Int J Parallel Prog (2017) 45:1073–1090
DOI 10.1007/s10766-016-0458-x

Explicit Fourth-Order Runge–Kutta Method on Intel Xeon Phi Coprocessor

Beata Bylina[1] · **Joanna Potiopa**[1]

Received: 29 February 2016 / Accepted: 22 September 2016 / Published online: 29 September 2016

Abstract This paper concerns an Intel Xeon Phi implementation of the explicit fourth-order Runge–Kutta method (RK4) for very sparse matrices with very short rows. Such matrices arise during Markovian modeling of computer and telecommunication networks. In this work an implementation based on Intel Math Kernel Library (Intel MKL) routines and the authors' own implementation, both using the CSR storage scheme and working on Intel Xeon Phi, were investigated. The implementation based on the Intel MKL library uses the high-performance BLAS and Sparse BLAS routines. In our application we focus on OpenMP style programming. We implement SpMV operation and vector addition using the basic optimizing techniques and the vectorization. We evaluate our approach in native and offload modes for various number of cores and thread allocation affinities. Both implementations (based on Intel MKL and made by the authors) were compared in respect of the time, the speedup and the performance. The numerical experiments on Intel Xeon Phi show that the performance of authors' implementation is very promising and gives a gain of up to two times compared to the multithreaded implementation (based on Intel MKL) running on CPU (Intel Xeon processor) and even three times in comparison with the application which uses Intel MKL on Intel Xeon Phi.

The authors are grateful to Czestochowa University of Technology for granting access to Intel CPU and Intel Xeon Phi platforms providing by the MICLAB Project No. POIG.02.03.00.24-093/13.

✉ Beata Bylina
beatas@hektor.umcs.lublin.pl

Joanna Potiopa
joannap@hektor.umcs.lublin.pl

1 Department of Computer Science, Maria Curie-Skłodowska University, Plac M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland

Keywords Intel Xeon Phi · Fourth-order Runge–Kutta method · CSR format · Intel Math Kernel Library (Intel MKL) · SpMV · OpenMP

1 Introduction

In recent years HPC computers are increasingly equipped with computation accelerators responsible for performing some operations in parallel. Accelerators based on graphic processing units (GPU) [26] are characterized by a very specific architecture and require specially designed programing tools and environments (like CUDA [12] and/or OpenCL [32]). Another type of coprocessors is Intel Xeon Phi [21] which can run the existing code without changes—only after recompilation.

Theoretically, thanks to such features, we could use Intel Xeon Phi for large scale parallel processing without the necessity of redesigning codes, because these coprocessors support traditional programming models. However, practically, to make a full use of the computational potential of massively parallel many-core systems we must quite often put a big effort and apply a lot of different optimization techniques to take advantage of the parallelism hidden in the code.

Modeling real complex systems with the use of Markov chains is a well-known and recognized method giving good results [31]. Examples of complex systems considered in this article are *call centers* [14,27,33] and wireless networks [3,8,9]. For large matrices (and such matrices arise during modeling complex system), the methods based on numerical solving of ordinal differential equations are the most useful [4,17,28,31]. There exist one-step methods (such as Euler method, its modifications, as well as Runge–Kutta methods) and multistep methods (like Adams method [31] or BDF method [31]).

In the following paper we use one of the Runge–Kutta methods—the explicit fourth-order Runge–Kutta method (RK4)—because of their accuracy and flexibility, and the possibility of changing the integration step. However, these methods have a disadvantage, namely relatively long time of computation. The dominant operations of the Runge–Kutta methods are SpMV (sparse matrix-vector multiplication) and vector addition. Gaining a good performance of SpMV is difficult on almost each architecture. We are going to perform multiplication of sparse matrix-vector (SpMV) on Intel Xeon Phi to speedup computation.

In our previous works [5,7,10,22] we have considered the numerical solution of Markov chains on different architectures: multi-core and GPUs. The novelty of this paper is a research of parallel implementations of explicit fourth-order Runge–Kutta method for sparse matrices arising from Markovian models on a new architecture, namely Intel Xeon Phi.

The aim of this work is to shorten the computation time of the explicit fourth-order Runge–Kutta methods for the matrices from Markovian models of complex systems with the use of the massively parallel many-core architecture of Intel Xeon Phi. We use the CSR format to represent sparse matrices. We present two implementations: one based on Intel MKL which uses BLAS and Sparse BLAS routines and the second, where SpMV operation was implemented by the authors. In this work, the time, the speedup and the performance are analyzed.

The paper is organized as follows. Section 2 presents related works. Section 3 introduces the Intel Xeon Phi architecture. In Sect. 4 the idea of the sparse matrix storage in the CSR format is presented. Section 5 provides information about the explicit fourth-order Runge–Kutta method and about the parallel algorithm for this method and its implementation. Section 6 presents the results of our experiments. The time, the speedup, the performance of two programming modes on the Intel Xeon Phi are analyzed. Section 7 is a summary of our experiments.

2 Related Works

The SpMV operation was studied on various architectures. Due to the popularity of GPUs, sparse matrix formats and different optimization techniques was proposed to improve the performance of SpMV on GPUs.

In the article [5] some computational aspects of GPU-accelerated sparse matrix-vector multiplication were investigated. Particularly, sparse matrices appearing in modeling with Markovian queuing models were considered. The efficiency of SpMV with the use of a ready-to-use GPU-accelerated mathematical library, namely CUSP [11] was studied. For the CUSP library, some data structures of sparse matrices and their impact on the GPU were discussed. The SpMV routine from the CUSP library was used for the implementation of the uniformization method. The uniformization method is one of the methods for finding transient probabilities in Markovian models. It was analyzed in the work [7] on a CPU–GPU architecture. Two parallel algorithms of the uniformization method—the first one utilizing only a multicore machine (CPU) and the second one, with the use of not only a multicore CPU, but also a graphical processor unit (GPU) for the most time-consuming computations—were presented. The uniformization method on a multi-GPU machines was considered in the work [22].

