J Supercomput (2018) 74:1497–1509
https://doi.org/10.1007/s11227-018-2235-7

Parallelization of stochastic bounds for Markov chains on multicore and manycore platforms

Jarosław Bylina[1]

Published online: 27 January 2018

Abstract The author demonstrates the methodology for parallelizing of finding stochastic bounds for Markov chains on multicore and manycore platforms. The stochastic bounds algorithm for Markov chains with the sparse matrices is investigated, thus needing a lot of irregular memory access. Its parallel implementations should scale across multiple threads and characterize with a high performance and performance portability between multicore and manycore platforms. The presented methods are built on the usage of two parallelization extensions of the C++ language: OpenMP and Cilk Plus. For this two extensions, we use two programming models, namely loop parallelism and task-based parallelism. The numerical experiments show the execution time of the implementations and the scalability on multicore and manycore platforms. This work provides the parallel implementations and at the same time presents an educational example of how computer science problems with irregular memory access can be implemented for high performance using OpenMP and Cilk Plus.

Keywords Intel Xeon Phi · Stochastic bounds of Markov chains · Sparse matrices · Intel Cilk Plus · Task-based parallelism · For-loop parallelism

1 Introduction

Modeling real complex systems with the use of Markov chains is a well-known and recognized method giving good results [20]. However, in spite of its numerous merits,

✉ Jarosław Bylina
jaroslaw.bylina@umcs.pl; jmbylina@hektor.umcs.lublin.pl

[1] Institute of Mathematics, Marie Curie-Skłodowska University, Pl. M. Curie-Skodowskiej 5, Lublin, Poland

it also has some disadvantages. One of them is the size of the model—to achieve needed accuracy we often have to create a large model (that is, a huge matrix) and such models require quite a lot of computation time [5]. There are some ways to reduce the number of investigated states (at the expense of some accuracy, of course) or to change the structure of the matrix (to make it more convenient to computations). Some of the methods [2,9–11,16] could use the Abu-Amsha–Vincent's (AAV) Algorithm [1,12]. In previous work [3], we focused on finding stochastic bounds for Markov chains with the use only of the loop parallelism from OpenMP on Intel Xeon Phi coprocessor. A known algorithm was adapted to study the potential of the MIC architecture in algorithms needing a lot of memory access and exploit it in the best way. This work is an extension of that previous work [3].

The advance of the shared memory multicore and manycore architectures caused a rapid development of one type of parallelism, namely the *thread level parallelism*. This kind of parallelism relies on splitting the program into subprograms which can be executed concurrently. Current parallel programming frameworks aid developers to a great extent in implementing applications that exploit appropriate programming model for parallel hardware resources. Nevertheless, developers require additional expertise to properly use and tune them to operate efficiently on specific parallel platforms. On the other hand, porting applications between different parallel programming models and platforms is not straightforward and demands considerable efforts and specific knowledge.

In this paper, we present parallel implementations of AVV algorithm designed for x86 based multicore manycore architecture such as Haswell CPU and Intel Knight Corner (KNC). We want to create approaches that are sufficiently general to be applied to implementations of other similar algorithms for sparse matrices which need a lot of irregular memory access. Our contribution consists of applying two C++ language extensions, namely OpenMP and the Intel Cilk Plus [8] framework for parallelization of the application. OpenMP is an API that supports multi-platform shared memory multiprocessing programming on most processor architectures and operating systems. It consists of a set of compiler directives, library routines, and environment variables Intel Cilk Plus is an extension to the C++ language to support loop parallelization and task parallelism using work-stealing policy. For these two extensions, we exploit two programming models, namely task parallelism and loop parallelism. In this work, we evaluate the suitability of OpenMP and Cilk Plus for both parallel programming models: task parallelism and loop parallelism. We want to know how the task model compares to loop parallelism model in terms of the time execution and scalability. We try to select the most suitable programming model, framework, and architecture for the algorithm for the sparse matrix to achieve the best performance.

