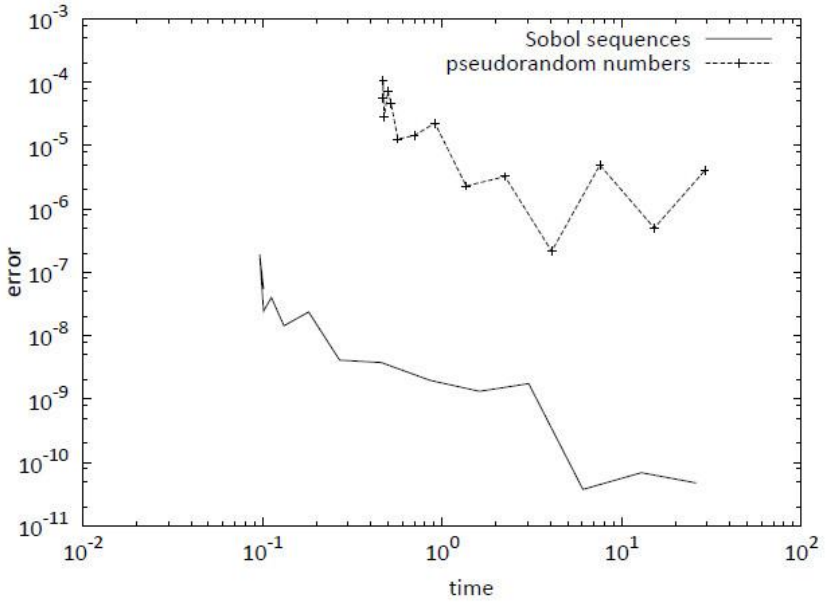


Figure 1: Integration errors for F_2.

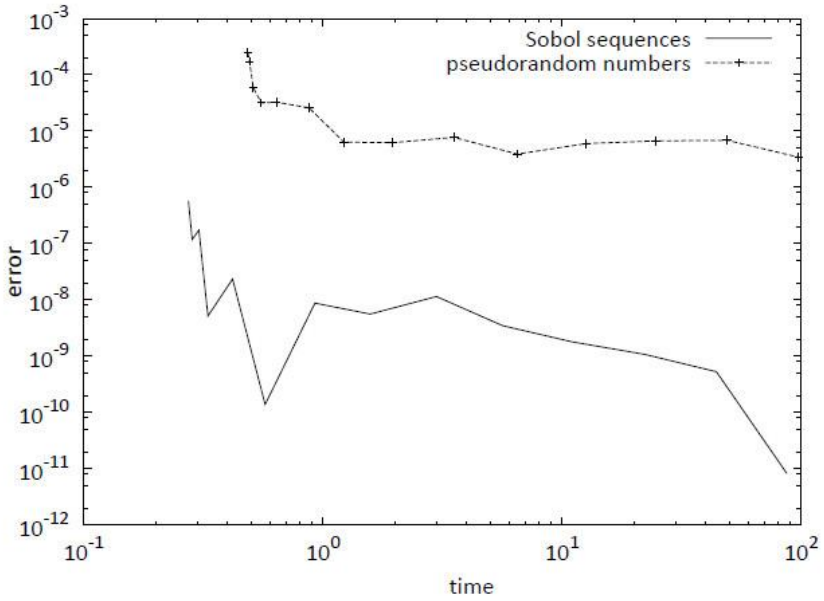
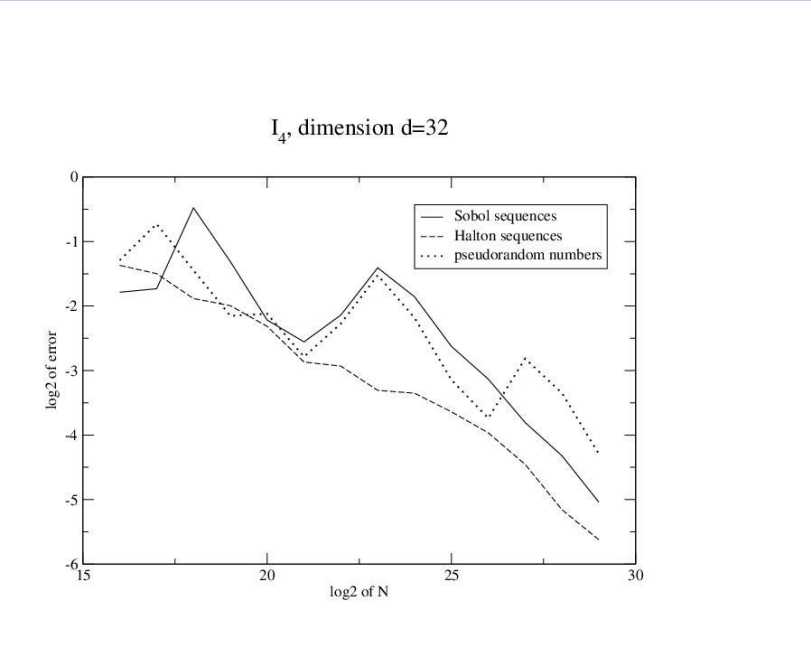