New algorithms for performing SpMV on multicore and multinodal architectures were presented in the paper [6]. A parallel version of the algorithm which can be efficiently implemented on the contemporary multicore architectures was considered. Next, a distributed version targeted at high performance clusters was shown. Both versions were thoroughly tested using different architectures, compiler tools and sparse matrices of different sizes. The performance of the algorithms was compared to the performance of the SpMV routine from the widely known MKL library.

The problem of efficiency of the SpMV operation on Intel Xeon Phi was considered in [13,25,30]. In paper [30], the performance of the Xeon Phi coprocessor for SpMV is investigated. One of the researched aspect in that work is the vectorization of the CRS format and showing that this approach is not suited for Intel Xeon Phi in particular for very sparse matrix (with short rows). An efficient implementation of SpMV on the Intel Xeon Phi coprocessor by using a specialized data structure with load balancing is described in [25]. The use of OpenMP based parallelization on a MIC (Intel Many Integrated Cores architecture) processor was evaluated in [13]. That work analyzed the speedup, throughput, scalability of the OpenMP version of the CG (conjugate gradients) kernel, which used the SpMV operation on Intel Xeon Phi and was application oriented.

3 Xeon Phi Architecture and Programming Models

Intel Xeon Phi [29] is a multicore coprocessor created on the basis of Intel MIC (Many Integrated Cores) technology, where many redesigned Intel CPU cores are connected by a bi-directional 512-bit ring bus. The cores are enriched with a 64-bit service instruction and a cache memory L1 and L2. Additionally, the cores ensure hardware support for FMA (Fused Multiply-Add) instruction and also have their own vector processing unit (VPU), which together with 32,512-bit vector registers allows to process many data with the use of one instruction (SIMD).

A single Intel Xeon Phi is made in 22 nm technology with the use of 3-DTri-Gate transistors. It has 57–61 cores of 1056–1238 GHz frequency and it serves 244 threads and communicates through PCI-Express 2.0. Advanced mechanisms of energy management are implemented.

The Intel accelerators are characterized by a typical memory hierarchy. Depending on a card model, the coprocessor has from 6 to 16 GB of main memory GDDR5 (Graphic Double Data rate 5 v). The access to this memory is gained through 6–8 main memory controllers, each having two access channels enabling sending 2×8 bites. The access to the main memory is characterized by 240–352 GB/s. To maximize the bandwidth, the memory data are organized in a specific way.

Intel Xeon Phi enables execution of applications written in C/C++ and Fortran languages. The Intel company offers a set of programming tools assisting programming processes such as compilers, debuggers, libraries that allow creating parallel applications (e.g. OpenMP, MPI) and different kinds of mathematic libraries (e.g. Intel MKL).

Intel Xeon Phi coprocessors cannot be used as independent computing units (they require a general purpose processor), however, they can work in different executing modes: native or offload mode.

In the native mode the task is executed directly by the coprocessor, which makes it a separate computing node. The compilation of the source code for the accelerator architecture demands a so-called cross-compliling, which produces an executing file on Intel Xeon Phi. The native application can be started by hand on the coprocessor or by `micnativeloadex` tool which automatically copies the program together with necessary files and then starts it.

The offload programming model allows designing programs in which only selected segments of the code are executed by the coprocessors. The chosen part of the code should be proceeded by a dedicated compiler directive `#pragma offload target(mic)` that also indicates available coprocessors which will be used to do calculations and send data between the coprocessor and the host. The program is compiled like a regular host application and is initiated on the host processor while the code segments which will be done by the coprocessor are automatically copied during the application performance and consequently started there.

Within a single core of the Intel MIC architecture it is possible to start maximum up to 4 threads which share the same memory cache. Thus, the essential aspect is to define the way in which processes or threads will be mapped on to a computing unit cores (affinity). The optimal setting enables the reduction of reference numbers to main memory consequently increasing the performance. In OpenMP standard it can be achieved by setting an environmental variable `KMP_AFFINITY` on one of the options:

- `compact` — the successive core is filled in with threads after assigning 4 threads to a former core,
- `balanced` — threads are placed equally between the available computing cores,
- `scatter` — threads are placed between the core based on *round and robin* algorithm.

4 Storage of a Sparse Matrix

Matrices that are generated while solving Markovs models of complex systems are very sparse, with a small number of entries in a row. In the literature [31], a lot of ways which represent sparse matrices and enable their effective storage and processing have been suggested. Generally, there is no single best way to represent sparse matrices as different data structures depend on different types of sparse matrices and different algorithms, and also some data structures turn out to be more susceptible to parallel implementation than others.

One of the formats to store any sparse matrices is *Compressed Sparse Row* (CSR). This format uses little space in the operational memory. Additionally, the operations on matrices stored in this format are part of the Intel MKL library on the Intel MIC architecture [20]. In the CSR format, the information about matrix **A**, where **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 $data$ 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 ith row in $data$ array, similary, $col[ptr[i]]$ is the column number of this element. Moreover, ptr array usually stores an additional element that equals the number of nonzero elements in the whole matrix at the end, which is incredibly useful for processing the CSR format.

5 Explicite Fourth-Order Runge–Kutta Method

In the case of computing transient probabilities in a continuous time Markov chain (CTMC), numerical techniques are based on solving the system of ordinary differential equations of order n:

$$\frac{d\boldsymbol{\pi}(t)}{dt} = \mathbf{Q}^T \boldsymbol{\pi}(t) \tag{1}$$

where the coefficient matrix $\mathbf{Q}^T$ is transition rate matrix of order n (n is the number of states of the Markov chain) and $\boldsymbol{\pi}(t)$ is the state probability vector of CTMC at time t.