The outline of the article is following. Section 2 gives some mathematical background of the problem and presents the original version of AAV Algorithm and shows its parallel version (for dense matrices). Section 3 discusses some aspects of the matrix representations and the parallel implementations that allow us to achieve better performance in our parallel version of the algorithm. Section 4 describes our experiments and Sect. 5 analyzes its results. Finally, Sect. 6 concludes the paper and gives some perspectives for further works.

Fig. 1 AAV Algorithm finding the upper bound—the input matrix **P** and the result matrix **V** are of the size $N \times N$

$$\begin{aligned}&\mathbf{V}_{1,*} \leftarrow \mathbf{P}_{1,*}\\ &\text{for } i = 2, \ldots, N:\\ &\quad \mathbf{V}_{i,*} \leftarrow \max_{st}\left(\mathbf{P}_{i,*}, \mathbf{V}_{i-1,*}\right)\end{aligned}$$

Fig. 2 Three steps of parallel AAV Algorithm

$$\begin{aligned}S\text{:}\quad &\text{parallel for } i = 1, \ldots, N:\\ &\quad \text{for } j = (N-1), \ldots, 1:\\ &\quad\quad \mathbf{A}_{i,j} \leftarrow \mathbf{A}_{i,j} + \mathbf{A}_{i,j+1}\\ M\text{:}\quad &\text{parallel for } j = 1, \ldots, N:\\ &\quad \text{for } i = 2, \ldots, N:\\ &\quad\quad \mathbf{A}_{i,j} \leftarrow \max(\mathbf{A}_{i,j}, \mathbf{A}_{i-1,j})\\ D\text{:}\quad &\text{parallel for } i = 1, \ldots, N:\\ &\quad \text{for } j = (N-1), \ldots, 1:\\ &\quad\quad \mathbf{A}_{i,j} \leftarrow \mathbf{A}_{i,j} - \mathbf{A}_{i,j+1}\end{aligned}$$

2 Stochastic bounds for Markov chains

Finding stochastic bounds is an idea widely used in modeling with Markov chains [7,14,17,19,21]. Some of its basics and examples can be found also in [3,7].

The main problem is to find a lower (upper) stochastic bound **V** of a probabilistic matrix **P**—that is, a matrix **V** which is less (greater) in the stochastic sense than **P**.

The crucial operation in the sequential AAV algorithm is the stochastic maximum function defined as (**p**, **q**, **r** are stochastic vectors of the size N):

$$\mathbf{r} = \max_{st}(\mathbf{p}, \mathbf{q}) \iff \forall i \in \{1, \ldots, N\} : \sum_{j=i}^{N} r_j = \max\left(\sum_{j=i}^{N} p_j, \sum_{j=i}^{N} q_j\right).$$

With the use of this function, we can obtain the upper stochastic bound in a straightforward manner—the sequential AAV algorithm—see Fig. 1 (both the matrices are stochastic ones of the size $N \times N$ and $\mathbf{M}_{k,*}$ is the kth row of a matrix).

The sequential AAV algorithm is also quite easy to parallelize when we divide it into three steps defined as the following operations—see Fig. 2 which shows these operations as in-place algorithms. Two of these routines can be parallelized row-wise (D and S) and the third one—column-wise (M) as they work on rows or on columns, respectively.

3 Parallel implementations

3.1 Representation of matrices

Because of large sizes of the probabilistic matrices, we cannot store them in a usuall, uncompressed manner. Fortunately, the Markov chains matrices are sparse matrices and there are a lot of storage schemes for such matrices [4,6]. We use a well-known

format CSR (compressed sparse rows)—because we focus on the row-wise operations (S and M). However, after the first operation, the matrix becomes dense, although regular—and for such a matrix we use a novel format, namely VCSR [3,7].

The CSR format stores only information about nonzero entries of the matrix in three arrays (values in `val`, sorted left-to-right-then-top-to-bottom by their position in the original matrix; their corresponding column indices in `col_ind`; the indices of the start of every row in `row_ptr`). The VCSR format is analogous; however, we do not store each nonzero element, but only the first ones in a constant sequence. Moreover, both the investigated operations (S and D) preserve two of the three input arrays (changing only `val`).

