Analysis of the Execution-Time Variation of OpenMP-based Applications on the Intel® Xeon Phi™

Roberto Camacho Barranco and Patricia J. Teller

Technical Report: UTEP-CS-16-31,
University of Texas at El Paso, El Paso TX 79928, USA,
Contact: rcamachobarranco@miners.utep.edu

Abstract. The Intel® Xeon Phi™ accelerator is currently being used in several large-scale computer clusters and supercomputers to enhance the execution-time performance of computation-intensive applications. While performing a comprehensive profiling of the Intel® Xeon Phi™ execution-time behavior of different applications included in the Rodinia Benchmark suite, we observed large variations in application execution times. In this report we present the average execution times for different runs of each application. In addition, we describe the different steps taken to try to solve this problem. For example, a brief study was performed using one of these applications, i.e., a matrix-multiply kernel. By improving the vectorization of this application, the variation was reduced from an average of 25% to an average of 10%. However, the root cause of the remaining variation was not identified. Because the execution times of the other applications also exhibit similar levels of variation, we hypothesize that this execution-time variation could be caused by the hardware or by performance issues associated with how OpenMP is utilized.

Keywords: Intel® Xeon Phi™, execution time variation, repeatability

1 Execution-Time Variation

It is very important that the results of High Performance Computing (HPC) applications are reproducible. This is particularly true for application performance analysis, which requires that the analyzed workload contains a steady-state phase that provides consistent and repeatable results [7]. In this case, if there is variability in the execution times of an application that is executed several times on the same computing platform with the same input and runtime configuration, then the related experiments are considered to be non-reproducible. In other cases this variability is assumed to be negligible and, therefore, not taken into account. In fact, it is common practice to use statistical values such as the minimum, mean, median, and/or maximum, which do not account for the observed variability [22].

This report demonstrates the execution-time variability of applications executed on the Xeon Phi™. This is done by presenting the results of experiments

that employ ten different applications from the Rodinia Benchmark suite [2] and a matrix-matrix multiplication kernel executed on a standalone test bed, which consists of an Intel® Xeon E5-1650 host processor and an Intel® Xeon Phi™ 5110P accelerator. Of these eleven applications, the matrix-matrix multiplication kernel exhibits the largest execution-time variation, i.e., around 10% (up to 1.5 seconds difference) for jobs that each consist of 10 executions of the kernel using the same input and runtime configuration, i.e., number of cores and number of threads per core. Tables 1 and 2 present the average execution-time variation observed for each instance of the eleven applications, including the matrix-matrix multiplication kernel. As shown, each application was executed with from one to three input sets and with two, three, and four threads per core using from 1 to 60 cores.

As shown in Figure 1, which presents the mean percentage variation of the execution times of the different instances of the eleven applications along with standard deviations, it is clear that the K-means, Pathfinder, and Streamcluster applications have the largest variations. However, after further analysis, it was determined that the large variations observed for these three applications are mainly caused by randomly initialized variables. Thus, Figure 2 presents these findings without the data for the instances of these three applications. As can be seen by examining the figure, for the instances of the remaining eight applications, the average variation is approximately 5%, which could be considered negligible if the study was not quantifying application performance on a particular computer architecture. For example, if an application developer is studying the performance effect of an optimization that could provide a 5% increase in performance, such variability in execution time would not allow her to determine with certainty if the optimization is useful or not.

Non-negligible program performance variations for parallel OpenMP applications were demonstrated in [12], but the authors neither quantified nor qualified the factors that caused the variations in execution time. Nonetheless, the causes and possible ways to reduce execution-time variability (which depend on the application and its context) have been explored in other publications. Each cause and possible solution that are presented in the literature are not only discussed below, but are also explored through experiments that use the matrix-matrix multiplication kernel. In this respect, Sections 3, 4, and 5 focus on the different overheads associated with using OpenMP for parallelization, the jitter introduced by the Linux operating system, and hardware-related issues that can affect repeatability, respectively.

