Proceedings of the Federated Conference on Computer Science and Information Systems pp. 665–668
DOI: 10.15439/2016F345
ACSIS, Vol. 8. ISSN 2300-5963

Data Structures for Markov Chain Transition Matrices on Intel Xeon Phi

Beata Bylina, Joanna Potiopa
Department of Computer Science, Maria Curie-Skłodowska University,
Plac M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland
Email: {beatas, joannap}@hektor.umcs.lublin.pl

***Abstract*—We employ Intel Xeon Phi as a high-performance coprocessor to solve Markov chains. Matrices arising from Markov models are very sparse with short rows. In this paper, the authors research two storage formats of Markov chain transition matrices on Intel Xeon Phi. In this work CSR and HYB (modification ELL) formats for such matrices are studied. Numerical experiments results for transition matrices of Markov chains from wireless networks and call-center models show that HYB format in offload version is more effective than CSR format. The obtained performance for HYB format is even `1.45` times better in comparison to multi-threaded CPU (dual Intel Xeon E5-2670) with the use of the CSR format (SpMV from the MKL library on CPU).**

I. Introduction

MARKOV chains are a tool for modeling various natural complex systems as well as computer systems and networks. Lately, they have been used to model wireless networks [3], [5], [6] and they often appear in computational biology [10] as well as in modeling call-centers [9].

In Markov modeling the models are very large because of exponential explosion of the states number, which happens due to the fact that complex systems usually consist of a certain number of subsystems and the states' space size of the whole complex system is usually exponentially dependent on the number of subsystems.

Any Markov chain can be described in terms of linear algebra with the use of a square matrix. A transition rate matrix $\mathbf{Q}$ (describing a Markov chain which models a system or a phenomenon) has some particular properties. It is a huge one and very sparse (with short rows). Sparse matrices are stored in special data structures and special algorithms are used to process these structures optimally. We can find descriptions of many such storage schemes in the literature (e.g. [14], [4], [2]).

The problem of efficiency of the sparse matrix-vector multiplication operation (SpMV) on Intel Xeon Phi was considered in [13], [12], [8]. In paper [13], the performance of the Intel Xeon Phi coprocessor for SpMV is investigated. One of the studied aspects in this work is CSR format for the sparse matrices. The authors showed that this format is not suited for Intel Xeon Phi for very sparse matrix (with short rows) in particular. The use of OpenMP based parallelization on Intel MIC (Intel Many Integrated Core Architecture) was evaluated in [8]. An efficient implementation of SpMV on the Intel Xeon Phi coprocessor by using a specialized ELLPACK-based format with load balancing is described in [12]. This implementation outperforms the implementation using CSR format even for matrices with very short rows.

The aim of this work is to shorten the computation time for the transition matrices from Markovian models of complex systems on Intel Xeon Phi by way of application of two data structures to store sparse matrices. Namely, CSR format will be used as in the works of [13], [8] and HYB format (as ELLPACK modification) similar as in the work [12]. HYB format was analyzed on GPU [7] and was much more effective than CSR for Markov chains.

Sparse matrix-vector multiplication (SpMV) operation on Intel Xeon Phi is implemented for these formats using the thread-level and the SIMD parallelism. Next, these SpMV's implementations are employed to the explicit fourth-order Runge-Kutta method. The numerical experiments were conducted for two groups of transition rate matrices, namely for a model of a call-center and a model of a wireless network on Intel Xeon Phi. A comparative analysis is also done with the CSR format from the Intel MKL library (Intel Math Kernel Library) on multithread CPU (dual Intel Xeon E5-2695).

The structure of the article is the following. Section II gives characteristics of the sparse matrix storage, chiefly CSR and HYB formats. Sections III and IV contain a description of the explicit fourth-order Runge-Kutta method and some details of its implementation on Intel Xeon Phi in particulary sparse matrix-vector multiplication operation (SpMV). Section V analyzes Intel Xeon Phi's performance on this kernel for two data structures (CSR and HYB). Section VI presents conclusions.

II. Storage of a sparse matrix

In the literature [14], a lot of ways which represent sparse matrices and enable their effective storage and processing have been suggested.

