Performance Evaluation of Probabilistic Time-Dependent Travel Time Computation

Martin Golasowski, Radek Tomis(✉), Jan Martinovič, Kateřina Slaninová, and Lukáš Rapant

IT4Innovations National Supercomputing Centre,
VŠB - Technical University of Ostrava, 17. listopadu 15/2172,
708 33 Ostrava, Czech Republic
{martin.golasowski,radek.tomis,jan.martinovic,
katerina.slaninova,lukas.rapant}@vsb.cz

Abstract. Computational performance of route planning algorithms has become increasingly important in recent real navigation applications with many simultaneous route requests. Navigation applications should recommend routes as quickly as possible and preferably with some added value. This paper presents a performance evaluation of the main part of probabilistic time-dependent route planning algorithm. The main part of the algorithm computes the full probability distribution of travel time on routes with Monte Carlo simulation. Experiments show the performance of the algorithm and suggest real possibilities of use in modern navigation applications.

Keywords: Probabilistic time-dependent route planning · Monte Carlo simulation · Probabilistic speed profiles · Xeon Phi · HPC

1 Introduction

Nowadays, there are many navigation services that help to find the best route to a requested destination. Drivers usually prefer the fastest route to the destination, which naturally depends on a departure time. Navigation services can find a variety of routes depending on many factors. Such factors can be map data, an algorithm used for the route planning, a possibility to use traffic data, etc. Recently, the traffic data are used extensively, because more precise results can be produced with their help. Many navigation services use traffic data to produce better and more driver friendly routes. However, such navigation services usually use only averages [6] of historic travel times on roads for computation of the routes, even though this is not sufficient in many cases. Traffic events can significantly affect the actual travel time of a route [1,2]. There can be recurrent traffic congestions at specific times which causes delays in traffic. Navigation services then should calculate with such events [3]. The probability of the traffic congestion can be low, but the delay in the case the congestion happens can be very long. Therefore, it is advantageous to count with uncertain traffic events and their probabilities in the computation of the route [4,9].

K. Saeed and W. Homenda (Eds.): CISIM 2016, LNCS 9842, pp. 377–388, 2016.
DOI: 10.1007/978-3-319-45378-1_34

Our main motivation for this paper is to give drivers some added value for suggested routes. This added value is a complete probability distribution of travel time on a suggested route. The probability distribution of travel time on a route is a valuable information, because it indicates how likely it is for the driver to reach the destination according to travel time. We use our previously developed algorithm [8] based on Monte Carlo simulation for the computation of probability distributions of travel time. The algorithm is briefly described at Sect. 2.

The added value in the form of the probability distribution of travel time brings increased computational complexity. As the first step, we suggest the best route like other navigation services, and as the second step we compute the probability distribution of travel time on this route, which requires additional computational time. The probability distribution of travel time on single route changes depending on a departure time and also on a day of the week. Traffic congestions are often connected to rush hours and some days of the week can have worse congestions than others, which is reflected in the probability distribution.

Many users might request a route at the same time and navigation services should be able to handle as many route requests at once as possible. There are two main ways to deal with a lot of route requests. The first one is to dynamically handle requests at the time they arrive and the second one is to perform the preprocessing of the probability distributions on all routes for all possible departure times and then handle requests from this precomputed data. However, it is almost impossible to cover all departure times in preprocessing, so some discretization of departure times have to be introduced. The computation time of such preprocessing is huge and depends on the number of departure times.

The main goal of this paper is to evaluate the limitations of our algorithm in a scenario with large amounts of simultaneous route requests in conjunction with a specific hardware. In other words, it would be beneficial to know how many route requests can be handled at once with different hardware. We should note that this paper evaluates only our previously developed algorithm [8] for the computation of the probability distribution of travel time, because the regular route planning used for the selection of the route is an ubiquitous topic. The organisation of the paper is as follows. Section 2 describes the tested algorithm, Sect. 3 presents experiments and an architecture overview of a testing hardware. Finally, Sect. 4 concludes the paper with final remarks.