Thus, storing the input matrix $\mathbf{P}$ in the CSR format and the first intermediate matrix $S(\mathbf{P})$ in the VCSR format, the operation S does not change the structure nor the memory requirements of the matrix. Moreover, no auxiliary arrays are needed for this operation. The same we can say about the operation D and storing the second intermediate matrix $M(S(\mathbf{P}))$ in the VCSR format and the output matrix $D(M(S(\mathbf{P})))$ in the CSR format.

3.2 Loop-level parallelism programming model

Loop-level parallelism denotes a kind of parallel computations for multiple processors with the use of a technique for distributing the data across different parallel processor nodes which operate on some data in parallel. Parallelization of loops is strongly connected to data-based parallelism. Parallelizing loops with OpenMP is very simple with the use of `#pragma omp parallel for` directives. This lets the OpenMP scheduler choose the default mode for cutting loop iterations in chunks and distribute them on available resources. The user can set the strategy for the scheduler to specify the size of chunks that can be executed statically or dynamically. For our experiments, we use the `static` scheduler because it gave us the best results. Parallelizing loops with Intel Cilk Plus is also very simple with the use of the keyword `cilk_for` instead of the standard C++ `for` keyword giving thus a hint that parallel execution is possible in this loop. Intel Cilk Plus does not give any way to choose a scheduler.

3.3 Task-based programming model

Task-based parallelism contrasts to both loop-level parallelism and data-based parallelism as another form of parallelism. It offers the programmer other ways of expressing parallelism than in data parallelism. A task is an abstract concept which defines some work to do but does not specify ant agent or scheduling. Asynchronous tasks are a powerful programming abstraction that offers flexibility in conjunction with great expressivity. Research involving standardized tasking models like OpenMP and non-standardized models like Cilk facilitate improvements in many tasking implementations.

This paradigm will make applications generate a great number of fine-grain tasks. The success of such an approach for parallelizing applications will greatly depend on

the runtime systems ability to map the tasks to the physical threads. The execution of the new task can be instant or delayed according to the task schedule and availability of threads. The OpenMP runtime provides a dynamic scheduler of the tasks while avoiding data hazards by keeping track of dependencies. The dynamic scheduler means that the tasks are queued and executed as quickly as possible.

We employ both the standard in our task-based implementations—the OpenMP `#pragma omp task` directives and Intel Cilk Plus `cilk_spawn` keyword.

Intel Cilk Plus support task parallelism using work-stealing policy. The `cilk_spawn` keyword is used to launch a task represented by a function (or any callable object, like lambdas—as in our implementations) in parallel with the current program.

3.4 Details of implementations

Listings 1–6 show the details of parallelization techniques used in tested implementations. For clarity, there are no function headers (except the last one, where there are some additional helper functions because of the recursion); however, they all are similar and take: `N` as the number of rows (and columns) of the matrix, `NZ` as the number of nonzero elements of the matrix, and `val`, `col_ind`, `row_ptr` as the pointers to the respective arrays of the CSR/VCSR storage schemes; some of them also use `PER_TASK` which is the number of rows processed in one task (and which also determines the number of tasks).

The listings present various models of parallelization and various C++ language extensions. Namely, Listings 1 and 2 show loop-level parallelism and Listings 3–6 show the task parallelism. On the other hand, Listings 1 and 3 utilize the OpenMP standard, and Listings 2, 4, 5/6 use Cilk Plus. And finally, Listing 5/6 presents a recursive approach to generating tasks.

Listing 1 The `omp-for` version

```
// step_S

#pragma omp parallel for default(none)\
      shared(N, row_ptr, val) schedule(static)
for(int row = 0; row < N; ++row) {
  for(int ci_ind = row_ptr[row+1]-1;
      ci_ind > row_ptr[row];
      --ci_ind) {
    val[ci_ind-1] += val[ci_ind];
  }
}

// step_D

#pragma omp parallel for default(none)\
      shared(N, row_ptr, val) schedule(static)
for(int row = 0; row < N; ++row) {
  for(int ci_ind = row_ptr[row];
      ci_ind < row_ptr[row+1]-1;
      ++ci_ind) {
    val[ci_ind] -= val[ci_ind+1];
  }
}
```

