Int J Parallel Prog (2017) 45:1091–1107
DOI 10.1007/s10766-016-0455-0

Benchmarking Performance of a Hybrid Intel Xeon/Xeon Phi System for Parallel Computation of Similarity Measures Between Large Vectors

Paweł Czarnul[1]

Received: 14 February 2016 / Accepted: 22 September 2016 / Published online: 29 September 2016

Abstract The paper deals with parallelization of computing similarity measures between large vectors. Such computations are important components within many applications and consequently are of high importance. Rather than focusing on optimization of the algorithm itself, assuming specific measures, the paper assumes a general scheme for finding similarity measures for all pairs of vectors and investigates optimizations for scalability in a hybrid Intel Xeon/Xeon Phi system. Hybrid systems including multicore CPUs and many-core compute devices such as Intel Xeon Phi allow parallelization of such computations using vectorization but require proper load balancing and optimization techniques. The proposed implementation uses C/OpenMP with the offload mode to Xeon Phi cards. Several results are presented: execution times for various partitioning parameters such as batch sizes of vectors being compared, impact of dynamic adjustment of batch size, overlapping computations and communication. Execution times for comparison of all pairs of vectors are presented as well as those for which similarity measures account for a predefined threshold. The latter makes load balancing more difficult and is used as a benchmark for the proposed optimizations. Results are presented for the native mode on an Intel Xeon Phi, CPU only and the CPU + offload mode for a hybrid system with 2 Intel Xeons with 20 physical cores and 40 logical processors and 2 Intel Xeon Phis with a total of 120 physical cores and 480 logical processors.

Keywords Many-core architectures · Hybrid parallelism · Parallelization of computing similarity measures · Intel Xeon · Intel Xeon Phi · Optimization

✉ Paweł Czarnul
pczarnul@eti.pg.gda.pl

[1] Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology, ul. G. Narutowicza 11/12, 80-233 Gdańsk, Poland

1 Introduction

In the modern high performance computing landscape, several possibilities have arisen for multi-core and many core computing. Multi-core CPUs offer powerful cores clocked with relatively high frequency, sophisticated instruction sets suitable for various codes including multiple program multiple data codes executed by several threads. On the other hand, accelerators such as GPUs have enabled efficient parallelization of massively parallel single program multiple thread type codes. Coprocessors such as Intel Xeon Phi x100 (Knights Corner) rely on Pentium type cores with in-order execution for efficient processing on roughly 240 threads per device.

These hardware advancements have enabled new approaches to certain problems. This is because proper optimizations may considerably affect performance. For instance, on an Intel Xeon Phi [3], high efficiency is possible if enough parallelism is exposed with the application and a high level of vectorization is achieved. Potential limitations may include false sharing, issues with lack of data locality/inefficient cache use etc.

In this paper, we focus on running of parallel codes on a hybrid Xeon/Xeon Phi system using an example of computing similarity measures of large vectors in order to find settings allowing efficient execution.

2 Related Work and Motivation

Computing similarity measures of vectors has many uses in real world applications. It is used for assessment of similarity of words, text documents, web pages, images etc. and can be applied in a variety of contexts [36,37]. Computing similarities between words has applications such as clustering of documents, detection of relations, disambiguation of entities, finding meta data etc. [29]. Comparison of text documents and web pages can be used for detecting plagiarism. Finding similarities between text fragments is used in web searching e.g. for suggestion of related queries [34]. Similarities between objects or object sets is used in medicine such as in finding clinical data applicable to the current clinical case [18]. Furthermore, computing similarity measures between sequences of vectors representing objects can be used in many applications such as in biology, audio recognition or handwritten word image retrieval as discussed in paper [31].

Many existing works [1,2,5,13,16,39] consider and optimize the problem of all pair similarity search which is to find such pairs of vectors for which the similarity measure exceeds a given threshold. In such a case, optimization techniques can be used to tune the algorithm itself. Various environments for running algorithms are considered, for example: CPUs [5,16], a multi-node cluster with multi-core CPUs and Hadoop MapReduce [1,13]. Work [24] considers computing similarity between terms by performing comparison between corresponding pmi context vectors with the following measures cosine, Jaccard, and Dice. Paper [24] uses an approach similar to sparse matrix multiplication strategy. In contrast to these works, this paper generally focuses on a scenario in which all pairs of vectors need to be analyzed or no implicit knowledge about similarity measure can be assumed.