For the system of Eq. (1) there is analytical solution in the following form:

$$\boldsymbol{\pi}(t) = \boldsymbol{\pi}(0)e^{\mathbf{Q}^T t} \tag{2}$$

where $\boldsymbol{\pi}(0)$ is the initial condition and is the initial probability vector of CTMC.

Computing expression $e^{\mathbf{Q}^T t}$ creates the problem for large sparse matrices. For determining this expression, we expand exponential function in infinite Taylor series

$$e^{\mathbf{Q}^T t} = \sum_{k=0}^{\infty} \frac{(\mathbf{Q}^T t)^k}{k!} \tag{3}$$

The complication during determining the formula (3) is connected with computing the k-th matrix power of $\mathbf{Q}^T$; such an algorithm is numerically instable.

We consider the explicite fourth-order Runge–Kutta method for numerical solving the ordinary differential Eq. (1) with the initial conditions (2).

5.1 Runge–Kutta Method

The most commonly used Runge–Kutta method is the explicit four order method. It has a standard form expressed by the formulas:

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4), \tag{4}$$

where

$$\begin{aligned}
k_1 &= f(t_i, y_i),\\
k_2 &= f\left(t_i + \frac{h}{2}, y_i + \frac{hk_1}{2}\right),\\
k_3 &= f\left(t_i + \frac{h}{2}, y_i + \frac{hk_2}{2}\right),\\
k_4 &= f(t_i + h, y_i + hk_3).
\end{aligned}$$

When we apply the standard fourth-order explicit Runge–Kutta method to Eq. (1), we obtain the following formulas:

$$\boldsymbol{\pi}_{i+1} = \boldsymbol{\pi}_i + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4), \tag{5}$$

where

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{Q}^T\boldsymbol{\pi}_i,\\
\mathbf{k}_2 &= \mathbf{Q}^T\left(\boldsymbol{\pi}_i + \frac{h\mathbf{k}_1}{2}\right),
\end{aligned}$$

Fig. 1 The explicite fourth-order Runge–Kutta method

```
1. $\mathbf{p} \leftarrow \boldsymbol{\pi}_0$;
2. For $k = 1, \ldots, t/h$ do:
    (a) $\mathbf{k}_1 \leftarrow \mathbf{Q}^T \mathbf{p}$
    (b) $\mathbf{k}_2 \leftarrow \mathbf{Q}^T(\mathbf{p} + h\mathbf{k}_1/2)$
    (c) $\mathbf{k}_3 \leftarrow \mathbf{Q}^T(\mathbf{p} + h\mathbf{k}_2/2)$
    (d) $\mathbf{k}_4 \leftarrow \mathbf{Q}^T(\mathbf{p} + h\mathbf{k}_3)$
    (e) $\mathbf{p} \leftarrow \mathbf{p} + \frac{h}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)$
3. $\boldsymbol{\pi}(t) \leftarrow \mathbf{p}$
```

$$\mathbf{k}_3 = \mathbf{Q}^T \left(\boldsymbol{\pi}_i + \frac{h\mathbf{k}_2}{2} \right),$$
$$\mathbf{k}_4 = \mathbf{Q}^T (\boldsymbol{\pi}_i + h\mathbf{k}_3).$$

The algorithm to compute the vector $\boldsymbol{\pi}(t)$ of transient probabilities in a given time t from the formula (5) has been presented in Fig. 1. In the algorithm we have: the matrix $\mathbf{Q}$ (the infinitesimal generator) and the initial condition $\boldsymbol{\pi}(0)$, the time t and the step h.

Recommendation for Runge–Kutta method [23]: the method is very accurate and most often applied. Its main advantage is the possibility to use a changeable integrate step. On the negative side, the Runge–Kutta methods take comparatively long time for computations and there are difficulties in error evaluation.

5.2 Parallel Runge–Kutta Algorithm

In the further part of this paper we will be dealing with the explicit fourth-order Runge–Kutta method in a parallel version. Its general form is presented as Algorithm 1.

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

```
Require: $Q^T$—transition rate matrix, $pi_0$—initial probability vector, $h$—step, $t$—time
Ensure: $pi_t$—vector of transient probabilities 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:     $k_2 \leftarrow Q^T * (pi_t + \frac{h}{2} \cdot k_1)$
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 $pi_t$
```

The total number of the floating point operations for the sparse matrix-vector multiplication is $2nz$ and for the Runge–Kutta method it is

$$8nz + 13n \tag{6}$$

5.3 Parallel Implementation

The first implementation was created on the basis of available functions in the Intel MKL (Math Kernel Library) library of computing functions on Intel processors and adapted to multithread parallel processing in multi-processor systems including Intel Xeon Phi. As the Eq. (5) is a vector equation, it allows us to express Algorithm 1 in terms of BLAS [2] (Basic Linear Algebra Subprograms) [1,24]. One function from Sparse BLAS package was used. Sparse BLAS [15,16] is a group of methods performing linear algebra operations on sparse matrices. In our implementation, we used BLAS package level 1 and one function from Sparse BLAS.

The matrix storage format in the memory was considered as the CSR format supported by the Intel MKL library. In the Intel MKL library the information about the matrix in the CSR format is stored in 4 arrays:

`values`—the nonzero matrix elements' array in a row order; its length equals the number of nonzero elements of the stored matrix;
`columns`—the column index array of nonzero elements from the `values` array; its length equals the length of the `values` array;
`pointerB`—the length of this array equals the number of rows of a stored matrix and each of its elements (equal to the number of the matrix row) gives an element index from the `values` array: the first nonzero element in a given row;
`pointerE`—this array also has the length equal to the number of rows in a stored matrix and each of its elements gives an element index from the `values` array: it is the first of nonzero elements in the next row.