2 Algorithm Overview

This section presents a brief description of the algorithm [8] used for the computation of the probability distribution of travel time. The input for the algorithm is some selected route and a departure time. The route is composed as a line of road segments. A Monte Carlo simulation is used for the computation of probability distributions. The simulation randomly selects probabilistic speed profiles on road segments and computes travel time at the end of the route. Probabilistic speed profiles are time-dependent sets of speeds and their probabilities assigned to road segments. Many Monte Carlo simulation iterations are needed to obtain enough

travel times for the construction of the probability distribution of travel time. The number of simulation iterations greatly affects the precision of the result.

Probabilistic speed profiles present a way to state that various levels of traffic can occur with some probability. Different profile types can exist for variety of situations, e.g. free flow, congestions, traffic jams. All individual profiles are determined by belonging road segment and the time interval of its validity. Probabilistic speed profiles can be created in many ways. One way is to use Level-of-Service (LoS), which is the measured speed divided by the free flow speed. We use this method and describe it in detail at Sect. 3.

Algorithm 1 shows a pseudo code of Monte Carlo simulation. The simulation is repeated for a specified number of samples, and then the probability distribution is created from computed travel times. Time t is an actual travel time at the end of a road segment. Notice that the computation of t for a particular road segment depends on t from its preceding road segment. This makes single simulation computationally dependent and hard to parallelize. However, individual simulations are very fast and strictly independent of each other and can be run in parallel.

Algorithm 1. Monte Carlo Simulation for the Computation of the Probability Distribution of Travel Time for Given Array of Road Segments, Departure Time and Number of Samples

```
1: procedure MonteCarloSimulation(segments, t_d, samples)
2:     for all samples do
3:         t = t_d                                  ▷ set time t to departure time t_d
4:         for s in segments do
5:             r = random number between 0 and 1
6:             t = ComputeTravelTime(s, t, r)       ▷ speed profile is selected by r
7:         end for
8:         add travel time t to result
9:     end for
10: end procedure
```

3 Experiment

Our algorithm is based on the well-known Monte Carlo method which is often regarded as embarrassingly parallel algorithm. The computation involves running a large number of independent instances of the algorithm with different data which makes it perfectly suitable for massively parallel computation platform such as Intel Xeon Phi. However, various problems can arise since sequential part of our implementation is memory/branch bound and is unable to fully exploit the power of vector processing unit available on the coprocessor. We expect good scalability on the coprocessor as well as on the host CPU even with this potential handicap. In this section we present results measured on the coprocessor and optimizations applied to our code to gain better speedup.

We do not try to establish a benchmark of the coprocessor since our code is not a representative sample of all Monte Carlo implementations. However, every optimization applied can improve run time on the coprocessor as well as on the host CPU. Therefore, at the end of this section we present a comparison of run times on both platforms.

3.1 Intel Xeon Phi Coprocessor Architecture Overview

The Intel Xeon Phi is a SIMD oriented coprocessor based on the many integrated core (MIC) architecture. The first version known as Knights Ferry was introduced in 2010 as a proof-of-concept targeted at high performance computing segment to counter-balance increased demand for GPGPU based architectures. The Phi has 61 x86-based cores, each core capable of running 4 threads at a time [5]. The cores are based on heavily modified P54c architecture extended by new vector processing unit (VPU) and larger cache memory. The cores communicate with the memory via ring interconnect bus based on the uncore topology introduced by Intel with the Sandy Bridge microarchitecture. It has 8 distributed memory controllers communicating with graphics double data rate (GDDR5) RAM memory. The connection with host processor is realized via 16x PCI express bus.

The main power of the Xeon Phi is in the new VPU which supports instructions from the AVX-512 instruction set. Using these instructions, each core of the Xeon Phi is able to perform arithmetical operations on either 16 single precision or 8 double precision floating point numbers at once. A theoretical speed up of 16x is achievable by correct usage of the Xeon Phi VPU. However, correct usage of the VPU depends on various factors, such as data access pattern, memory alignment, and size of the data set (ideally, the entire working data set should fit in the on-board memory, since PCIe bus introduces a significant bottleneck).