The problem of parallel computing similarity measures between vectors was also previously considered in [11]. The paper focused on creation of a model of the algorithm. It also described an environment called MERPSYS that allows simulation of execution time of a parallel application run on a potentially large scale system, energy consumed by the application on resources as well as reliability of the run, being important for very large scale systems. In [11] the model was validated across real runs which allowed extrapolation of results of this application for other data and system (in terms of the number of nodes) sizes.

Intel Xeon Phi seems to be an interesting candidate for this type of problem and computations due to the large number of cores available and efficient vectorization that itself allows speeding up the code up to 8 (double precision) or 16 (single precision) times [30,38]. Implementation challenges [6,19], however, include maximizing locality for cache use, elimination of false sharing, maximizing memory bandwidth, using proper thread affinity, dealing with potential memory limitations. Intel Xeon Phi has enabled parallelization of many real problems. In [17] large scale feature matching using a linear algorithm is presented. Paper [21] presents an algorithm for large-scale DNA analysis for various numbers of cores and thread affinities. Paper [23] presents acceleration of the subsequence similarity search based on DTW distance. In work [10] performing a divide-and-conquer application on an Intel Xeon Phi is optimized using a lightweight framework in OpenMP. In work [22] programming models and best practices are presented that enable making the most of the Xeon Phi coprocessor. Performance of sparse matrix vector (SpMV) and sparse matrix multiplication (SpMM) is analyzed for various compute devices in paper [35]. For instance, for SpMV Sandy CPUs were approximately twice as fast than Westmere with a performance between 4.5 and 7.6GFlop/s, NVIDIA K20 obtains between 4.9 and 13.2 GFlop/s while Xeon Phi reaches and exceeds 15GFlop/s. The authors of [14] have benchmarked Xeon Phi for leukocyte tracking. They argue that for this particular application vectorization requires intervention from the programmer. Furthermore, they compared performance of an implementation on Xeon Phi to that on NVIDIA K20 with the latter offering better performance due to efficient reduction performed in shared memory. Paper [20] presents that 1 TFlop/s performance can be reached for Intel Xeon Phi when using 240 threads on a 60 core model for large data sizes for processing of astronomical data. In [32], the author benchmarked a multi phase Lattice Boltzmann code run on Intel Xeon Phi. It is shown how various thread affinities such as compact, scatter, balanced impact performance. The code achieved the performance of 2.5x dual Intel Xeon E5-2680 socket nodes. Overheads of typical OpenMP constructs was in turn benchmarked in work [7]. Even more challenging is execution in a hybrid environment with various performances of cores, communication between compute devices and potentially specific optimization for the latter. Nodes that contain Xeon Phi coprocessors can be integrated into clusters for even higher performance, especially with Infiniband. In [26–28], the authors tune the MVAPICH2 MPI library and demonstrate considerable improvements in performance of intra-MIC, intranode and internode communication over the standard implementation. Furthermore, services installed on HPC clusters or servers that incorporate multicore CPUs and many core systems can be integrated into computationally oriented workflow systems [8,9].

Generally, the problem considered in this paper can be stated as optimization of parallel computations of a given similarity measure between every pair out of `vector_count` vectors on a hybrid Xeon/Xeon Phi architecture. Each vector has `dim_size` elements in it. For the test purposes, the similarity measure used in experiments was a square root of the sum of differences of values on corresponding dimensions power 2. Additionally, for the sake of testing load balancing in the case computations of various pairs of vectors take various amounts of time, finding pairs for which similarity exceeds a threshold is also considered. Successive vector elements are assigned values in the [1, 2] range.

3 Testbed Implementation, Optimization and Testing

3.1 Initial Implementation, Optimization and Tests on Intel Xeon Phi

For the initial approach we started with the idea proposed in [11] which was tested there in a cluster environment using MPI. The application itself consists of a few steps: allocation and initialization of vectors, computations of similarity values between every pair of vectors and storing results in memory, computing the minimum difference between two vectors across all pairs.