Listing 2 The `cilk-for` version

```
// step_S

cilk_for(int row = 0; row < N; ++row) {
  for(int ci_ind = row_ptr[row+1]-1;
      ci_ind > row_ptr[row];
      --ci_ind) {
    val[ci_ind-1] += val[ci_ind];
  }
}

// step_D

cilk_for(int row = 0; row < N; ++row) {
  for(int ci_ind = row_ptr[row];
      ci_ind < row_ptr[row+1]-1;
      ++ci_ind) {
    val[ci_ind] -= val[ci_ind+1];
  }
}
```

Listing 3 The `omp-task` version

```
// step_S

#pragma omp parallel
{
  #pragma omp single
  {
    for(int row_ = 0; row_ < N; row_ += PER_TASK) {
      #pragma omp task
      {
        for(int row=row_;
            row < min(N, row_+PER_TASK);
            ++row)
          for(int ci_ind = row_ptr[row+1]-1;
              ci_ind > row_ptr[row];
              --ci_ind) {
            val[ci_ind-1] += val[ci_ind];
          }
      }
    }
  }
}

// step_D

#pragma omp parallel
{
  #pragma omp single
  {
    for(int row_ = 0; row_ < N; row_ += PER_TASK) {
      #pragma omp task
      {
        for(int row=row_;
            row < min(N, row_+PER_TASK);
            ++row)
          for(int ci_ind = row_ptr[row];
              ci_ind < row_ptr[row+1]-1;
              ++ci_ind) {
            val[ci_ind] -= val[ci_ind+1];
          }
      }
    }
  }
}
```

Listing 4 The `cilk-spawn` version

```
// step_S

for(int row_ = 0; row_ < N; row_ += PER_TASK) {

  cilk_spawn [&] {
    for(int row=row_;
        row < min(N, row_+PER_TASK);
        ++row)
      for(int ci_ind = row_ptr[row+1]-1;
          ci_ind > row_ptr[row];
          --ci_ind) {
        val[ci_ind-1] += val[ci_ind];
      }

  } ();
}

// step_D

for(int row_ = 0; row_ < N; row_ += PER_TASK) {

  cilk_spawn [&] {
    for(int row=row_;
        row < min(N, row_+PER_TASK);
        ++row)
      for(int ci_ind = row_ptr[row];
          ci_ind < row_ptr[row+1]-1;
          ++ci_ind) {
        val[ci_ind] -= val[ci_ind+1];
      }

  } ();
}
```

Listing 5 The `cilk-rek` version (S)

```
void aux_S(int n0, int n1) {
  if (n1 - n0 <= PER_TASK) {
    for (int row = n0; row < n1; ++row) {
      for(int ci_ind = row_ptr[row+1]-1;
          ci_ind > row_ptr[row];
          --ci_ind) {
        val[ci_ind-1] += val[ci_ind];
      }
    }
  } else {
    int n_mid = (n1+n0)/2;
    cilk_spawn aux_S(n0, n_mid);
    aux_S(n_mid, n1);
  }
}

void step_S() {
  aux_S(0, N);
}
```

Listing 6 The `cilk-spawn` version (D)

```
void aux_D(int n0, int n1) {
  if (n1 - n0 <= PER_TASK) {
    for (int row = n0; row < n1; ++row) {
      for(int ci_ind = row_ptr[row];
          ci_ind < row_ptr[row+1]-1;
          ++ci_ind) {
        val[ci_ind] -= val[ci_ind+1];
      }
    }
  } else {
    int n_mid = (n1+n0)/2;
    cilk_spawn aux_D(n0, n_mid);
    aux_D(n_mid, n1);
  }
}

void step_D() {
  aux_D(0, N);
}
```

4 Details of experiments

The whole AAV algorithm consists of the three steps (S, M, and D). In the work [3], we had shown the tests for the first step only, namely S steps on coprocessor Intel Xeon Phi. The results of the numerical experiments of the first and the third steps are presented here. We investigated the execution time and the scalability of our parallel implementations on CPU and coprocessor Intel Xeon Phi under Linux. We used two metrics to compare the computing performance: *time* and *relative speedup*. Time is the time spent in the first and the third steps of the AAV algorithm. Relative speedups are calculated by dividing the time of work with a single thread by the time with n threads.