Thanks to its roots in x86 architecture, the programming model for the Phi is more or less the same as for the regular CPUs. As the card itself runs a common Linux-based operating system, standard threading models such as OpenMP, Intel TBB, or pthreads can be used, as well as MPI. Intel supplies its own SDK with optimized version of the math kernel library (MKL) and C/C++ and Fortran compilers. However, to achieve optimal exploitation of the Xeon Phi computational power, a substantial amount of attention should be paid to effective partitioning of the workload and to the overall differences of the architecture by profiling and optimization of the code.

Although cores of the Xeon Phi have superscalar properties, the cores itself are relatively slow ($1.2\,\text{GHz} < 2.5\,\text{GHz}$) compared to the Haswell CPUs and lack several features such as level 3 cache or out of order execution. Due to this deficiency, the Phi cores may perform worse in sequential or branch-heavy code compared to the host CPU (such as Haswell). On the other hand, well vectorized code can run significantly faster on the Phi than on the host CPU, thanks to the VPU and significantly larger ($12 < 61$) number of available cores.

3.2 Traffic Data

The data for our experiment come from motorways and 'A' roads managed by the Highways Agency, known as the Strategic Road Network, in England[1,2]. This data contain time series of average journey time, speed and traffic flow

[1] https://data.gov.uk/dataset/dft-eng-srn-routes-journey-times.
[2] http://www.nationalarchives.gov.uk/doc/open-government-licence/version/3/.

information for segments of various length (from few hundred meters to several kilometers) with 15-minute aggregation periods (i.e. four values for each hour). Traffic speed in these data sets is calculated using a combination of stationary sensors (Automatic Number Plate Recognition, Induction loops) and floating car data sources.

In our experiment, we used data from January to December 2014. Route from Sunderland to Thetford was chosen, because it is one of the longest direct routes contained in the data set. This data were used to calculate probabilistic speed profiles for the segments along the route and to calculate empirical free flow speed for these segments. These free flow speeds were calculated as average of top 10 % aggregated speed measurements on the chosen segment (as these speeds are most probably not influenced by the traffic density).

Profiles used for experiments were created for four LoS intervals, specifically $\langle 100; 75)$, $\langle 75; 50)$, $\langle 50; 25)$ and $\langle 25; 0\rangle$. An example of such speed profiles for the entire route from Sunderland to Thetford, Monday 8:00 is presented in Fig. 1.

The interpretation of what these intervals mean can be as follows. The first interval $\langle 100; 75)$ represents free traffic, the second interval $\langle 75; 50)$ corresponds to light traffic, the third interval $\langle 50; 25)$ means heavy traffic, and the last one $\langle 25; 0\rangle$ is for traffic jams.

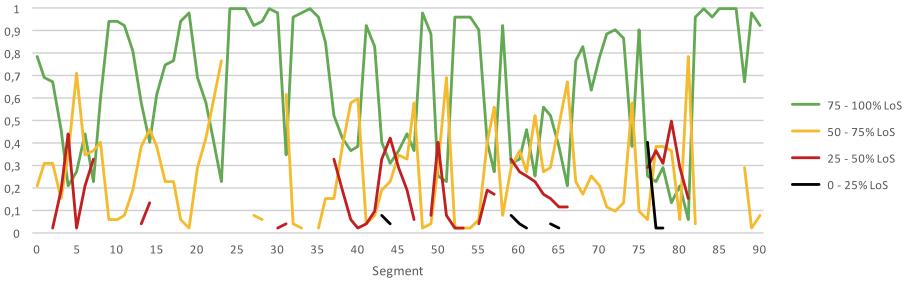


Fig. 1. Speed profiles for the selected route, monday 8:00 AM

Table 1. Example of probabilistic speed profiles

Day of week	Hour	Level of service							
		100 %		75 %		50 %		25 %	
		s [km/h]	p [%]	s [km/h]	p [%]	s [km/h]	p [%]	s [km/h]	p [%]
Monday	8:00	83.45	0.21	61.33	0.058	39.46	0.29	14.39	0.44
	8:15	83.40	0.19	56.25	0.086	36.18	0.21	15.44	0.52
	8:30	85.71	0.13	66.33	0.057	35.29	0.32	14.12	0.48
	8:45	84.00	0.17	63.08	0.230	40.23	0.25	17.05	0.34

Figure 2 shows an example of speed profiles of the segments on the whole selected route for the interval 50–75 % of LoS. The color indicates the probability that this interval of LoS occurrs. The speed profiles were created on the basis of data from January to December 2014 considering days of the week. Table 1 shows example of probabilistic speed profiles for single road segment and for one hour. Each LoS is represented by two values, speed s in kilometers per hour and probability $p \in \langle 0; 1 \rangle$.