In this paper, we started with this basic implementation, which was ported to C+OpenMP. It follows these steps:

1. allocation and initialization of input data,
2. entering a `#pragma omp parallel` block in which each thread:
   - fetches its id,
   - goes through a loop iterating through `first_vector`—index of a first vector other vectors from a batch would be compared to,
   - goes through a loop iterating through batches of vectors; each batch has a predefined size (that in case of one version decreases in time); each thread selects its own batch based on the thread id,
   - each thread compares the current `first_vector` to each of the vectors in the batch and stores similarity measures for each pair,
3. the largest similarity measure is found as it enforces dependence on all pair vector comparisons.

For performance testing of the code, several versions were run and compared on an Intel Xeon Phi Coprocessor 5000 series with 8 GBs of memory, clocked at 1.053 GHz with 60 cores (240 threads):

- `m1`—the presented basic version following the concept presented in [11]. Separate, dynamically allocated arrays with alignment are used for reading input vectors and storage of results.
- `m2`—in this version of the code an additional array is used so that each iteration of the loop through vectors in a second batch are independent and results are stored in temporary space for each vector being compared to.
- `m21`—this is the `m2` version modified in such a way that initialization of the temporary result space are performed first for each vector of the second batch,

Table 1 Execution times (s) for various versions of the code, 10,000 vectors, `dim_size = 10,000`, balanced affinity, `batch_size = 2500`

Version of code	Execution time (s)
m1	32.59
m2	32.08
m21	31.31
m21-vbs	31.12

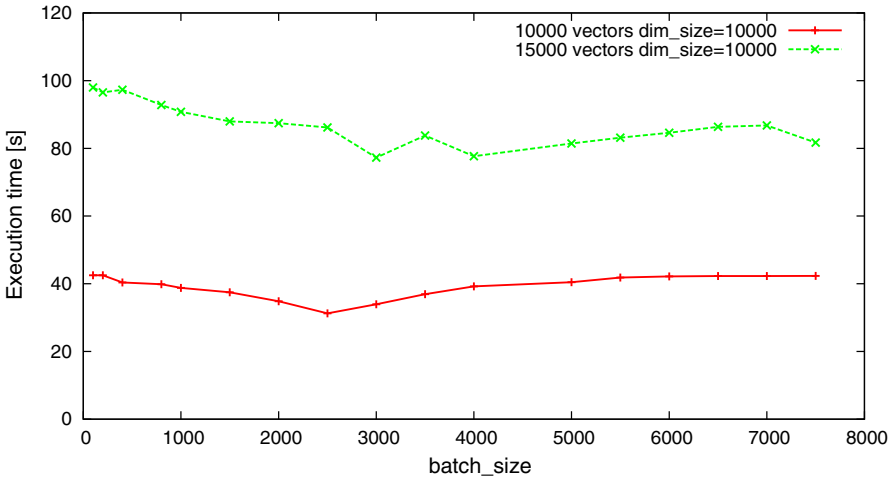

Fig. 1 Execution time versus batch size

then computations of similarities are performed for each vector of the second batch and finally copying of whole results for a batch is performed.

– `m21-vbs`—the `m21` version with a decreasing batch size. The algorithm is as follows:

```
threshold_step=vector_count/2;
threshold=vector_count/2;

(...)
  // in the main loop iterating through first_vector
  if ((first_vector>threshold) && (batch_size>batch_size_threshold)) {
    batch_size/=2;
    threshold_step/=2;
    threshold+=threshold_step;
  }
(...)
```

Successive versions brought small improvements in execution time tested for 10,000 vectors, `dim_size = 10,000`, balanced affinity, `batch_size = 2500` as shown in Table 1. For `m21-vbs` 78 was used as `batch_size_threshold`.

Figure 1 presents results of version `m21-vbs` for various sizes of a batch for the number of vectors equal to 10,000 and the dimension size equal to 10,000. It can be seen that the best results are obtained for the batch size equal to 2500.