We compared five parallel implementations of the first and the third step of the AAV algorithm (and a sequential implementation as a baseline), namely:

- the `omp-for` implementation using `#pragma omp parallel for` from OpenMP standard based on the fork-join paradigm with static scheduler (Listing 1);
- the `cilk-for` implementation using the `cilk_for` keyword from Intel Cilk Plus (Listing 2);
- the `omp-task` implementation using `#pragma omp task` from OpenMP standard based on the task paradigm (Listing 3);
- the `cilk-spawn` implementation using the `cilk_spawn` keyword from Intel Cilk Plus (Listing 4);
- the `cilk-req` implementation using the `cilk_spawn` keyword with the recursive manner (Listing 5/6).

Table 1 shows details of the specification of the hardware and software used in the numerical experiment.

The tested implementations were written in the language C++ language (with the use of the OpenMP pragmas and Intel Cilk Plus framework for parallelization) and compiled with Intel C++ Compiler (`icc`) with optimization flag `-O3`. All the results are presented for the `double` data type and the *native* mode for MIC. Carrying out the numerical experiments, we were changing the number of available threads. For CPU, we studied the number of the threads from 1 to 24 (the number of the physical cores). For MIC in our tests, we used 60 cores in native mode. In case of native execution model, when the application is started directly on Xeon Phi card, we can use 60 cores (with all 4 threads on each). For MIC, the numbers of threads studied were up to 120—bigger numbers of threads degraded the performance.

The tests were performed with the use of random matrices. Their size was 40000×40000 with different numbers of nonzero elements on CPU and MIC. There were also some other tests (not presented here) for other sizes and densities of the matrix; however, their results were very similar to the results shown below—the number of the nonzero elements was crucial.

A single run of our operations was quite short, so to measure the performance time accurately we run every test repeatedly and average the time. Every run covered every action needed in the algorithm from the very beginning—initialization, allocation, etc., to the very end.

Table 1 Hardware and software used in the experiments

	CPU	MIC
	2 × Intel Xeon E5-2670 v.3 (Haswell)	Intel Xeon Phi 7120 (Knights Corner)
# Cores	24 (12 per socket)	61
# Threads	48 (2 per core)	244 (4 per core)
Clock	2.30 GHz	1.24 Ghz
Level 1 instruction cache	32 kB per core	32 kB per core
Level 1 data cache	32 kB per core	32 kB per core
Level 2 cache	256 kB per core	512 kB per core
Level 3 cache	30 MB	–
RAM memory	128 GB	16 GB
Compiler	Intel ICC 16.0.0	Intel ICC 16.0.0
BLAS/LAPACK libraries	MKL 2016.0.109	MKL 2016.0.109
Max. mem. bandwidth	68 GB/s	352 GB/s
SIMD register size	256 b	512 b

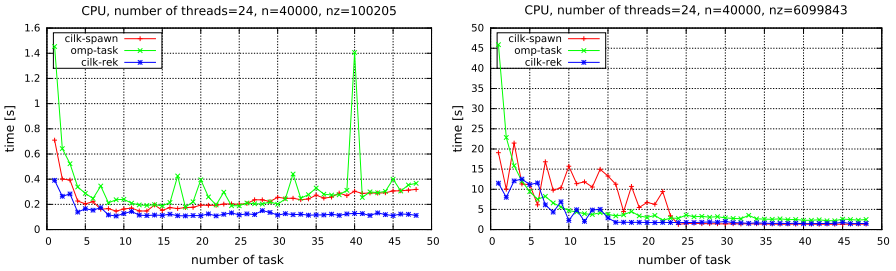

Fig. 3 Execution time for various number of tasks on CPU (left: 100 205 nonzeros; right: 6 099 843 nonzeros)

5 Experimental results

All the processing times are reported in seconds. The time is measured with an OpenMP function `omp_get_wtime()`.