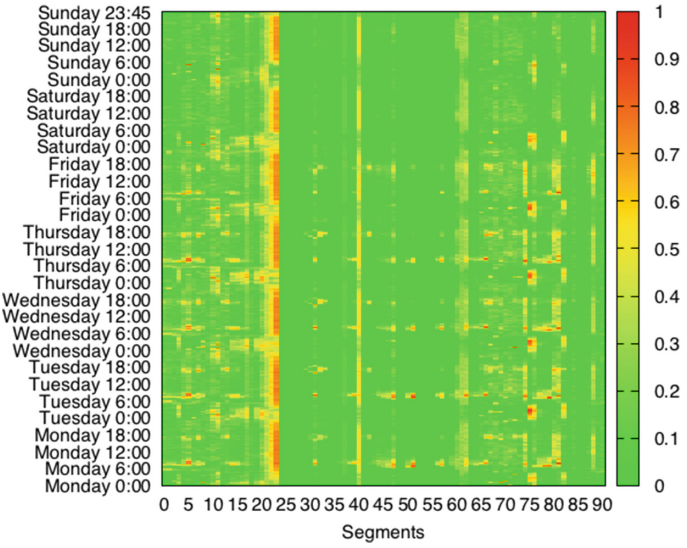
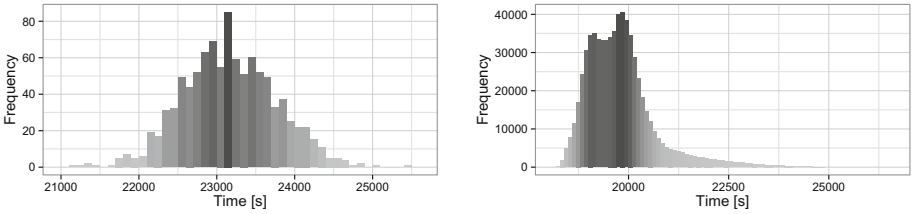


Fig. 2. Heatmap of speed profiles for 50–75 % of LoS

3.3 Experimental Simulations

The simulation was performed with the traffic data described in Subsect. 3.2 on the Salomon supercomputer operated by IT4Innovations National Supercomputing Centre, the Czech Republic. Nodes of the Salomon cluster contain a version of the Xeon Phi coprocessor based on the Knights Corner microarchitecture (7210P). This model has 16 GB of on board RAM memory, 61 compute cores, and there are 2 cards per compute node of the cluster. In total, there are 432 nodes containing 2 Xeon Phi coprocessors. Each compute node has two 12 core Haswell CPUs and 128 GB RAM.

An example of simulation output is presented in Figs. 3a and 3b. The single profile simulation (a) uses a selected profile as a starting point, then runs the simulation for the entire path. In this case, the starting point was determined by week day (Tuesday) and time (8:00 AM). The all profiles simulation (b) runs the algorithm for all available speed profiles determined by its parameters (i.e. days of the week and 15 min intervals of a single day). Both the simulations have been run with 1000 Monte Carlo samples.



(a) Single Profile (b) All Profiles

Fig. 3. Example of the simulation output

The first simulation provides information about the most probable travel time along the given path and for a selected time of departure. Based on results of our simulation shown in Fig. 3a, it can be said that if you drive from Sunderland to Thetford on Tuesday around 8:00 AM, you will most probably reach your destination in about 6 h and 30 min (23,200 s). By looking at results of the simulation for all profiles in Fig. 3b, it is evident that the estimated travel time can vary through the week, however the journey will most likely take no more than 5 h and 30 min (19,800 s).