One of the formats to store any sparse matrices is *Compressed Sparse Row* (CSR). The operations on matrices stored in this format are part of Intel MKL library in version on Intel MIC architecture [11]. In this format, the information about $\mathbf{A}$ matrix, where $\mathbf{A}$ is a sparse matrix of $m \times n$ size and nz nonzero elements, is stored in three one-dimentional arrays:

- $data[\cdot]$, of nz size stores values of nonzero elements (in increasing order of row indices);
- $col[\cdot]$, of nz size stores column indices of nonzero elements (in order conforming to the data array content);

- $ptr[\cdot]$, of $m+1$ size, stores indices of beginnings of successive rows in $data$ array — that is $data[ptr[i]]$ is the first nonzero element of i-th row in $data$ array, similar, $col[ptr[i]]$ is the column number of this element.

Hybrid format (HYB) comprises of two other formats of sparse matrix storage in the memory disregarding its weaknesses and simultaneously making use of its advantages: COO format (*Coordinate format*) and ELLPACK package format (ELL).

ELLPACK [1] package format (ELL) is a sparse matrix format which is helpful in vector architecture. It is useful for matrices in which the number of the elements in almost every row is the same, especially when many rows reach maximum length in a given matrix or approach it, it becomes useless when the number of elements in a row is dispersed, e.g. when there are many rows which are longer than the mean. In this format the sparse matrix is stored in two two-dimensional arrays.

In HYB format the matrix is stored in two two-dimensional arrays (ELL) and three one-dimensional arrays (COO):

- $ell\_data[\cdot]$ stores values of nonzero elements as two-dimensional rectangular array of $M \times MNNZ$ size, where M is the number of rows in the matrix and $MNNZ$ denotes the mode of number of nonzero elements in a row. The rows with fewer nonzero elements than $MNNZ$ are aligned to the left and filled with zero (meaningless) values in the remaining part, while the longer rows are cut off.
- $ell\_indices[\cdot]$ stores column indices of a matrix elements placed appropriately in $ell\_data[\cdot]$. The size and the structure of this array is the same as $ell\_data[\cdot]$.
- $coo\_data[\cdot]$ stores nonzero elements values, which were cut off from $ell\_data[\cdot]$.
- $coo\_col[\cdot]$ stores column indices of nonzero elements from $coo\_data[\cdot]$ (in the same order as $coo\_data[\cdot]$).
- $coo\_row[\cdot]$ stores row indices of nonzero elements from $coo\_data[\cdot]$ (in the same order as $coo\_data[\cdot]$).

III. Parallel Runge-Kutta algorithm

General form explicit fourth-order Runge-Kutta method in parallel version is presented as Algorithm 1.

The implementation of the Algorithm 1 contains our implementation of SpMV operation and vector addition. We use the OpenMP standard and the `for` directives to parallelize all operations. We use a `static` scheduler for the distribution of the matrix rows and the values of vector. The SpMV operation is a simple task to assign a row block to a single thread in a parallel execution. The idea of vectorization is to process all the nonzero elements in row at once. Since the Intel Xeon Phi architecture has 32 512-bit registers, the matrices should have at least 8 values in each row to fully utilize the register. For one-row block we use a pragma compilator `#pragma simd` which enforces vectorization of the inner loops. This vectorization is not effective because our matrices have got short rows — shorter than 8 elements (see table I).

Algorithm 1 The parallel algorithm which determines the transient probabilities vector, where $*$ operation denotes parallel sparse matrix-vector multiplication and $+$ operation denotes parallelized and vectorized vector addition

```
Require: $Q^T$ — transition rate matrix, $pi_0$ — initial probability vector, $h$ — step, $t$ — time
Ensure: vector of transient probabilities $pi_t$ in the time $t$
 1: $lk \leftarrow t/h$
 2: $pi_t \leftarrow pi_0$
 3: for $k = 1$ to $lk$ do
 4:     $k_1 \leftarrow Q^T * pi_t$
 5:     $k_2 \leftarrow Q^T * (pi_t + \frac{h}{2}k_1)$
 6:     $k_3 \leftarrow Q^T * (pi_t + \frac{h}{2}k_2)$
 7:     $k_4 \leftarrow Q^T * (pi_t + hk_3)$
 8:     $pi_t = pi_t + \frac{h}{6} \cdot (k_1 + 2k_2 + 2k_3 + k_4)$
 9: end for
10: return $pi_t$
```