5.1 Time

Figures 3 and 4 present the execution time of our tests for different numbers of tasks for two matrices with the same matrix size and a different number of nonzeros elements on CPU and MIC, respectively. Left, we have results for a very sparse matrix (100 205 nonzeros; that is about 2.5 elements per row on average). Right, we have results for a denser matrix (6 099 843; about 152 elements per row on average).

After these tests, we chose 24 tasks and 24 threads for further investigations on CPU. Similarly, we chose 120 tasks and 120 threads for further investigations on MIC.

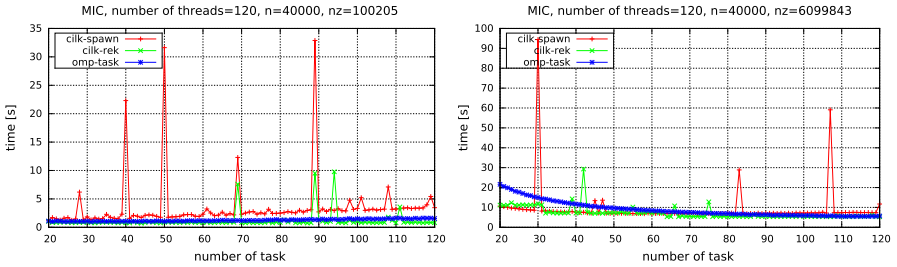

Fig. 4 Execution time for various number of tasks on MIC (left: 100 205 nonzeros; right: 6 099 843 nonzeros)

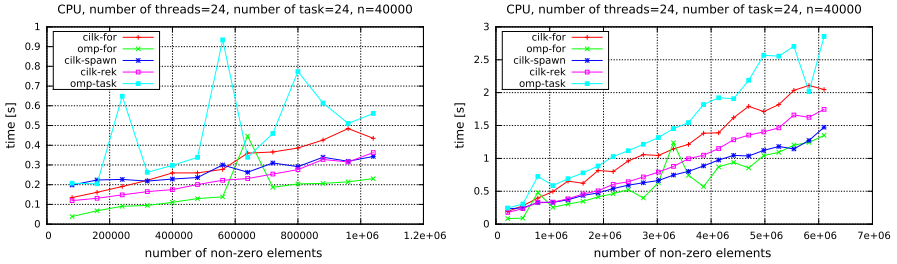

Fig. 5 Execution time for various matrices on CPU (left: up to about 1 000 000 nonzeros; right: up to about 6 000 000 nonzeros)

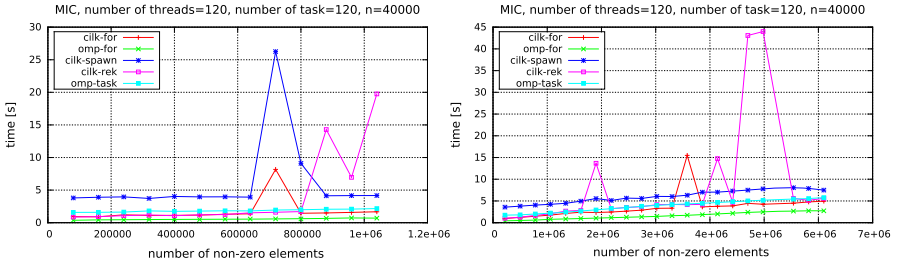

Fig. 6 Execution time for various matrices on MIC (left: up to about 1 000 000 nonzeros; right: up to about 6 000 000 nonzeros)

Figures 5 and 6 show the execution time of our tests for different numbers of nonzeros for the constant size of the matrix (40000 × 40000) on CPU and MIC, respectively.