Additionally, Intel MKL supports indexation both from zero and one in the `columns`, `pointerB`, `pointerE` arrays. While calling a function from the Sparse BLAS group, it is necessary to give all parameters describing the processed matrix. Moreover, the `matdescra` parameter which is a six-element char array containing additional information about the matrix, should be filled in.

`matdescra[0]`—information about the matrix structure (`G` for general matrix);
`matdescra[1]`—in case of a triangular matrix, the information about whether it is upper or lower triangular;
`matdescra[2]`—the type of the main diagonal;
`matdescra[3]`—information about the indexation type (`F` from one, `C` from zero)
`matdescra[4]`, `matdescra[5]`—not used, reserved for the future.

None of the matrices tested here has a characteristic structure, hence `matdescra[0]` == `G`, `matdescra[1]` and `matdescra[2]` are ignored and indexation starts from zero, so `matdescra[4]` == `C`.

For the implementation of the operation $*$ (that is a sparse matrix-vector multiplication) a function from Sparse BLAS 2 package was used:

– `mkl_dcsrmv`: $\mathbf{y} \leftarrow \alpha \cdot \mathbf{A} * \mathbf{x} + \beta \cdot \mathbf{y}$, where $\alpha = 1$, $\beta = 0$, while the vectorized vector addition (that is the operation $+$) was needed in the following form

$$\mathbf{x} \leftarrow a \cdot \mathbf{x} + \mathbf{y}$$

and was implemented with a combination of the following functions:

- `cblas_dcopy`: $\mathbf{y} \leftarrow \mathbf{x}$,
- `cblas_dscal`: $\mathbf{x} \leftarrow a \cdot \mathbf{x}$,
- `cblas_daxpy`: $\mathbf{y} \leftarrow a \cdot \mathbf{x} + \mathbf{y}$.

The headers of the functions described here are the same on different Intel architectures.

Next, we consider our implementations of the SpMV operation and the vector addition. The matrix $\mathbf{Q}$ is represented in CRS. 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 the vector.

In the SpMV operation we can assign some consecutive rows of the matrix to a single thread in a parallel execution. One of the limitations in the SpMV implementation without optimization option is that only a single nonzero element is processed at a time. To change it, we should switch on the `-O3` compiler option for the automatic vectorization. The automatic vectorization is able to change (if it is safe) scalar instructions into vector ones during compilation of the source code.

The idea of vectorization is to process all the nonzero elements in a 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 blocks we do not use a SIMD kernel, because the test matrices have very short rows' lengths (about 5 values per row) and the matrix-vector multiplication using our CSR kernel usually uses only a part of the SIMD slot. Thus, the low SIMD efficiency is a problem for CSR for matrices with short rows (see [13,25,30]).

In the algorithm we used the operation of the vector addition `x` $\leftarrow$ `alpha * x + y`. For this operation we applied the *strip-mining* technique—division of the loop into two loops nested, allowing to separate the multithread and the vectorization. The outer loop is parallelized using the pragma: `#pragma omp parallel for schedule(static)`. The `#pragma simd` pragma enables vectorization of the inner loop. In addition, the information about the data independence is passed by `#pragma ivdep`.

6 Numerical Experiment

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

the MKL-CSR version	it is a version using parallelism and vectorization offered by the function of the Intel MKL library in the version of the Intel MIC architecture, where the sparse matrix was stored in the CSR format.

Table 1 The properties of the tested matrices

Lp.	Name	n	nz	$\frac{nz}{n}$
1.	CC1	335,421	1,996,701	5.95
2.	CC2	937,728	5,588,932	5.56
3.	WF1	962,336	4,434,326	4.61
4.	WF2	1,034,273	4,660,479	4.51

the CSR version	it is a version, where the sparse matrix was stored in the CSR format; all vector and matrix operations were implemented by the authors.

The tests were conducted in different configurations:

– The impact of the program execution mode (native and offload) was tested.
– The impact of mapping the number of threads to the core (various settings of environmental variable `KMP_AFFINITY`) were analyzed.

For each version, the program was compiled by using the Intel C++ compiler (`icc`) with the compiler flag `-O3`, which resulted in the automatic vectorization. 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.

The `-mkl` option was also used, which allowed to introduce parallelism in the MKL-CSR version. The Intel MKL library was applied to measure the elapsed time. The application of a `const` qualifier and the reference to array elements instead of indices were the additional optimization elements applied to multithread processing. The elapsed time of the algorithm was measured along with the data allocation in the memory while all computations were done in double precision.

6.1 The Test Models

We tested the implementations on two models. The matrices generated while modeling real systems with the use of Markov chains were applied in the test. Markov chains can be used for modeling telecommunication networks and computer and network systems. As an example of telecommunication network, a model of a call-center was considered, while in case of computer and network systems we investigated a DCF mechanism model (ang. *distributed coordination function*). It is a part of IEEE [18] used to avoid collisions in wireless networks. Sparse matrices, which are used to carry out tests, were generated and written to files. Each file contains the size of the matrix in the first row and the number of nonzero elements (nz) in the second row while the information about nonzero elements' placement (row, column, value) is put in successive nz rows. In Table 1, the properties of matrices used during the test are given: WF1, WF2 describe wireless networks, CC1, CC2 describe call-centers.

All the tested matrices have very short rows; the mean number of elements in a row is between 4.51 and 5.95 elements. In both computing models the number of computing steps was the same and was equal 2000. However, due to the specific quality of each model it generated different step sizes, as the basic time units in the **Q** matrix were different for both models.