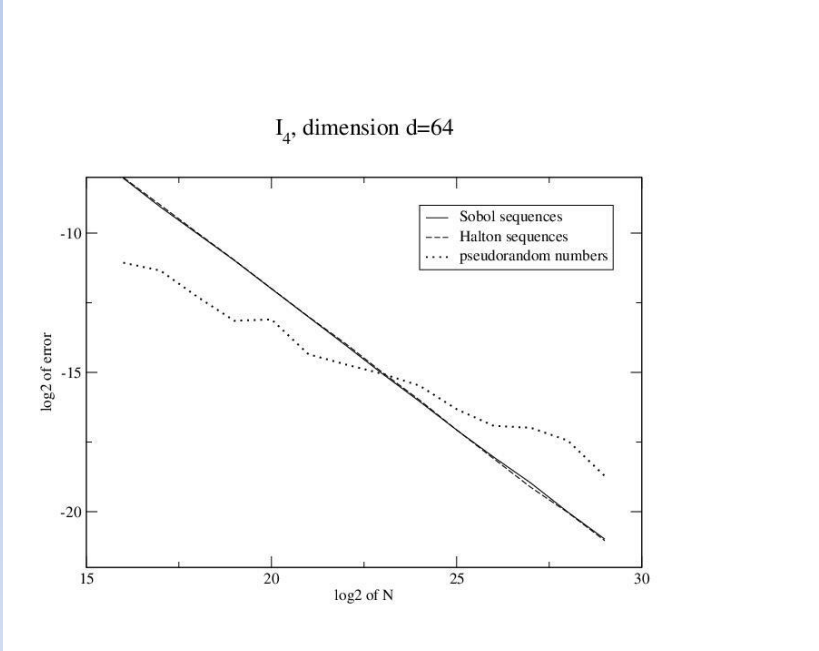
Figure 2: Integration errors for F_3.

Numerical results (F4)

- Figure 3: Integration errors for F4.



- Figure 4: Integration errors for F4.



Numerical results

Figure 1: Integration errors for F4.

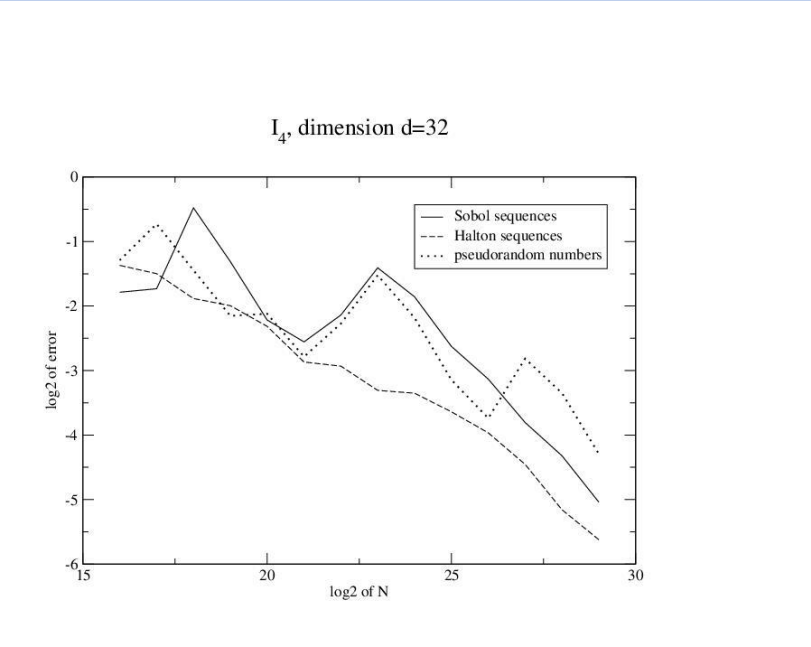
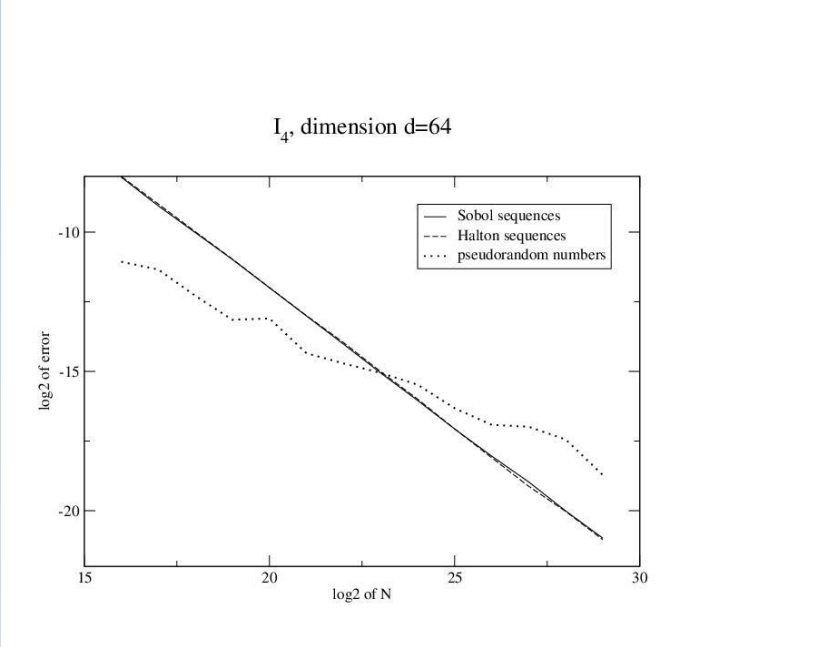


Figure 2: Integration errors for F4.