Additionally, tests for various values of `batch_size_threshold` were performed. Figure 2 shows execution times for the best batch size 2500 and various

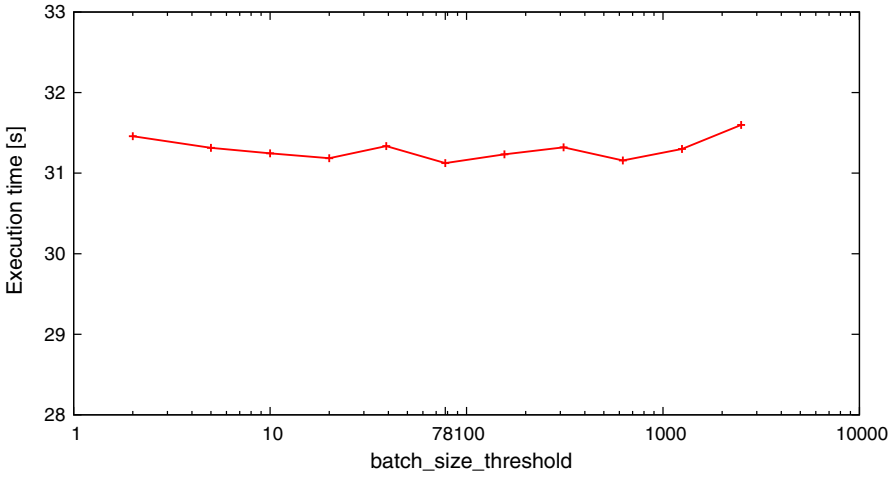

Fig. 2 Execution time versus `batch_size_threshold`

lowest batch size limits. It can be seen that execution times are the lowest for middle values of the batch size limit. The value of 78 was then used in subsequent tests as it gives smallest execution times.

However, the proposed code has several limitations and can be optimized even further, especially for a hybrid environment including Intel Xeon Phi. Potential optimizations include:

1. increasing computations to communication/synchronization ratio even further as comparison of a single first vector to a batch of vectors may not be optimal,
2. dynamic load balancing among compute cores of various performance,
3. optimizing the best number of cores used per device,
4. applying optimization techniques such as loop tiling etc.

3.2 Implementation, Optimization and Tests on a Hybrid Intel Xeon/Xeon Phi System

Based on the aforementioned code and results, a new version of the code was developed that would be suitable for efficient execution in a hybrid CPU-accelerator environment. Specifically, the target API was C+OpenMP for execution in the CPU + offload mode.

For the following tests, a server with 128 GBs of RAM and the following processors was used:

1. 2x Intel Xeon CPU E5-2680 v2 @ 2.80GHz CPUs,
2. 2x Intel Xeon Phi Coprocessor 5000 series each with 8GBs of memory, clocked at 1.053 GHz with 60 cores (240 threads).

In this environment, several matters may impact performance of the code:

1. potential imbalance due to various total performance of host Xeon and Xeon Phi cores,

2. parallelization potential of the code assuming 40 logical processors of the CPUs (with HT) and 2x240 threads of two Xeon Phi coprocessors,
3. communication and synchronization overhead between the host and the Xeon Phis (PCI-E) as well as within the host and coprocessors.

In order to mitigate these issues, the code optimizations from the `m21-vbs` version were improved with the following modifications:

– In order to increase computation/communication ratio of processing a batch, batches for the "first vector" are considered to which vectors of the previously mentioned batch is compared (this is now called second vector batch). Specifically, the following pseudocode illustrates which vectors are compared to which ones in this case:

```
for(cc=first_vector;cc<first_vector_batch_size;cc++) {
  if (second_vector_batch_start<=cc) second_vector_batch_start=cc+1;
  for(i=second_vector_batch_start;i<min(vector_count,
  second_vector_batch_start+second_vector_batch_size);i++)
    compare(vectors[cc],vectors[i]);
}
```

It should be noted that while vectors of the "first vector" batch are not compared, the code generates second vector batches (equivalent to batches in the first version considered in Sect. 3.1) starting from after the first vector of the first vector batch so that all pairs of vectors are finally compared.

– The code was designed to make the most of all cores of Xeon and Xeon Phis with some cores used by managing and other cores by computing threads. At the highest level the number of threads equal to the number of (1 + number of Xeon Phis used) is launched which fetch successive batches (dynamic assignment of batches to idle threads managing CPUs or Xeon Phis) and manage computations in lower level parallel regions (nested parallelism enabled): using a `#pragma omp parallel` block for the host CPUs and `#pragma offload` and next `#pragma omp parallel` for Xeon Phi cards. It should be noted that `out` clauses in the offloads for two Xeon Phis copy results to distinct arrays. Apart from the managing host threads, the `#pragma omp parallel` constructs use `num_threads(x)` clauses to specify the number of computing threads used for CPUs or a Xeon Phi. On the host the best number of computing threads launched turned out to occupy (<the number of physical cores> − <the number of Xeon Phi cards used>) physical cores. For each physical core 2 computing threads were launched. On the Xeon Phi in the offload mode the best number of threads launched turned out to be 236. In the hybrid CPU + Xeon Phi tests, the following numbers of computing threads were used:
  – CPUs + 1 Xeon Phi − 38 threads for CPUs, 236 threads for Xeon Phi,
  – CPUs + 2 Xeon Phis − 36 threads for CPUs, 236 threads for each Xeon Phi.