The parameters of the Algorithm 1 for wireless network models were the following:

- initial condition $pi_0 = [1, 0, \ldots, 0]$,
- step $h = 0.001$,
- time $t = 2$.

The parameters for the call center models were as follows:

- initial condition $pi_0 = [1, 0, \ldots, 0]$,
- step $h = 0.000001$,
- time $t = 0.002$.

In the call center model 1 h was a time unit, thus $t = 0.002$ denoted about 7 s. In the wireless network model 1 ms was a time unit, thus $t = 2$ denoted 2 ms.

6.2 The Test Environment

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

Platform Serwer Actina Solar 220 X5 (Intel R2208GZ4GC Grizzly Pass)
CPU 2x Intel Xeon E5-2695 v2 (2×12 cores, 2.4 GHz)
Memory 128GB DDR3 ECC Registred 1866MHz (16×8 GB)
Network card 2x InfiniBand: Mellanox MCB191A-FCAT(Connect-IB, FDR 56Gb/s)
Coprocessor 2x Intel Intel Xeon Phi Coprocessor 7120P (16GB, 1.238 GHz, 61 cores)
Software Intel Parallel Studio XE 2015 Cluster Edition for Linux (Intel C++ Compiler, Intel Math Kernel Library, Intel OpenMP)

6.3 Methodology

We use three metrics to compare the computing performance: *time-to-solution*, *speedup* and *performance*. Time-to-solution is the time spent to reach a solution of the explicit fourth-order Runge–Kutta method (RK4). Speedups (called relative speedups) are calculated by dividing the time-to-solution of RK4 with a single thread on a single core on Intel Xeon Phi by time-to-solution of RK4 with n threads on Intel Xeon Phi. The performance [Gflops] is calculated by dividing the total number of the floating point operation (6) by the best time-to-solution.

In our tests we use 60 cores in native and offload mode. In case of native execution model, when application is started directly on Xeon Phi card, we can use all available 61 cores, but when we execute our code in offload mode the last physical core (with all 4 threads on it) is used to run the services required to support data transfer for offload [19].

6.4 Affinity

The impact of mapping the number of threads to the core by different settings of the environmental variable `KMP_AFFINITY` for chosen matrices were analyzed both

Table 2 Thread-to-core mapping [time in seconds], *native* mode for MKL-CSR version

Matrix	Number of threads	No affinity	`KMP_AFFINITY`		
			`Compact`	`Balanced`	`Scatter`
CC1	60 Threads	34	53	33	34
	120 Threads	39	44	165	40
	180 Threads	45	49	228	49
	240 Threads	53	63	279	60
WF1	60 Threads	80	129	80	80
	120 Threads	66	84	65	71
	180 Threads	65	67	60	68
	240 Threads	86	74	332	86

Table 3 Thread-to-core mapping [time in seconds], *offload* mode for MKL-CSR version

Matrix	Number of threads	No affinity	`KMP_AFFINITY`		
			`Compact`	`Balanced`	`Scatter`
CC1	60 Threads	35	54	36	37
	120 Threads	38	45	124	40
	180 Threads	46	47	199	47
	240 Threads	53	53	423	74
WF1	60 Threads	82	145	85	91
	120 Threads	63	92	64	71
	180 Threads	60	70	62	76
	240 Threads	75	81	482	134

for the MKL-CSR implementation and the CSR version. Regarding the fact that there is a possible increase in performance accompanying thread-to-core mapping, the tests were carried out with different settings of the `KMP_AFFINITY` variable (`compact/balanced/scatter`), and also without a particular setting of this variable value. Tables 2 and 3 present the results of control check of thread-to-core mapping in the MKL-CSR version for the native and the offload modes respectively. The results suggest that when using a function from the optimized Intel MKL library the best solution is to avoid checking thread-to-core mapping so consequently the most optimal way to load resources is chosen.

Similar tests were carried out for the application in the CSR version. Tables 4 and 5 present the results for the native and offload modes respectively. In this version differences between settings of `KMP_AFFINITY` variable are not so clear. However, one can notice that the `KMP_AFFINITY=balanced` gives the best results.

Table 4 Thread-to-core mapping [time in seconds], *native* mode for CSR version

Matrix	Number of threads	No affinity	KMP_AFFINITY		
			Compact	Balanced	Scatter
CC1	60 Threads	16	21	15	16
	120 Threads	14	14	12	14
	180 Threads	12	11	11	13
	240 Threads	12	10	10	13
WF1	60 Threads	43	70	43	39
	120 Threads	37	44	35	36
	180 Threads	38	33	32	38
	240 Threads	39	29	31	37

Table 5 Thread-to-core mapping [time in seconds], *offload* mode for CSR version

Matrix	Number of threads	No affinity	KMP_AFFINITY		
			Compact	Balanced	Scatter
CC1	60 Threads	18	25	18	17
	120 Threads	15	16	14	14
	180 Threads	13	13	13	13
	240 Threads	13	11	11	12
WF1	60 Threads	40	61	38	39
	120 Threads	33	41	31	32
	180 Threads	30	34	30	30
	240 Threads	30	32	29	30

6.5 Results

Figures 2, 3, 4 and 5 represent the elapsed time of the algorithm in the MKL-CSR and CSR versions for each of the test matrices. For every algorithm version, the tests were carried out with the most advantageous affinity setting: lack of affinity defining for MKL-CSR and `KMP_AFFINITY=balanced` for CSR.

Basing on the achieved results it can be concluded that the suggested implementation of the CSR format makes use of Intel MIC architecture better for very sparse matrices than the CSR format from Intel MKL library. It is clear that for all matrices, even with a very small n in CC1, the runtime decreases along with the increase in the number of threads, what does not happen with the MKL–CSR version.