2 Experimental Setup

For the experiments presented in the following sections, we executed a matrix-matrix multiplication kernel with four different problem sizes, i.e., with 1,024 x 1,024, 2,048 x 2,048, 4,096 x 4,096, and 6,000 x 6,000 matrix sizes, on a standalone test bed that consists of an Intel® Xeon E5-1650 host processor and an Intel®

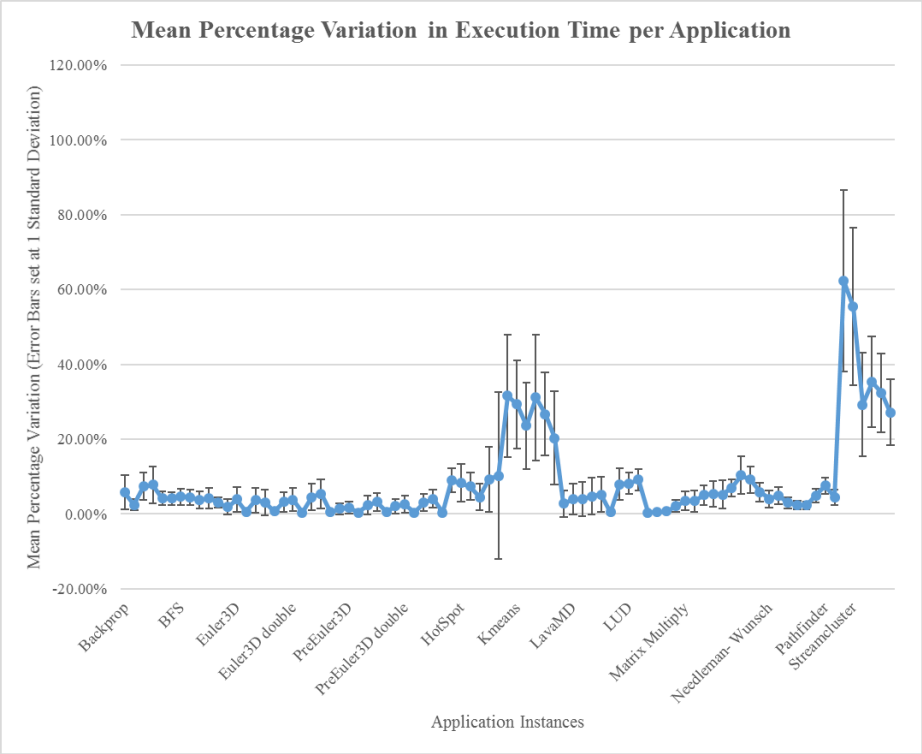

Fig. 1. Variation of execution time per application instance including standard deviation.

Xeon Phi™ 5110P accelerator. The accelerator was used in native mode and no other application was executed concurrently.

3 OpenMP Sources of Variation

As described in this section, the execution time of an OpenMP-based parallel application may vary due to several reasons, which are associated with OpenMP. Each of these are discussed next.

3.1 Thread binding and affinity

On the Intel® Xeon and Xeon Phi™ it is possible to restrict or bind the execution of virtual threads to a subset of the physical processing units (cores). For OpenMP, this binding or affinity can be set through KMP_AFFINITY and KMP_PLACE_THREADS. And, for MPI, it can be set through I_MPI_PIN, I_MPI_PIN_MODE, and I_MPI_PIN_PROCESSOR_LIST [15]. Using the default value for affinity might not be the best choice for the application under test