  For CPU only scalability tests, up to 40 computing threads were used. Within the blocks for `#pragma omp parallel` constructs the same parallelization strategy was used for CPUs and Xeon Phis i.e. `#pragma omp for schedule(dynamic)` for loop parallelization with dynamic assignment of iterations to threads.
– Various values of the first vector batch and the second vector batch influence load balance but also synchronization overheads thus proper configurations need to be obtained for best execution times.

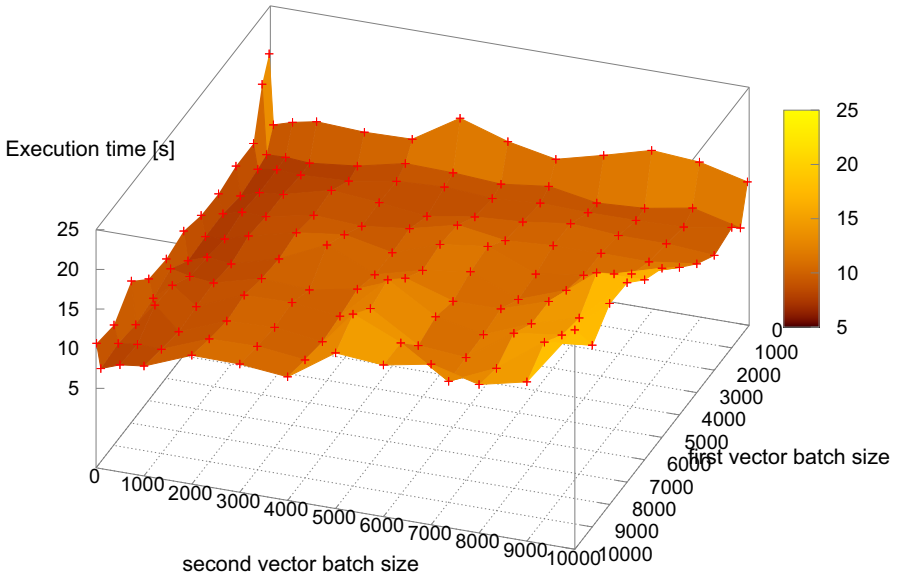

Fig. 3 host+1x mic—execution time (s) versus first vector batch size and second vector batch size, 10,000 vectors, `dim_size = 10,000`, dynamic batch adjustment, `batch_size_threshold = 78`

– Loop tiling when comparing a given first vector to a second vector. In this case, the loop going through dimensions of the second vector was partitioned into batches of 1024 iterations.
– Optimization using overlapping communication and computations in a hybrid environment [12].
– Setting proper parameters specific to a device such as `MIC_USE_2MB_BUFFERS` [12] as well as proper thread affinity [4,15].

The following tests clearly demonstrate impact of these optimizations and configuration settings on performance of the application. Firstly, the application with all these improvements was tested for various values of the first vector batch size and second vector batch sizes. For 10,000 vectors each 10,000 dimension in size Fig. 3 presents execution times in a hybrid environment with 2 host CPUs and 1 Xeon Phi (mic) card while Fig. 4 presents execution times in a hybrid environment with 2 host CPUs and 2 Xeon Phi cards. For 10,000 vectors each 40,000 dimension in size Fig. 5 presents execution times in a hybrid environment with 2 host CPUs and 1 Xeon Phi card while Fig. 6 presents execution times in a hybrid environment with 2 host CPUs and 2 Xeon Phi cards. The following settings were used: `export MIC_ENV_PREFIX = PHI`, `export PHI_KMP_AFFINITY = granularity = fine,compact`, `export MIC_USE_ 2MB_BUFFERS = 64k`. It can be seen very clearly that there is an area of best settings with the second vector batch size around 100 and the first vector batch size is around 1000–2000 for best execution times. Either smaller or larger settings result in higher overall execution times. This is because smaller values result in larger overhead to the number of packets that need to be handled while larger sizes do not allow good balancing. Tests have revealed that these settings are also appropriate for