IV. Numerical experiment

In this section we tested the time, the speedup and the performance of the explicit fourth-order Runge-Kutta method (RK4). The programs were implemented in C++ language and three implementations of this algorithm were created:

- MKL-CSR version — it is a version using parallelism and vectorization offered by the function of the Intel MKL library in the version of Intel MIC architecture, where the sparse matrix was stored in CSR format.
- CSR version — it is a version, where the sparse matrix was stored in CSR format; all vector and matrix operations were implemented by the authors.
- HYB version — it is a version, where the sparse matrix was stored in HYB format; all vector and matrix operations were implemented by the authors.

The impact of the program execution mode (native and offload) and the various numbers of threads were tested.

For each version, the program was compiled by using the Intel C++ compiler (`icc`) with a compiler flag `-O3`, which resulted in automatic computing vectorization, `-openmp`, `-mmic` (enabling the cross compilation needed for a native execution on Intel Xeon Phi). Additionally, during development, we used the flag `-vec-report2` to verify whether the RK4 kernel was successfully vectorized. In every case, alignment of the memory data was used as vectorization support; the data were aligned with 64 bytes limit, which was recommended by the documentation. `-mkl` option was also used to allow introducion of parallelism in MKL-CSR version. Intel MKL library was applied to measure the elapsed time.

In the table I, the properties of matrices used during the test are given: WF1, WF2 describe wireless networks, CC1, CC2 describe the call-center.

The matrices we tested are very sparse, however, the pattern varies in dependence on the model (table I). L_i, $1 \leq i \leq n$, denoted the number of nonzero elements per row. CC matries

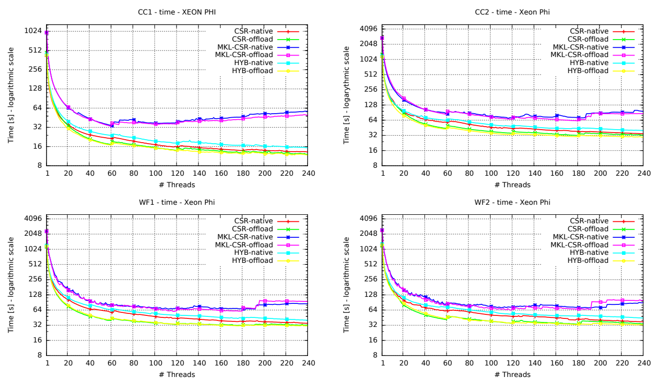

Fig. 1. Runtime of explicit fourth-order Runge-Kutta method on Intel Xeon Phi for CC1, CC2, WF1 and WF2 matrices

have similar number of nonzero elements per row (3—6). WF matrices have different number of elements per row.

TABLE I
THE PROPERTIES OF THE TESTED MATRICES

No	Name	n	nz	$\frac{nz}{n}$	$\min L_i$	$\max L_i$
1.	CC1	335421	1996701	5.95	3	6
2.	CC2	937728	5588932	5.56	3	6
3.	WF1	962336	4434326	4.61	1	12
4.	WF2	1034273	4660479	4.51	1	11

All tested matrices have very short rows; the mean number of elements in a row is between 4.51 and 5.95 elements. The input arrays size is long enough and we provide enough work for each thread.

The tests were carried out using computing node of the following parameters:

- Platform: Intel Server Chassis R2000WTXXX, Intel Server Board S2600WT2
- CPU: 2xIntel Xeon E5-2670 v3 (2x12 cores, 2.3 GHz)
- Memory: 128 GB DDR4 2133MT/s (8xCrucial CT16G4RFD4213)
- Network card: FDR InfiniBand ConnectX-3 Mellanox AXX1FDRIBIOM (FDR 56GT/S)
- Coprocessor: Intel Xeon Phi Coprocessor 7120P (16GB, 1.238 GHz, 61 cores)
- Software: Intel Parallel Studio XE 2016 Cluster Edition for Linux (Intel C++ Compiler, Intel Math Kernel Library, Intel OpenMP)

V. RESULTS

In this section we evaluate the time, the speedup and the performance of our approach to the problem of the different ways of sparse matrices storage, two execution modes for various numbers of threads.