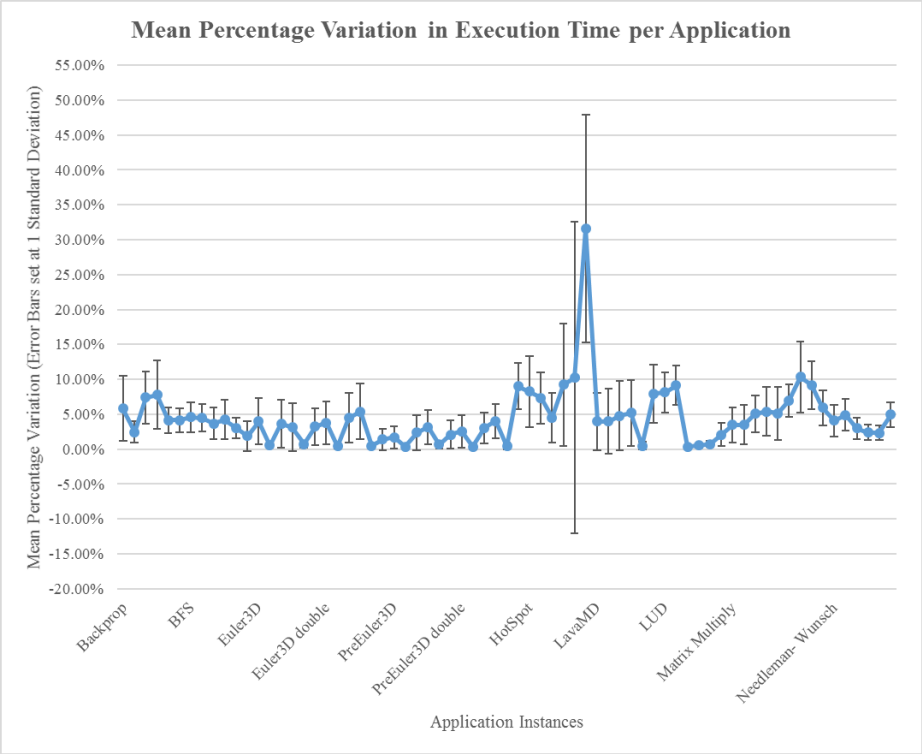

Fig. 2. Variation of execution time per application instance including standard deviation without randomly initialized applications.

and could cause execution-time variability. Therefore, the advice is to explicitly specify the affinity setting [15]. Another OpenMP parameter that can affect performance is KMP_BLOCKTIME, which defines the length of time that a thread waits before going to sleep. Setting this parameter to zero should help in cases where there is load imbalance [6]. And, finally, the KMP_AFFINITY parameter `permute`, which can be used with both *compact* and *scatter* affinity, controls which levels of the system topology map are most significant when it is being sorted. This parameter can be used to avoid binding multiple threads to the same core while leaving some unused since a thread normally executes faster on a core where it is not competing for resources with another active thread on the same core.

Figure 3 presents the execution-time variation of the matrix-matrix multiplication kernel executed on a maximum of 60 cores with four threads per core. The experiments differ with respect to the employed number of threads, affinity, problem size, and permute options. As shown in [11], binding threads to cores can significantly decrease execution-time variability; it did for most of the applications studied by the authors.

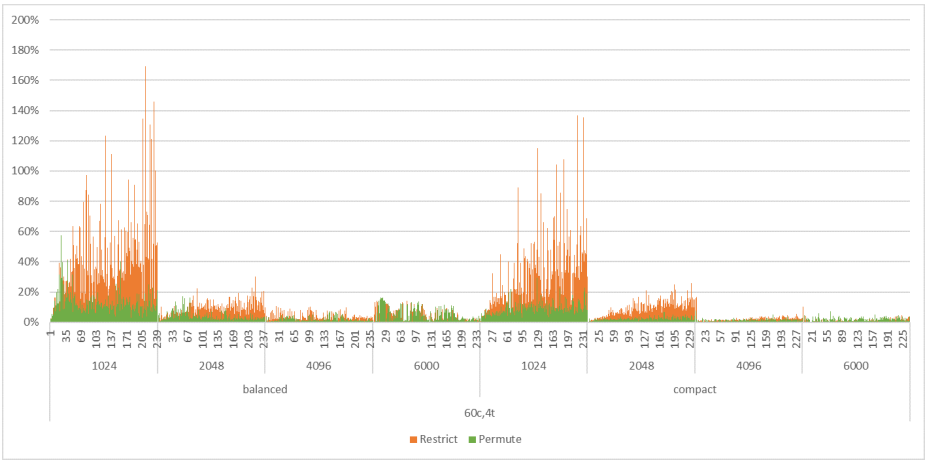

Fig. 3. Variation of execution time with different OpenMP affinity settings.

3.2 Overhead

The use of OpenMP incurs execution-time overhead, which can be tracked to different sources. For example, the actions associated with library startup, thread startup, per-thread loop scheduling, and lock management [5] consume a portion of the total execution time of an OpenMP-based application. This overhead represents the cost of parallelizing a code using OpenMP and often is negligible considering the speedup attained. However, for performance-related studies, the cause of execution-time variation can sometimes be tracked to the sources of OpenMP overhead. For example, it has been shown that: (1) the thread startup overhead, which averages around 170-190 microseconds, is incurred only the first time that threads are used [5] and (2) the first iteration of an OpenMP one-thread code takes about 62 microseconds longer than the subsequent iterations, while the OpenMP *parallel for* and *section* directives have thread startup overheads between 110 and 130 microseconds [10]. As mentioned in [8], reducing the effect of this overhead can be achieved by placing the computational phase of the code inside a loop and measuring the average execution time. This is a particularly good solution for small problems.