Figure 6 presents the speedup of RK4 in the native and offload modes on Intel Xeon Phi. This figure shows that our algorithm is scalable (as in ‘scalable parallelization’, see [29]—that is, the algorithm’s ability to scale to all the cores and/or threads) when we increase the number of threads on the Intel Xeon Phi coprocessor. Our algorithm scales well up to 120 threads. Increasing the number of threads to 180 or 240 results in

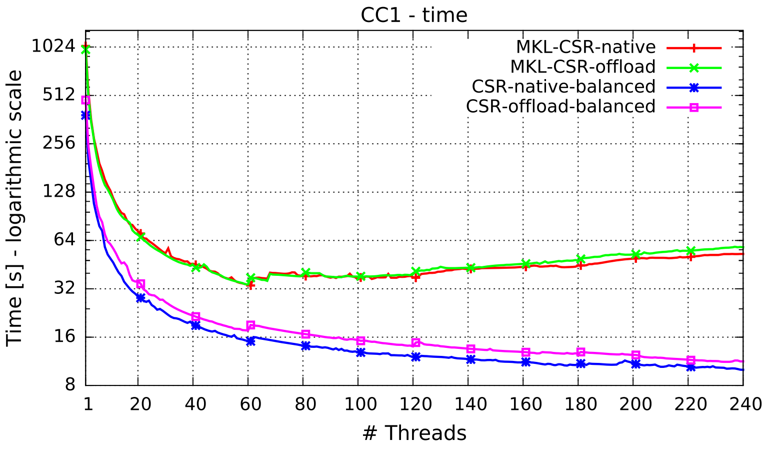

Fig. 2 Runtime of explicit fourth-order Runge–Kutta method on Intel Xeon Phi for the CC1 matrix

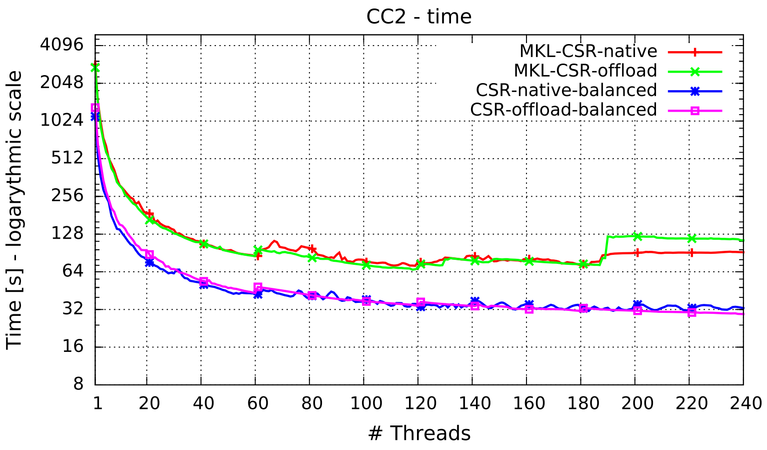

Fig. 3 Runtime of explicit fourth-order Runge–Kutta method on Intel Xeon Phi for the CC2 matrix

a modest speedup improvement due to the thread management overhead. We achieve the bigger speedup gain with the increase of the number of threads for the denser matrices (in our case they are the call-center model matrices).

The highest speedups are achieved during tests with 4 threads per core in the offload mode. The maximal speedup (of 45) is achieved for the matrix CC2 and 240 threads.

Figure 7 compares the performance of the Runge–Kutta fourth-order method using our CSR implementation for Intel Xeon Phi with Intel MKL implementation of CSR for Intel Xeon Phi with its multithreaded CPU implementation. The MKL source code is strictly the same on CPU and Intel Xeon Phi.

The application with our implementation of CSR achieves more than 3 GFlops, which gives more than double increase of the performance compared to the MKL implementation of CSR on Intel Xeon Phi. For the matrix CC1, the performance is even 4.05 GFlops, which is more the three times better than MKL-CSR.

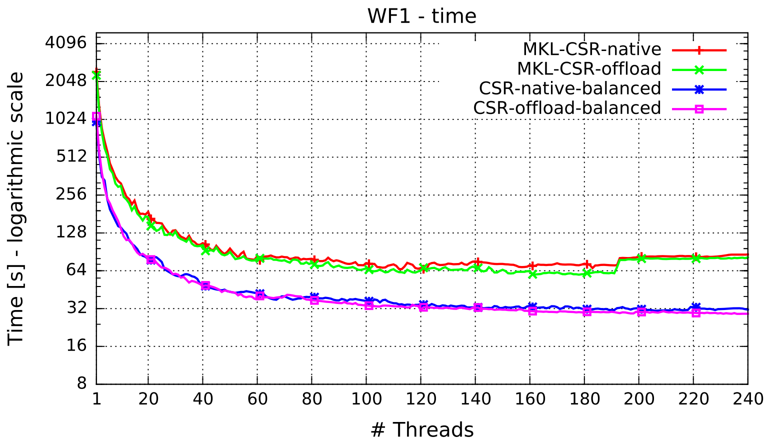

Fig. 4 Runtime of explicit fourth-order Runge–Kutta method on Intel Xeon Phi for the WF1 matrix

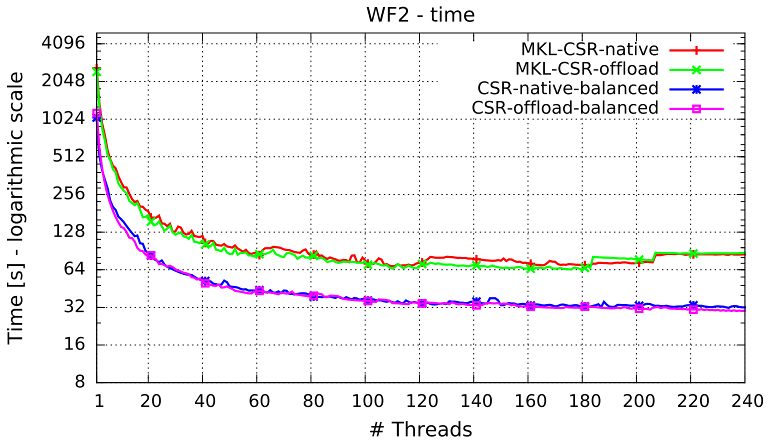

Fig. 5 Runtime of explicit fourth-order Runge–Kutta method on Intel Xeon Phi for the WF2 matrix