A. Time

Fig. 1 presents execution time of RK4 algorithm. It is obvious that independently of the models and matrices size, the version using MKL library routines is the slowest. In case of our implementations offload versions are the fastest but there is no clear difference between CSR and HYB formats. Our implementations (in offload version) always perform about two times faster than MKL-CSR version, for a similar number of threads (except CC1 matrix, when execution time for 240 threads in HYB-offload version and CSR-offload is even four times faster than MKL-CSR-offload version).

For every solution, the time for the first 60 threads decreases most rapidly (with 1 thread per core activated). The gain with the use of a large number of threads per core is meaningless.

We obtain the best time for each matrix for HYB-offload version. Basing on the received charts, it seems that for MKL-CSR version the size matrix is essential; for CC1 matrix execution time increases with 60 threads and for the remaining matrices with 3 times bigger size, the time increases with 180 threads. In case of our implementations, execution time is even independent of the matrix size and model.

B. Speedup

Fig. 2 shows the speedup of RK4 method on Intel Xeon Phi with respect to a sequential version running on one thread of Intel Xeon Phi. The speedup for CC1 matrix gives a bit different charts in comparison with other matrices. It is due to a small matrix size in relation to others. For the number of threads from 1 to 60, HYB-native and CSR-native implementations give the lowest speedup, for other version the results are similar.

For more than 60 threads we can see that the speedup of the MKL-CSR version decreases. Moreover, for over 140 threads it gives the poorest results. With 60 to 240 threads (2-4 threads per core) we can see that CSR-offload and HYB-offload perform the best (with minimal superiority of the first implementation).

CC1 matrix achieves maximum speedup (which is 40) for CSR-offload version with 240 threads. The other matrices (CC1, WF1, WF2) have similar sizes and their speedup charts look similar. The lowest speedup is obtained for the CSR-native and HYB-native versions with 1 to 180 threads (1-3 threads per core).

The remaining implementations give similar results. The difference appears when we start over 180 threads (4 threads per core). HYB-offload and CSR-offload have the best speedup while MKL-CSR (native and offload version) clearly decrease. The best achieved speedup is 44 for CC2 matrix in MKL-CSR-offload version with 180 threads. For WF matrices the lowest speedup is 39 for WF1 and almost 40 for WF2 in MKL-CSR-offload version.

C. Performance

Fig. 3 presents the performance of RK4. Each of the figure's bars shows maximum performance obtained for a given stored matrix in a given format and for the program executed in the given mode. In comparison we also present the performance of RK4 method on CPU where all matrix and vector operations were realized with the use of the kernels from MKL library (denoted MKL-CSR-CPU).

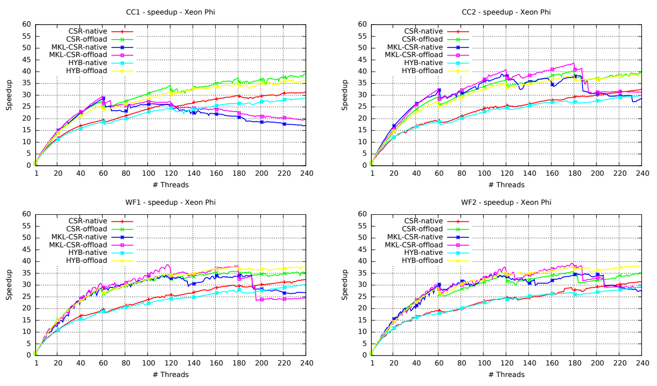

Fig. 2. Speedup of explicit fourth-order Runge-Kutta method on Intel Xeon Phi with respect to a sequential version running on one thread Intel Xeon Phi for CC1, CC2, WF1 and WF2 matrices

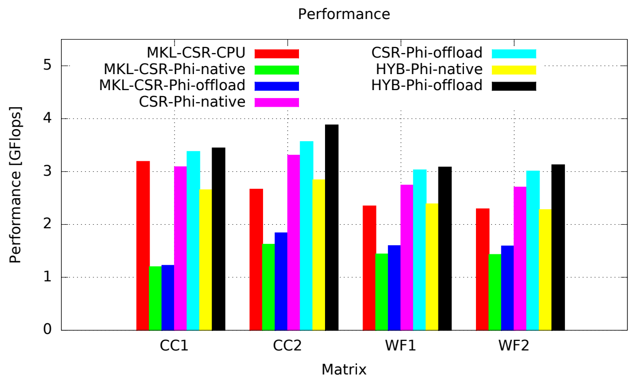

Fig. 3. Performance of explicit fourth-order Runge-Kutta method on Intel Xeon Phi