Parallel synchronization overhead is higher on the Intel® Xeon Phi™ than on the Xeon E5. This is because the Phi™, compared to the E5, has many more cores running at a lower frequency. This also increases the complexity of the coherence mechanism required to obtain high memory bandwidth [15].

Figures 4 and 5 show the overhead measured for each of the nested loops of the matrix-matrix multiplication kernel for problem sizes 1,024 x 1,024 and 4,096 x 4,096, respectively. The X-axis represents the index of the outer loop.

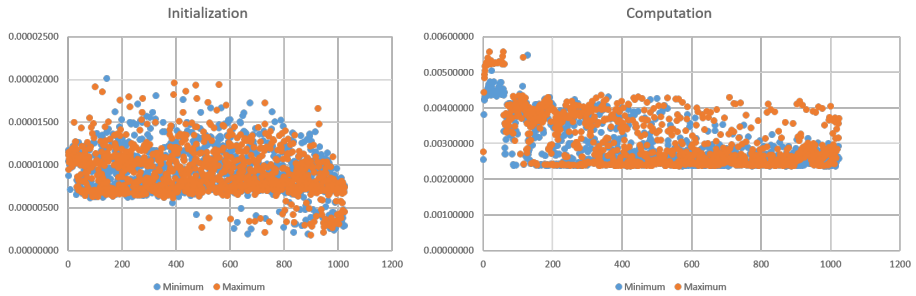

Fig. 4. OpenMP overhead per loop for the 1,024 x 1,024 matrix-matrix multiplication kernel.

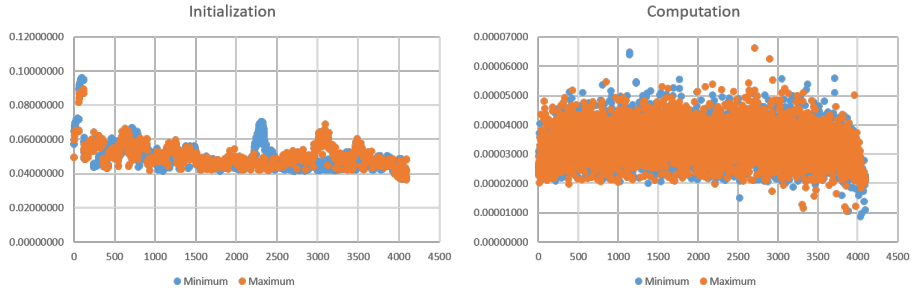

Fig. 5. OpenMP overhead per loop for the 4,096 x 4,096 matrix-matrix multiplication kernel.

3.3 Blocking

The blocking of OpenMP threads is caused by locks in the code, which can have a significant impact on performance. Thus, the use of locks in a program is discouraged and should be minimized. Nonetheless, note that the Intel® Thread Profiler can isolate blocking for further analysis. And, blocking can be reduced significantly by employing mechanisms like the double-checked locking optimization, which often can be used to produce lock-free parallelization [5].

3.4 Scheduling

Scheduling is used in OpenMP to keep the workload as balanced as possible across the available threads. The Intel® Thread Profiler can be useful to identify problems in thread scheduling. OpenMP's three different scheduling options, i.e., static, dynamic, and guided, allow the user to control thread scheduling [5]. Figure 6 shows the effect of the scheduling method on the execution-time variation of a 1,024 x 1,024 matrix-matrix multiplication executed on 60 Xeon Phi™ cores. The data in the figure were generated by varying the blocking time, i.e.,

using a blocking time of 0, 10, 50, and 5,000, and the chunk size, i.e, employing a chunk size of 1, 10, 100, and 500. As is made clear by the figure, increasing the chunk size increases the execution time and as the execution time increases the execution-time variation decreases.