3.4 Performance on the Intel Xeon Phi Coprocessor

Various other factors involving configuration of the run time environment can affect performance of the coprocessor. One of the factors in our case was thread allocation strategy. Our code is parallelized using the OpenMP standard. Intel provides its own OpenMP implementation as part of the Xeon Phi Manycore Software Stack (MPSS) toolchain. The library allows to choose between various thread allocation strategies by setting `KMP_AFFINITY` environment variable[3].

There are three main strategies. The *compact* strategy allocates whole cores in blocks. For example, running 40 threads on the Phi would use only 10 cores since single core runs 4 hardware threads. The *scatter* strategy assigns threads to cores evenly in round-robin fashion across all cores. The *explicit* strategy is the last one and allows explicit assignment of a given software thread to a particular core.

Measurements of the code scalability with the *compact* and *scatter* strategy are presented in Fig. 4. The *scatter* strategy is more suitable in our case since our code is memory/branch bound and running more than 2 threads per core creates unwanted congestion during memory reads.

Another factor involved is the cache hierarchy of the coprocessor. The Xeon Phi has 32 kB + 32 kB (instruction, data) L1 cache and 512 kB L2 data cache per core. The graph in Fig. 5 shows differences in performance of the code on cold and hot caches. The cold cache times were measured from single execution

[3] Thread affinity interface documentation: https://software.intel.com/en-us/node/522691.

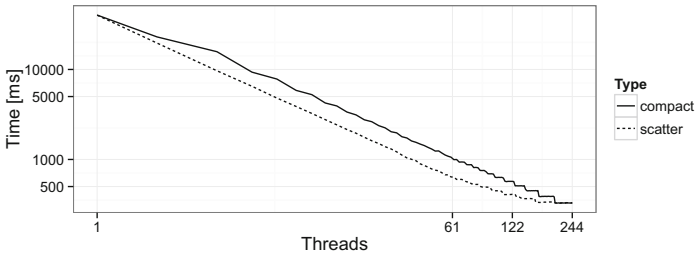


Fig. 4. Scalability of simulation for all available profiles on the Xeon Phi coprocessor

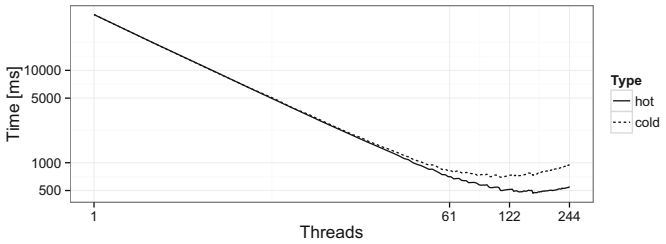


Fig. 5. Scalability of simulation for all available profiles on the Xeon Phi coprocessor

of the code on the coprocessor for different number of threads. The hot cache times were measured as average of 10 successive runs of the code.

The graphs show that proper use of the cache memory can provide significant speed up of our code. Unfortunately, the input data for single run will most likely differ each time, since most of the user requests will be computed for different road segments and start time. The real world usage of the coprocessor is reflected by the cold cache measurement which means we are able to effectively use only half of the resources available for efficient usage of the coprocessor.

Scalability of the code on the host CPU is presented in Fig. 6. The code scales very well while execution time is approx. 2 times shorter when running on single 12 core Haswell CPU that on the coprocessor. The code running on the CPU benefits from L3 data cache as well as from out-of-order execution and more sophisticated memory prefetching.

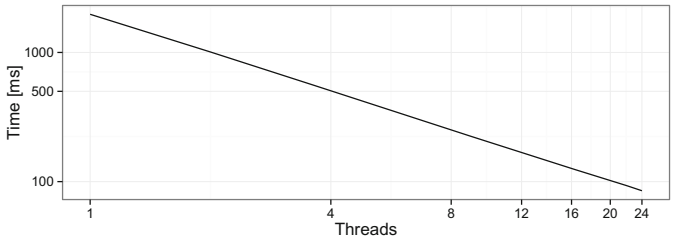


Fig. 6. Scalability of simulation for all available profiles on the host CPU

Table 2. Measured run times on the coprocessor

Threads	Times [ms]			
	Scatter	Compact	Cold	Hot
1	40,111	40,207	40,162	40,118
61	637	995	819	705
122	410	571	731	514
244	332	331	955	553

Table 3. Measured run times on the host CPU

Threads	Times [ms]
1	1,973
12 (single CPU)	166
24 (two CPUs)	85

Measured times of execution on the coprocessor are presented in the Table 2 and measured times on the host CPUs in Table 3. All measurements were performed with 1,000 samples run, threads were distributed evenly across all available cores of the target platform (except for the compact thread affinity strategy).