Numerical experiments show that:

- the `omp-for` version has the shortest execution time—both on CPU and on MIC; however, it acts strange because it jumps on CPU;
- the `omp-task` version has the longest execution time on CPU;
- the `cilk-spawn` version has the longest execution time on MIC;
- the `omp-for` version is better than `cilk-for`—that is, loop-level parallelism is better in OpenMP;

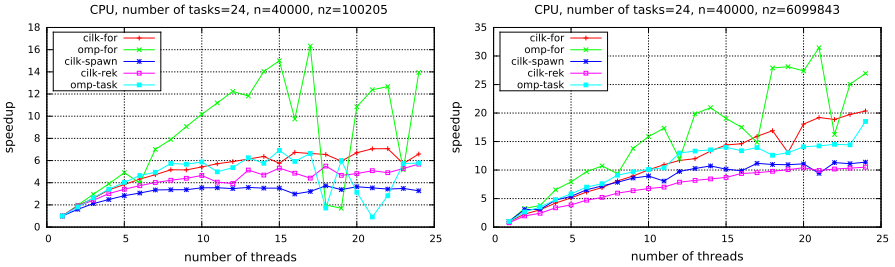

Fig. 7 Speedup for various number of threads on CPU (left: 100 205 nonzeros; right: 6 099 843 nonzeros)

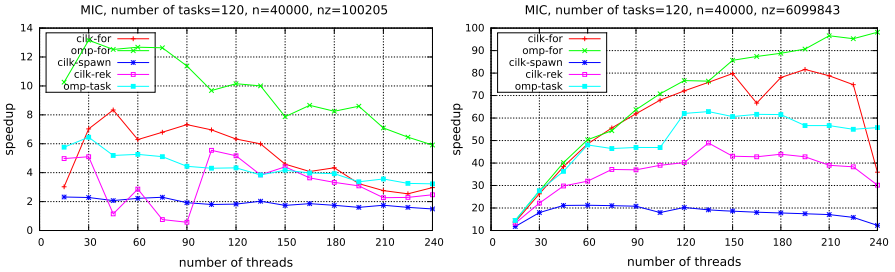

Fig. 8 Speedup for various number of threads on MIC (left: 100 205 nonzeros; right: 6 099 843 nonzeros)

– on the other hand, task-based parallelism is not implemented very efficiently in OpenMP—in comparison with Intel Cilk Plus;
– CPU runs faster than MIC.

5.2 Speedup

Figures 7 and 8 show the speedup on CPU and MIC, respectively—also for various matrices of the same size (40000×40000).

For a small number of nonzeros, all the implementations are poorly scalable—both on CPU and on MIC. For a bigger number of nonzeros, the loop-level parallelism gives the best scalability. However, `cilk-for` is more stable on CPU than `omp-for` which jumps. On MIC, the situation is opposite (`omp-for` is more stable). The worst scalability is achieved by `cilk-spawn` and `cilk-rek` both on CPU and MIC. It seems that the overhead grows with the growth of the number of threads because the scalability is better for fewer threads.

The poor speedup was caused by noneffective vectorization due to sparsity, the overhead due to irregular memory access in the VCSR format, and load-imbalance due to the nonuniform matrix structure—such problems were also indicated in [15].

6 Conclusion and future works

In this article, we have presented an approach for parallelization of stochastic bounds for Markov chains on CPU and Intel Xeon Phi. The parallelization of the matrix

operations, where matrices are sparse and there is a lot of irregular memory access, is the difficult issue. We have shown the abilities of two parallel extensions of C++ language, namely OpenMP and Cilk Plus in the terms of the loop parallelism and task parallelization. Based on the conducted experiments, we can clearly state that:

- the better results were obtained for CPU than Intel Xeon Phi;
- the use of `#pragma omp parallel for` from the OpenMP standard performed better than the `cilk_for` constructs from Cilk Plus;
- the use of `#pragma omp task` from the OpenMP standard performed worse than the `cilk_spawn` constructs from Cilk Plus;
- the best results were obtained for `#pragma omp parallel for` from the OpenMP standard;
- the poorest results were achieved for `#pragma omp task` from the OpenMP standard.

This paper presents the strength of the OpenMP standard for parallelizing with the use of `#pragma omp parallel for`. While the Intel Cilk Plus standard offers similar functionality, in our applications we were not able to achieve satisfactory performance results with Cilk Plus despite investing a greater development effort than we did with OpenMP. The results depend on the number of nonzeros elements in sparse matrices for Intel Cilk Plus.

MIC (KNC) performed worse than CPU, so we should focus on the ways to exploit the performance potential of not only many cores but also wide vector units in each core and data locality [13]. We are also going to utilize other extensions like OmpSs [18] and Intel Threading Building Blocks (TBB) [22] and then to compare new results with the current ones.