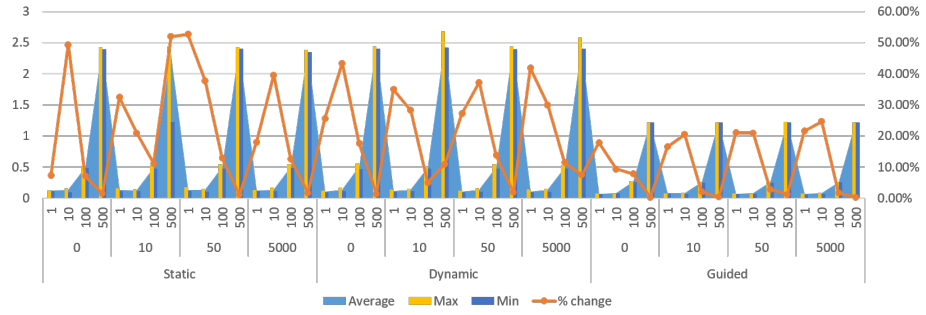

Fig. 6. Effect of OpenMP scheduling on the execution-time variation of the 1,024 x 1,024 matrix-matrix multiplication kernel executed on 60 Xeon Phi™ cores.

3.5 Compiler optimization level

Figure 7 presents data that indicate the effect of changing the optimization level of the compilation of the 1,024 x 1,024 matrix-matrix multiplication kernel with the Intel® OpenMP compiler. The data indicate that the effect of the optimization level is negligible as long as at least the O1 level is employed.

4 Operating System Jitter

Operating system (OS) jitter caused by system processes and daemons as well as concurrent user processes can introduce performance variations [3]. Several studies have been performed to investigate the events that cause this jitter and to quantify the effect that they have on performance [4, 16, 17, 20, 21, 24, 23]. For example, the research presented in [4] indicates that a slowdown of less than 1% of the user processes can result in a performance decrease of up to 50% when using a very large number of processors. And, the authors of [23] and [24] indicate that the impact of OS jitter increases with the number of the different computational phases present in the code. To ameliorate the effects of OS jitter on execution time variation, the authors of [17] recommend using no less than 30 runs per configuration and problem size, as well as using the median of the sample instead of the average, which is also suggested in [22]. The identification and/or quantification of the sources of execution-time variation and the impact of system activity on application performance can be accomplished by employing the different methodologies presented in [4] and [18].

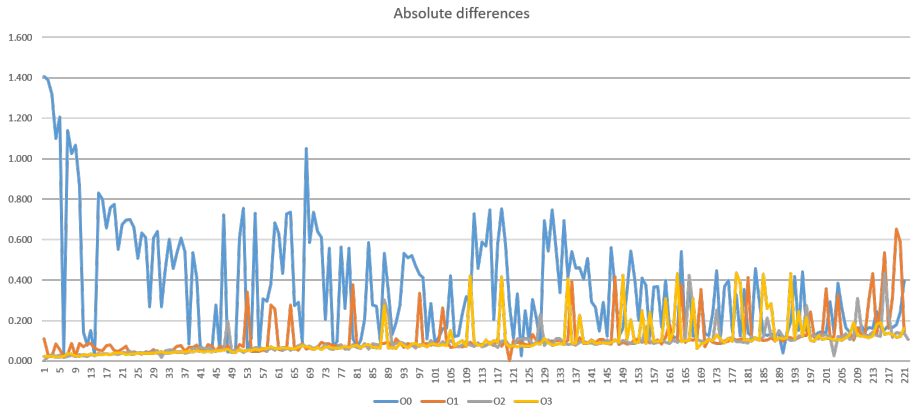

Fig. 7. Effect of the compiler optimization level on the execution-time variation of the 1,024 x 1,024 matrix-matrix multiplication kernel.

5 Hardware-Related Sources of Variation

There are several hardware-related sources of execution-time variation and the variation of computational results. We discuss six of these sources below.

5.1 Memory-access related sources

Memory access can be a performance bottleneck for an application. The differences in how memory is accessed by a program can result in differences in its execution time. For example, in [14] a behavior particular to the Xeon Phi^{TM} is identified where an L2-cache eviction results in the same data being evicted from the L1 cache, which can result in some threads running significantly slower than others. Also in relation to cache behavior, the authors of [7] suggest that warm-cache effects could also cause execution time variation. Furthermore, in [13] the effects of L2-cache sharing, automatic hardware prefetching, and memory page sizes are studied but considered insignificant with respect to the effects of thread binding.