The results analysis shows a clear difference between the performance of our implementations as opposed to MKL-CSR version on Intel Xeon Phi. We can also notice that independently of the matrix size and storage format (MKL-CSR, CSR, HYB) we obtained better performance when starting the application in offload version. The biggest differences between native and offload modes occurred for HYB format. In case of our implementations we achieved the best results for every matrix in HYB-Phi-offload version. Moreover it was from 1.9 to 2.8 times more efficient than MKL-CSR-Phi-offload version due to the fact that our implementations is less general and enables better control over multithreading and vectorization.

The matrices generated for call-center model achieve better performance than the matrix from wireless network models. Despite the size, is connected with slightly higher matrix density and more regular pattern (tab. I).

Our implementations also have better performance in relation to MKL-CSR-CPU; with small matrix CC1 the differences become insignificant, but for the bigger matrices our approach is clearly favourable even up to 50% for HYB-Phi-offload version.

VI. Conclusion

In this article we investigated the use of two sparse matrices format storage in context of Markov chain problems for accelerating on the Intel Xeon Phi. Our approach exploits the thread-level parallelism and vectorization for SpMV operation and the thread-level parallelism and vectorization for the vector addition.

Based on the conducted experiments, we can clearly state that the Intel MKL library for Intel MIC architecture performed worse than our own CSR and HYB implementations.

Our implementations significantly outperform the optimized implementation routine SpMV from Intel MKL library using the CSR format on Intel Xeon Phi. The CSR and HYB versions are scalable to a large number of threads and they use all the cores on Intel Xeon Phi. We achieve the best performance for HYB format offload version.

Our implementation can still be improved. In future works we will employ thread affinity strategies which allowe to improve the performance and scalability of our approach.

References

[1] ELLPACK, 2014. http://www.cs.purdue.edu/ellpack.
[2] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical report, NVIDIA, 2008. Tech. Report No. NVR-2008-004.
[3] G. Bianchi. Performance analysis of the IEEE 802.11 distributed coordination function. *IEEE Journal on Selected Areas in Communications*, 18(3):535–547, March 2000.
[4] B. Bylina, J. Bylina, and M. Karwacki. Computational aspects of GPU-accelerated sparse matrix-vector multiplication for solving Markov models. *Theoretical and Applied Informatics*, 23(2):127–145, 2011.
[5] J. Bylina and B. Bylina. A Markovian queuing model of a WLAN node. *Communications in Computer and Information Science*, 160:80–86, 2011.
[6] J. Bylina, B. Bylina, and M. Karwacki. A Markovian model of a network of two wireless devices. *Communications in Computer and Information Science*, 291:411–420, 2012.
[7] Jarosław Bylina, Beata Bylina, and Marek Karwacki. An efficient representation on GPU for transition rate matrices for Markov chains. In *Parallel Processing and Applied Mathematics*, pages 663–672. Springer, 2013.
[8] Tim Cramer, Dirk Schmidl, Michael Klemm, and Dieter an Mey. OpenMP programming on Intel Xeon Phi coprocessors: An early performance comparison. In *Proc. of the Many-core Applications Research Community Symposium at RWTH Aachen University*, pages 38–44, 2012.
[9] N. Gans, G. Koole, and A. Mandelbaum. Telephone call centers: tutorial. *Review and Research Prospects, Manufact. and Service Oper. Manag.*, 5:79–141, 2003.
[10] N. A. Hamilton, K. Burrage, and A. Bustamam. Fast parallel markov clustering in bioinformatics using massively parallel computing on gpu with cuda and ellpack-r sparse format. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 9(3):679–692, 2012.
[11] Intel. Intel Math Kernel Library (MKL). http://software.intel.com/en-us/intel-mkl, 2014.
[12] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 273–282, New York, NY, USA, 2013. ACM.
[13] Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. *CoRR*, abs/1302.1078, 2013.
[14] W. J. Stewart. *An Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, Princeton, NJ, 1994.