The Intel Xeon Phi has a peak performance of more than 1 Tflops and it is about 20 times faster than an Intel multicore CPU. In our experiments the performance is only 3–4 Gflops and a speedup of about 2. The poor performance and speedup was caused by noneffective vectorization due to sparsity, overhead due to irregular memory access in the CSR format and load-imbalance due to the non-uniform matrix structure—such problems were also indicated in [25].

We can also notice that for smaller matrices (like CC1) a bigger performance is achieved in the native mode, but for bigger sizes (CC2, WF1, WF2) the offload mode is faster.

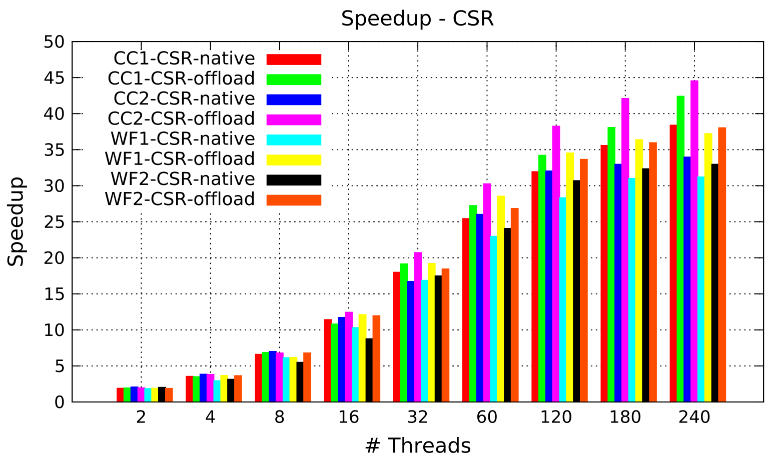

Fig. 6 Speedup of explicit fourth-order Runge–Kutta method on Intel Xeon Phi with respect to a sequential version running on one thread Intel Xeon Phi with the balanced thread affinity mode

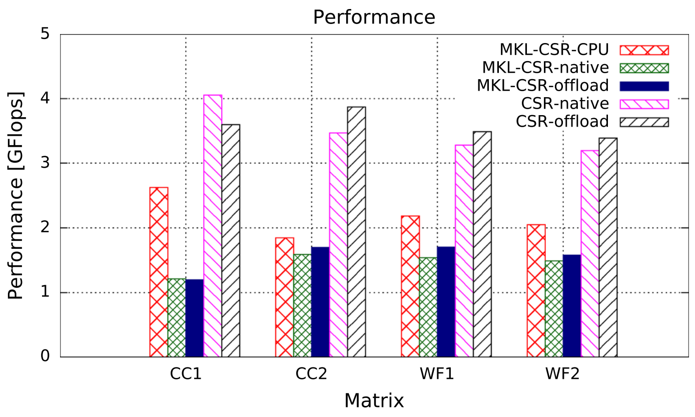

Fig. 7 The performance of the fourth-order Runge–Kutta method on Intel Xeon Phi in both modes and CPU

7 Conclusions and Future Work

In this article we have presented an approach for accelerating explicit fourth-order Runge–Kutta method using the Intel Xeon Phi. Our approach exploits the thread-level parallelism for SpMV operation and the thread-level parallelism and vectorization for the vector addition.

Based on the conducted experiments and Figs. 2, 3, 4 and 5 we can clearly state that the use of BLAS and Sparse BLAS routines available in MKL for the Intel MIC architecture performed worse than our own CSR implementation.

The MKL version of CSR achieves poor results (compared to ours) because there are a lot of barriers—that is, there is a barrier after each call of an Intel MKL function (e.g.: after `axpy`, after matrix-vector multiplication etc.). Moreover, the details of the

MKL version of the CSR format is hidden in the Intel MKL functions what makes its analysis harder.

In each implementation the difference between performances in the native and offload modes are very small (as in the work [13]). It results from the fact that sending data from the host to the coprocessor was hidden behind quite a big amount of computation and from the fact that the overhead of the offload pragma is quite low.

In the MKL version of CSR for the MIC architecture we achieve the best results with the use of 60 cores and two threads per core. In this version we can see that for 1, 2, 3 threads per core, we achieve the best results without any explicit setting the `KMP_AFFINITY` environment variable compared to any value of this variable. We can also see that for more than 3 threads per core the algorithm saturated the capabilities of the architecture—that is, the bigger number of threads do not increase the performance (and it can even decrease the performance).

Our implementation with the use of the CSR format is definitely better for every type and size of the matrix. It also gives the chance of the better usage of the new architecture—the figures imply that increasing the number of threads enables to achieve even better speedup.

In our implementation of the CSR it is worth to use 4 threads per core. We can also see here some differences dependent on the choice of the value of the `KMP_AFFINITY` environment variable. In most cases the best time is ensured by the value `balanced`. However, in the call center model (with somewhat denser matrices) we can see bigger differences for various choices of this variable's values (that is, for various thread-to-core assignments) but in the other model (with sparser matrices) the differences become blurred.