5.2 Variation in results

The focus of the research presented in [9] is not differences in execution time but on differences in the actual result that an algorithm returns after performing floating-point operations with identical input data on an identical processor. For serial code, the identified cause for this behavior is variation in data alignment that result from changes in the execution environment, which change the order of reduction operations. Suggested ways to diminish the effects of this problem include explicit alignment of data and the use of specific compiler flags. However, this can result in performance degradation. For parallel code that contains

reductions, the parallel decomposition of the operations can change from run to run, which cause variations in the results. However, the observed variation is typically very small, but may become significant due to cancellations caused by an algorithm.

5.3 Concurrent jobs / contention of resources

Executing more than one job on the same CPU can have a significant affect on performance. For example, in [1] the authors mention that in multi-core architectures concurrent cache contention and task migration can cause significant variations in execution time. Other factors that contribute to this variation include prefetching-hardware contention, memory-controller contention, and memory-bus contention. As shown in [25], the variation of execution time caused by these factors ranged from 60% to 100% for different executions of the same program. Accordingly, the authors of [7] suggest that when analyzing the performance of an application, it should be the only application that consumes a significant portion of CPU time when performance data is being collected.

In [19] the variability caused by concurrently-executing applications is predicted by studying different possible overlaps of the phases of the different applications. And, it is shown in [11] that executing separate co-running processes in parallel can improve user-level application execution time and can reduce the number of conflicts at shared cache levels.

5.4 Asymmetry between cores

In [1] the authors mention that in some cases processors present undocumented asymmetry between their cores with respect to memory bandwidth or when using features such as Turbo Boost.

5.5 Dynamic Voltage Scaling

The research presented in [11] shows that high variations in execution time are present even when dynamic voltage scaling (DVS) and the automatic hardware prefetcher are disabled. However, when enabled, DVS and automatic prefetching can be sources of high execution-time variation.

5.6 Device temperature

To investigate the correlation of execution-time variation and the temperature of the device we used the `micsmc -t` utility to measure the execution time and the accelerator's temperature of 15 executions of a 1,024 x 1,024 matrix-matrix multiplication. To detect observable trends, each experiment was repeated four times, each time after a cool-off period.

Figure 8 presents the results of these experiments. As indicated, for this application, in some cases there is a slight correlation between execution-time variation and temperature.

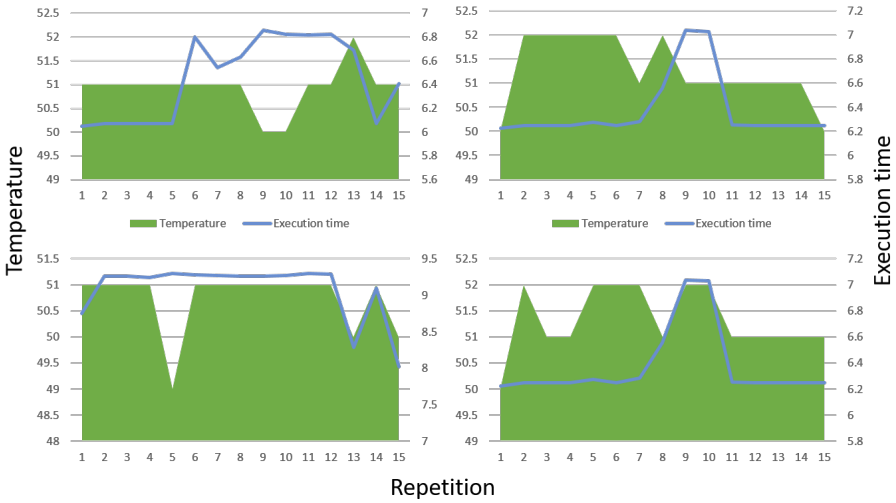

Fig. 8. Effect of temperature on the execution-time variation of the 1,024 x 1,024 matrix-matrix multiplication kernel.