MCM for matrix computations

Monte Carlo algorithms for computation of approximation of inverse matrix based on sampling of Markov chains:

- Monte Carlo algorithm - Markov chain is based on jumping along the non-zero elements either along rows or columns (depending on the problem)
- Quasi-Monte Carlo algorithm - each Markov chain corresponds to a different term of the low-discrepancy sequences and then use different coordinates of this term to move along the non-zero elements

Quasi-MCM for matrix computations

- **Problem :** when sampling from a discrete distribution speed advantage of the Monte Carlo methods is lost
- **Solution** for Monte Carlo algorithm:
  - Using faster methods that use O(1) number of operations for the sampling

- **Problem** for Quasi-Monte Carlo algorithm:
  - The implementation of the method depends of the smoothness of the underlining function which in case of matrices is a multidimensional step function
  - Any kind of permutation of indices may decrease the smoothness
- **Solution:**
  - Using *tabular method* for sampling discrete distribution;
  - The pre-computing of this table does not increase the total order of the number of operations

Tabular method

- When we have a pseudo-random number or a coordinate of the low-discrepancy sequence x and need to find the corresponding index among a column, we find integer r:

$$r/2^k <= x < (r+1)/2^k$$

- take from the table the indices that correspond to r and r + 1 and than perform binary search to find the exact non-zero elements needed.

Feachers of MCMs and QMCMs for matrix computations

- Usually based on splitting the Markov chains among processors
- When multiple right-hand-sides are present - they can be divided among the processors
- When accelerators are used the available memory can be less than what CPU-base servers offer, so for parallel implementation combination of OpenMP and MPI can be used
- Spitting along the right-hand-sides between the different MPI processes and then along Markov chains (trajectories) between the OpenMP threads.
- Blocking parallelization approach for the quasi-Monte Carlo algorithm can be used.

Tuning the parallel execution of QMC matrix algorithms on MIC

- Many parameters of the MPI system are available for tuning by the user
- The I_MPI_ADJUST_BCAST parameter is important for decreasing the time for matrix broadcast
- Selection of I_MPI_DAPL_PROVIDER - ofa-v2-mcm-1 vsofa-v2-mlx4_0-1
- ***The broadcasting of the matrix takes most of the communication time, but the data is best packed before sending, saving bandwidth***

Numerical results

- Mean squared error when solving Ax = B, where A is a sparse matrix with 11 283 503 nonzeros elements, dimension 155331 x 155331 and 100 right-hand-side vectors (B is a rectangular matrix 155331 x 100):

	eps=0.05	0.01	0.005	0.001
Sobol	4.21e-005	4.28e-005	4.26e-005	4.34e-005
MT2203	9.67e-006	6.61e-006	5.87e-006	5.64e-006
Halton	8.08e-006	3.68e-006	2.64e-006	2.61e-006

Numerical results

- Comparison of execution times and speedup using different number of threads for sparse matrix

Table 5: Comparison of execution times and speedup using different number of threads for sparse matrix

Cards	Threads	Sobol		MT2203		Halton	
		Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
1	1	109.05		107.61		132.12	
	61	3.21		3.23		3.58	
	122	2.97	1.08	2.99	1.08	3.24	1.10
	244	3.71	0.87	3.69	0.87	3.83	0.93
2	1	52.12		56.33		51.62	
	61	1.92	1.67	2.14	1.51	2.11	1.69
	122	1.63	1.97	1.75	1.85	1.93	1.85
	244	8.32	0.38	8.89	0.36	8.35	0.43
8	1	14.03		13.72		16.11	
	61	0.55	5.83	0.54	5.98	0.63	5.68
	122	0.69	4.65	0.62	5.21	0.67	5.34
	244	1.03	3.12	0.93	3.47	1.00	3.58
16	1	7.24		7.53		9.00	
	61	0.38	8.45	0.35	9.23	0.42	8.53
	122	0.49	6.55	0.44	7.34	0.53	6.75
	244	0.68	4.72	0.80	4.04	0.84	4.26

Conclusion

- Vectorization by hand is the best way to make use of the power of Intel Xeon Phi accelerators.
- The benefits are substantial.
- Careful tuning allows to save memory that may be required by other parts of the parallel application.
- Quasi-random numbers can be substantially faster than popular pseudo-random number generators.