3.5 Code Optimization

In parallel Monte Carlo based simulations, the optimization effort usually focuses on finding a suitable method for random number generation and on optimizing data access pattern since only single synchronization occurs at the end of all runs and no data dependencies exist between the threads. The core of our algorithm relies on the selection of a speed profile given a canonical probability $p \in (0; 1)$. Effective indexing of the speed profiles for a given path segment can significantly improve overall run time of the algorithm. The speed profile for single time interval (identified by a day of the week and a time) is represented by pairs of velocity and probability values (see Table 1). The velocity values are distributed in an array of fixed length according to their probability.

The size of the array determines the resolution of the probability values which can be represented by a linear index. An example of the mapping between speed profiles and the storage array is presented in Table 4. The first table shows a single probabilistic speed profile with four LoS. The second table shows the final distribution and encoding of the velocities to the linear array indexes according to their probability. All the speed profiles for single road segment are represented by matrix of the velocity arrays stored in single linear array to improve usage of the cache memory. This type of the representation lowered the amount of necessary processor instructions used for the selection of the appropriate speed from the speed profile.

Table 4. Example of probabilistic speed profile encoding

Speed				83.45	61.33	39.46	14.39			
Probability				0.21	0.058	0.29	0.442			
Index	0	1	2	3	4	5	6	7	8	9
Speed	83.45	83.45	61.33	39.46	39.46	39.46	14.39	14.39	14.39	14.39

The early version of our code computed the index of the profile in a given day from time specified in UNIX timestamp format. The computation of the profile index involved several system function calls (namely `localtime`) which posed a significant bottleneck in the multithreaded environment on the coprocessor. The next version was based on simple data types and did not use any date-time related functions from the C standard library.

However, profiling in Intel VTune[4] suggested that another bottleneck appeared as we used pseudo-random number generation routines (introduced in C++11) available in C++ Standard Library. The slowdown was apparent especially when using large number of threads such as on Xeon Phi accelerator, since the library implementation was mainly sequential and thus unsuitable for usage on the Xeon Phi due to lack of any parallelism and vectorization.

Intel provides optimized version of the math kernel library (MKL)[5] for the Xeon Phi in its SDK[6] and we have decided to use their Mersenne-Twister implementation. This implementation uses vector instructions and allows us to generate the random numbers in batches which leads to more efficient use of the coprocessor. In the current version of our code, the random number generation takes only approx. 20 % of the computation time according to the Intel VTune profiler in comparison with 90 % in the previous version of the code.

4 Conclusion

The paper evaluated the computational performance of our algorithm for the computation of probabilistic time-dependent travel time on the Intel Xeon Phi coprocessor and Haswell CPU. Various code optimizations described in Sect. 3.5 have been done with appropriate speed up. We expected good scalability on the Xeon Phi coprocessor due to large parallelization potential of the algorithm even though it was not taking any advantage of the vector processing unit, and seemed generally unsuitable for the coprocessor. However, we have been able to use the VPU by using Intel MKL Vector Statistics Library. Finally, the good scalability was achieved on the coprocessor as well as on the host CPU. Measured

[4] https://software.intel.com/en-us/articles/how-to-analyze-xeon-phi-coprocessor-applications-using-intel-vtune-amplifier-xe-2015.

[5] https://software.intel.com/en-us/mkl-reference-manual-for-c.

[6] https://software.intel.com/en-us/articles/intel-manycore-platform-software-stack-mpss.

execution times and scalability graphs are presented in Sect. 3.3. By performing experiments, we estimated that it is the best to run our code on 122 threads evenly distributed across all 61 cores of the coprocessor.