References

1. T. P. Baker. What to make of multicore processors for reliable real-time systems? In *Proc. of the Int. Conf. on Reliable Software Technologies*, pages 1–18. Springer, 2010.
2. S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the Int. Symp. on Workload Characterization*, pages 44–54. IEEE, 2009.
3. Dr. Dobb's. Programming Intel's Xeon Phi: a jumpstart introduction, 2015. Retrieved from: http://www.drdobbs.com/parallel/programming-intels-xeon-phi-a-jumpstart/240144160?pgno=2.
4. R. Gioiosa, F. Petrini, K. Davis, and F. Lebaillif-Delamare. Analysis of system overhead on parallel computers. In *Proc. of the 4th IEEE Int. Symp. on Signal Processing and Information Technology*, pages 387–390. IEEE, 2004.
5. Intel. Performance obstacles for threading: how do they affect OpenMP code?, 2009. Retrieved from: https://software.intel.com/en-us/articles/performance-obstacles-for-threading-how-do-they-affect-openmp-code.
6. Intel. OpenMP* thread affinity control, 2012. Retrieved from: https://software.intel.com/en-us/articles/openmp-thread-affinity-control.
7. Intel. Optimization and performance tuning for Intel Xeon Phi coprocessors, Part 2: Understanding and using hardware events, 2012. Retrieved from: https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding.
8. Intel. Tips to measure the performance of Intel MKL with small matrix sizes, 2012. Retrieved from: https://software.intel.com/en-us/articles/a-simple-example-to-measure-the-performance-of-an-intel-mkl-function.

9. Intel. Run-to-run reproducibility of floating-point calculations for applications on Intel Xeon Phi coprocessors (and Intel Xeon processors), 2013. Retrieved from: https://software.intel.com/en-us/articles/run-to-run-reproducibility-of-floating-point-calculations-for-applications-on-intel-xeon.
10. P. Lindberg. Basic OpenMP threading overhead, 2009. Retrieved from: https://software.intel.com/en-us/articles/basic-openmp-threading-overhead.
11. A. Mazouz, S. A. A. Touati, and D. Barthou. Measuring and Analysing the Variations of Program Execution Times on Multicore Platforms: Case Study. Research report, 2010.
12. A. Mazouz, S. A. A. Touati, and D. Barthou. Study of variations of native program execution times on multi-core architectures. In *Proc. of the Int. Conf. on Complex, Intelligent and Software Intensive Systems*, pages 919–924. IEEE, Feb 2010.
13. A. Mazouz, S. A. A. Touati, and D. Barthou. Analysing the variability of OpenMP programs performances on multicore architectures. In *Proc. of the 4th Workshop on Programmability Issues for Heterogeneous Multicores*, page 14. HAL, Jan. 2011.
14. J. McCalpin. Randomly slower cores, 2015. Retrieved from: https://software.intel.com/en-us/forums/topic/392340.
15. J. D. McCalpin. Native computing and optimization on the Intel Xeon Phi coprocessor, 2013. Retrieved from: https://portal.tacc.utexas.edu/documents/13601/933270/MIC_Native_2013-11-16.pdf/56b4a5c9-be24-4c41-8625-eee21879ca8b.
16. A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero. A quantitative analysis of os noise. In *Proc. of the Int. Parallel and Distributed Processing Symp.*, pages 852–863. IEEE, May 2011.
17. P. E. Nogueira, R. Matias, Jr., and E. Vicente. An experimental study on execution time variation in computer experiments. In *Proc. of the 29th Annual ACM Symp. on Applied Computing*, pages 1529–1534. ACM, 2014.
18. F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance. In *Proc. of the ACM/IEEE Conf. on Supercomputing*, page 55. ACM, 2003.
19. A. Sandberg, A. Sembrant, E. Hagersten, and D. Black-Schaffer. Modeling performance variation due to cache sharing. In *Proc. of the 19th Int. Symp. on High Performance Computer Architecture*, pages 155–166. IEEE, Feb. 2013.
20. D. Skinner and W. Kramer. Understanding the causes of performance variability in HPC workloads. In *Proc. of the IEEE Workload Characterization Symp.*, pages 137–149. IEEE, Oct 2005.
21. M. Sottile and R. Minnich. Analysis of microbenchmarks for performance tuning of clusters. In *Proc. of IEEE Int. Conf. on Cluster Computing*, pages 371–377. IEEE, Sept 2004.
22. S. A. A. Touati, J. Worms, and S. Briais. The speedup-test: a statistical methodology for programme speedup analysis and computation. *Concurrency and Computation: Practice and Experience*, 25(10):1410–1426, 2013.
23. E. Vicente and R. Matias. Exploratory study on the Linux OS jitter. In *Proc. of the Brazilian Symp. on Computing System Engineering*, pages 19–24. IEEE, Nov 2012.
24. E. Vicente and R. Matias. Modeling and simulating the effects of OS jitter. In *Proc. of the Winter Simulations Conference*, pages 2151–2162. IEEE, Dec 2013.
25. S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGARCH Comput. Arch. News*, 38(1):129–142, Mar. 2010.