In future works the basic Intel MKL routines for the MIC architectures should be analyzed to understand their use of the Intel Xeon Phi coprocessor. Another possibility to enhance the CSR result is the use of a hybrid algorithm—small tasks would be launched on a multicore CPU and big tasks—on a MIC architecture.

References

1. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Ostruchov, S., Sorensen, D.: LAPACK User's Guide. SIAM, Philadelphia (1992)
2. Basic Linear Algebra Subprograms. http://www.netlib.org/blas/
3. Bianchi, G.: Performance analysis of the IEEE 802.11 distributed coordination function. IEEE J. Sel. Areas Commun. **18**(3), 535–547 (2000)
4. Butcher, J.C.: The Numerical Analysis of Ordinary Differential Equations: Runge–Kutta and General Linear Methods. Wiley-Interscience, New York (1987)
5. Bylina, B., Bylina, J., Karwacki, M.: Computational aspects of GPU-accelerated sparse matrix-vector multiplication for solving Markov models. Theor. Appl. Inform. **23**(2), 127–145 (2011)
6. Bylina, B., Bylina, J., Stpiczyński, P., Szałkowski, D.: Performance analysis of multicore and multinodal implementation of SpMV operation. In: Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 569–576 (2014)

7. Bylina, B., Karwacki, M., Bylina, J.: A CPU–GPU Hybrid Approach to the Uniformization Method for Solving Markovian Models: A Case Study of a Wireless Network. Springer, Berlin (2012). doi:10.1007/978-3-642-31217-5_42
8. Bylina, J., Bylina, B.: A Markovian queuing model of a WLAN node. Commun. Comput. Inform. Sci. **160**, 80–86 (2011)
9. Bylina, J., Bylina, B., Karwacki, M.: Markovian model of a network of two wireless devices. Commun. Comput. Inform. Sci. **291**, 411–420 (2012)
10. Bylina, J., Bylina, B., Karwacki, M.: An Efficient Representation on GPU for Transition Rate Matrices for Markov chains. Springer, Berlin (2014). doi:10.1007/978-3-642-55224-3_62
11. CUSP. http://cusplibrary.github.io/
12. Corporation, N.: CUDA Programming Guide. NVIDIA Corporation http://www.nvidia.com/ (2014)
13. Cramer, T., Schmidl, D., Klemm, M., an Mey, D.: OpenMP programming on Intel Xeon Phi coprocessors: An early performance comparison. In: Proceedings of the Many-core Applications Research Community Symposium at RWTH Aachen University, p. 3844 (2012)
14. Deslauriers, A., L'Ecuyer, P., Pichitlamken, J., Ingolfsson, A., Avramidis, A.N.: Markov chain models of a telephone call center with call blending. Comput. OR **34**(6), 1616–1645 (2007)
15. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.S.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Softw. **16**(1), 1–17 (1990)
16. Duff, I.S., Heroux, M.A., Pozo, R.: An overview of the sparse basic linear algebra subprograms: the new standard from the BLAS technical forum. ACM Trans. Math. Softw. **28**(2), 239–267 (2002)
17. Hairer, E., Wanner, G.: Algebraically stable and implementable Runge–Kutta methods of high order. SIAM J. Numer. Anal. **18**(6), 1098–1108 (1981). doi:10.1137/0718074
18. IEEE Standard for Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications P80211 (1997)
19. Intel: Best known methods for using OpenMP on Intel Many Integrated Core (Intel MIC) Architecture https://software.intel.com/sites/default/files/refresh_bkms_for_openmp_on_intel_mic_architecture_vol_1.pdf
20. Intel: Intel Math Kernel Library (MKL) http://software.intel.com/en-us/intel-mkl (2014)
21. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor high performance programming, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2013)
22. Karwacki, M., Bylina, B., Bylina, J.: Multi-GPU implementation of the uniformization method for solving Markov models. In: Conference on Computer Science and Information Systems (FedCSIS), 2012 Federated, pp. 533–537 (2012)
23. Klamka, J., Ogonowski, Z., Jamicki, M., Stasik, M.: Metody Numeryczne. Wydawnictwo Politechniki Iskiej, Gliwice (2004)
24. Lawson, C., Hanson, R., Kincaid, D., Krogh, F.: Basic linear algebra subprograms for fortran usage. ACM Trans. Math. Soft. **5**, 308–329 (1979)
25. Liu, X., Smelyanskiy, M., Chow, E., Dubey, P.: 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, pp. 273–282. ACM, New York. doi:10.1145/2464996.2465013 (2013)
26. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Comput Graph Forum 26(1), 80–113 http://www.blackwell-synergy.com/doi/pdf/10.1111/j.1467-8659.2007.01012.x (2007)
27. Perrone, L.F., Wieland, F.P., Liu, J., Lawson, B.G., Nicol, D.M., Fujimoto, R.M.: Variance Reduction in the Simulation of Call Centers. Proceedings of the 2006 Winter Simulation Conference (2006)
28. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes in C. The Art of Scientific Computing, 2nd edn. Cambridge University Press, New York (1992)
29. Rahman, R.: Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers, 1st edn. Apress, Berkely (2013)
30. Saule, E., Kaya, K., Çatalyürek, Ü.V.: Performance Evaluation of Sparse Matrix Multiplication kernels on Intel Xeon Phi. CoRR arxiv:1302.1078 (2013)
31. Stewart, W.J.: An Introduction to the Numerical Solution of Markov Chains. Princeton University Press, Princeton (1994)
32. Stone, J.E., Gohara, D., Shi, G.: OpenCL: A parallel programming standard for heterogeneous computing systems. Comput. Sci. Eng. **12**(3), 66–73 (2010). doi:10.1109/MCSE.2010.69
33. Whitt, W.: Engineering solution of a basic call-center model. Manag. Sci. **51**(2), 221–235 (2005)