The experiments met our expectations, however a number of specific optimizations had to be applied to obtain these results. Each optimization that led to the improvement for the coprocessor also improved the execution time on the CPU, often significantly. Finally, the execution time was at best 2 times slower on the coprocessor than on the single host CPU. In general, we have been able to lower the overall execution time quite significantly. The early version of the code ran for approx. 5 min on single coprocessor while optimized version of our code runs for approx. 400 ms for the same input data.

It follows that the algorithm can easily handle many simultaneous route requests and the number of requests should be large enough to utilize the full power of the available hardware. Using the current implementation we can offload half of the work to the coprocessor and efficiently utilize the entire power of the HPC infrastructure.

In the future work, we would like to focus on further optimizations related to the cache memory subsystem of the coprocessor and to finding a vectorizable version of the algorithm. Another possible area of the research emerged during the optimization process. Some of the optimization techniques can be generalized for this type of Monte Carlo based algorithms by using some type of domain specific language, for example DSL and autotuning techniques being developed within the ANTAREX[7] project [7]. This way, the specific optimizations can be applied by users without the need to perform such extensive studying of the target hardware microarchitecture.

Acknowledgment. This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project 'IT4Innovations excellence in science - LQ1602', from the Large Infrastructures for Research, Experimental Development and Innovations project 'IT4Innovations National Supercomputing Center - LM2015070', and co-financed by the internal grant agency of VŠB - Technical University of Ostrava, Czech Republic, under the project no. SP2016/179 'HPC Usage for Transport Optimisation based on Dynamic Routing II'.

References

1. Fan, Y., Kalaba, R., Moore I, J.E.: Arriving on time. J. Optim. Theor. Appl. **127**(3), 497–513 (2005)
2. Hofleitner, A., Herring, R., Abbeel, P., Bayen, A.: Learning the dynamics of arterial traffic from probe data using a dynamic Bayesian network. IEEE Trans. Intell. Transp. Syst. **13**(4), 1679–1693 (2012)
3. Miller-Hooks, E.: Adaptive least-expected time paths in stochastic, time-varying transportation and data networks. Networks **37**, 35–52 (2000)

[7] ANTAREX (AutoTuning and Adaptivity appRoach for Energy efficient eXascale HPC systems), Project Reference: 671623, Call: H2020-FETHPC-2014 - http://www.antarex-project.eu.

4. Nikolova, E., Brand, M., Karger, D.R.: Optimal route planning under uncertainty. In: ICAPS, vol. 6, pp. 131–141 (2006)
5. Rahman, R.: Intel® Xeon Phi™ Coprocessor Architecture and Tools: The Guide for Application Developers. Apress, New York (2013)
6. Rice, J., Van Zwet, E.: A simple and effective method for predicting travel times on freeways. Intell. Transp. Syst. **5**(3), 200–207 (2004)
7. Silvano, C., Agosta, G., Cherubin, S., Gadioli, D., Palermo, G., Bartolini, A., Benini, L., Martinovič, J., Palkovič, M., Slaninová, K., Bispo, J., Cardoso, J.M.P., Abreu, R., Pinto, P., Cavazzoni, C., Sanna, N., Beccari, A.R., Cmar, R., Rohou, E.: The ANTAREX approach to autotuning and adaptivity for energy efficient HPC systems. In: ACM International Conference on Computing Frontiers 2016 (2016)
8. Tomis, R., Rapant, L., Martinovič, J., Slaninová, K., Vondrák, I.: Probabilistic time-dependent travel time computation using monte carlo simulation. In: Kozubek, T., Blaheta, R., Šístek, J., Rozložník, M., Cermák, M. (eds.) HPCSE 2015. LNCS, vol. 9611, pp. 161–170. Springer, Heidelberg (2016). doi:10.1007/978-3-319-40361-8_12
9. Yang, B., Guo, C., Jensen, C.S., Kaul, M., Shang, S.: Multi-cost optimal route planning under time-varying uncertainty. In: Proceedings of the 30th International Conference on Data Engineering (ICDE), Chicago, IL, USA (2014)