Table 1. Execution-time variation (Part I)

Application	Input	Threads per core	Mean Percentage Error	Standard deviation
Backprop	10000000	2	7.53%	3.20%
		3	5.80%	4.66%
		4	2.44%	1.53%
	20000000	2	7.39%	3.74%
		3	7.76%	4.91%
		4	4.16%	1.84%
BFS	8M	2	4.15%	1.75%
		3	4.57%	2.18%
		4	4.48%	1.99%
	16M	2	3.69%	2.25%
		3	4.22%	2.82%
		4	3.04%	1.45%
Euler3D	193K	2	1.90%	2.13%
		3	3.96%	3.32%
		4	0.55%	0.29%
	0.2M	2	3.64%	3.39%
		3	3.14%	3.44%
		4	0.71%	0.46%
Euler3D double	193K	2	3.21%	2.62%
		3	3.77%	3.09%
		4	0.41%	0.28%
	0.2M	2	4.48%	3.56%
		3	5.36%	3.99%
		4	0.47%	0.27%
PreEuler3D	193K	2	1.41%	1.54%
		3	1.73%	1.59%
		4	0.38%	0.29%
	0.2M	2	2.36%	2.50%
		3	3.19%	2.45%
		4	0.66%	0.55%
PreEuler3D double	193K	2	2.09%	1.98%
		3	2.53%	2.29%
		4	0.35%	0.37%
	0.2M	2	3.04%	2.20%
		3	4.02%	2.42%
		4	0.43%	0.53%
HotSpot	1024 x 1024	2	9.00%	3.28%
		3	8.26%	5.06%
		4	7.36%	3.69%
	8192 x 8192	2	4.53%	3.55%
		3	9.26%	8.77%
		4	10.21%	22.33%
Kmeans	kdd_cup	2	31.60%	16.31%
		3	29.29%	11.73%
		4	23.60%	11.53%
	819200	2	31.13%	16.84%
		3	26.72%	11.17%
		4	20.31%	12.50%
LavaMD	10 boxes per dimension	2	2.77%	3.57%
		3	3.96%	4.14%
		4	3.97%	4.66%
	20 boxes per dimension	2	4.77%	4.95%
		3	5.18%	4.73%
		4	0.49%	0.56%

Table 2. Execution-time variation (Part II)

Application	Input	Threads per core	Mean Percentage Error	Standard deviation
LUD	512 x 512	2	7.93%	4.21%
		3	8.14%	2.88%
		4	9.15%	2.87%
	2048 x 2048	2	0.35%	0.28%
		3	0.52%	0.34%
		4	0.74%	0.44%
Matrix Multiply	2048 x 2048	2	2.09%	1.65%
		3	3.46%	2.52%
		4	3.48%	2.82%
	4096 x 4096	2	5.06%	2.65%
		3	5.39%	3.46%
		4	5.15%	3.80%
	6000 x 6000	2	6.97%	2.33%
		3	10.33%	5.05%
		4	9.16%	3.47%
Needleman- Wunsch	4096	2	5.91%	2.51%
		3	4.07%	2.28%
		4	4.90%	2.24%
	8192	2	3.01%	1.52%
		3	2.38%	1.13%
		4	2.34%	1.05%
Pathfinder	10000 x 100000	2	4.93%	1.75%
		3	7.58%	2.18%
		4	4.48%	1.99%
Streamcluster	10 x 20 x 256	2	62.32%	24.30%
		3	55.47%	21.07%
		4	29.23%	13.88%
	30 x 40 x 1024	2	35.27%	12.15%
		3	32.37%	10.46%
		4	27.12%	8.81%