References

1. Abu-Amsha O, Vincent J.-M (1998) An algorithm to bound functionals on Markov chains with large state space. In: 4th INFORMS Conference on Telecommunications, Boca Raton, Floride, E.U, INFORMS
2. Busic A, Fourneau J.-M (2005) A matrix pattern compliant strong stochastic bound. In: 2005 IEEE/IPSJ international symposium on applications and the internet workshops (SAINT 2005 Workshops), Trento, Italy, pages 260263. IEEE Computer Society
3. Bylina J (2018) Stochastic bounds for Markov chains on Intel Xeon Phi coprocessor. PPAM 2017. LNCS, Springer [in print]
4. Bylina B, Bylina J, Karwacki M (2011) Computational aspects of GPU-accelerated sparse matrix-vector multiplication for solving Markov models. Theor Appl Inform 23(2):127–145, ISSN 1896-5334
5. Bylina J, Bylina B, Karwacki M (2012) A Markovian model of a network of two wireless devices. In: Proceedings of computer networks 2012, pp. 411–420. https://doi.org/10.1007/978-3-642-31217-5_43
6. Bylina J, Bylina B, Karwacki M (2014) An efficient representation on GPU for transition rate matrices for Markov chains. PPAM 2013, Part I, Lecture Notes in Computer Science 8384, pp. 663–672. https://doi.org/10.1007/978-3-642-55224-3_62

7. Bylina J, Fourneau J.-M, Karwacki M, Pekergin N, Quessette F (2015) Stochastic bounds for Markov chains with the use of GPU. In: Proceedings of computer networks 2015, CN 2015, CCIS 522, pp. 357–370. https://doi.org/10.1007/978-3-319-19419-6_34
8. Cilk Plus. https://software.intel.com/en-us/intel-cilk-plus
9. Dayar T, Pekergin N, Youns S (2006) Conditional steady-state bounds for a subset of states in Markov chains. In: Structured Markov chain (SMCTools) workshop in the 1st International Conference on Performance Evaluation Methodolgies and Tools, VALUETOOLS 2006, Pisa, Italy. ACM
10. Fourneau J-M, Le Coz M, Quessette F (2004) Algorithms for an irreducible and lumpable strong stochastic bound. Linear Algebra Appl 386:167185
11. Fourneau J.-M, Le Coz M, Pekergin N, Quessette F (2003) An open tool to compute stochastic bounds on steady-state distributions and rewards. In: 11th international work-shop on modeling, analysis, and simulation of computer and telecommunication systems (MASCOTS 2003), Orlando, FL, IEEE Computer Society
12. Fourneau J.-M, Pekergin N (2002) An algorithmic approach to stochastic bounds. In: Performance evaluation of complex systems: techniques and tools, performance 2002, tutorial lectures, volume 2459 of LNCS, pages 6488. Springer
13. Jeffers J, Reinders J (2013) Intel Xeon Phi coprocessor high performance programming, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco
14. Kijima M (1997) Markov processes for stochastic modeling. Chapman & Hall, London
15. Liu X, Smelyanskiy M, Chow E, Dubey P (2013) Efficient sparse matrix-vector multiplication on x86-based many-core processors. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, pp 273–282
16. Mamoun M Ben, Busic A, Pekergin N, Pekergin N (2007) Generalized class C Markov chains and computation of closed-form bounding distributions. Probab Eng Inf Sci 21:235260
17. Muller A, Stoyan D (2002) Comparison methods for stochastic models and risks. Wiley, New York
18. OmpSs Specification: Barcelona Supercomputing Center https://pm.bsc.es/ompss-docs/spec/
19. Shaked M, Shantikumar JG (1994) Stochastic orders and their applications. Academic Press, San Diego
20. Stewart WJ (1995) Introduction to the numerical solution of Markov chains. Princeton University Press, New Jersey
21. Stoyan D (1983) Comparison methods for queues and other stochastic models. Wiley, Berlin
22. Threading Building Blocks (Intel TBB). https://www.threadingbuildingblocks.org/