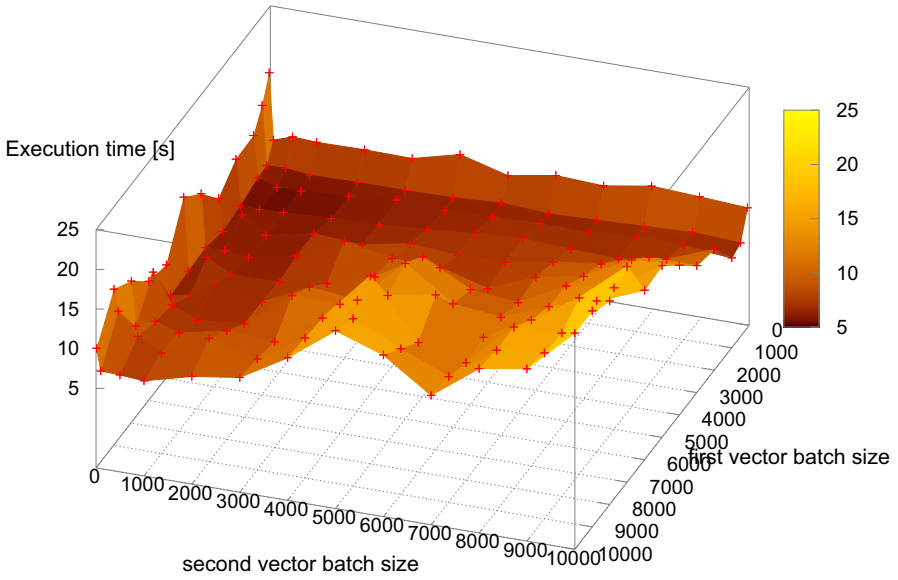

Fig. 4 host+2x mic—execution time (s) versus first vector batch size and second vector batch size, 10,000 vectors, `dim_size = 10,000`, dynamic batch adjustment, `batch_size_threshold = 78`

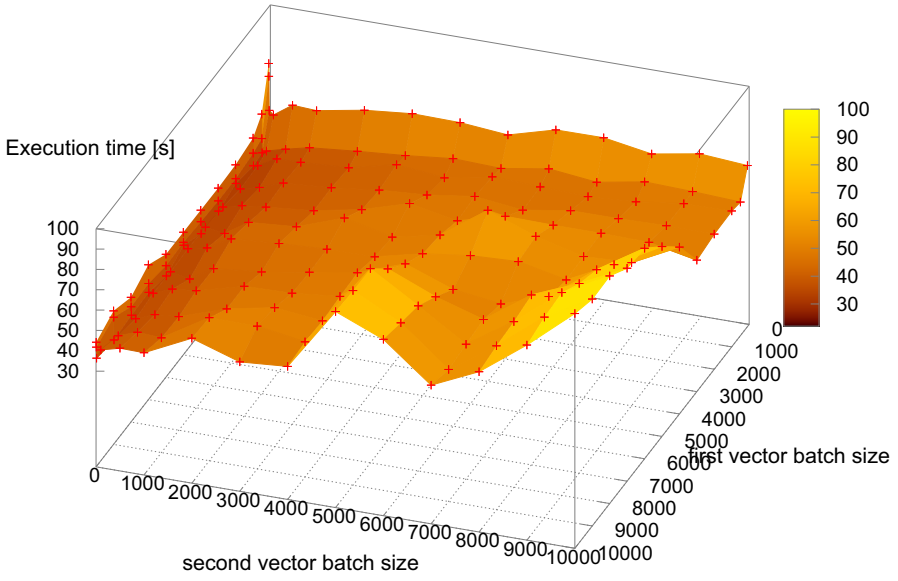

Fig. 5 host+1x mic—execution time (s) versus first vector batch size and second vector batch size, 10,000 vectors, `dim_size = 40,000`, dynamic batch adjustment, `batch_size_threshold = 78`

several other combinations of the total number of vectors and dimension sizes tested next. These are different values than those obtained for the initial version discussed in Sect. 3.1 due to consideration of first vector batches in the improved version. However,

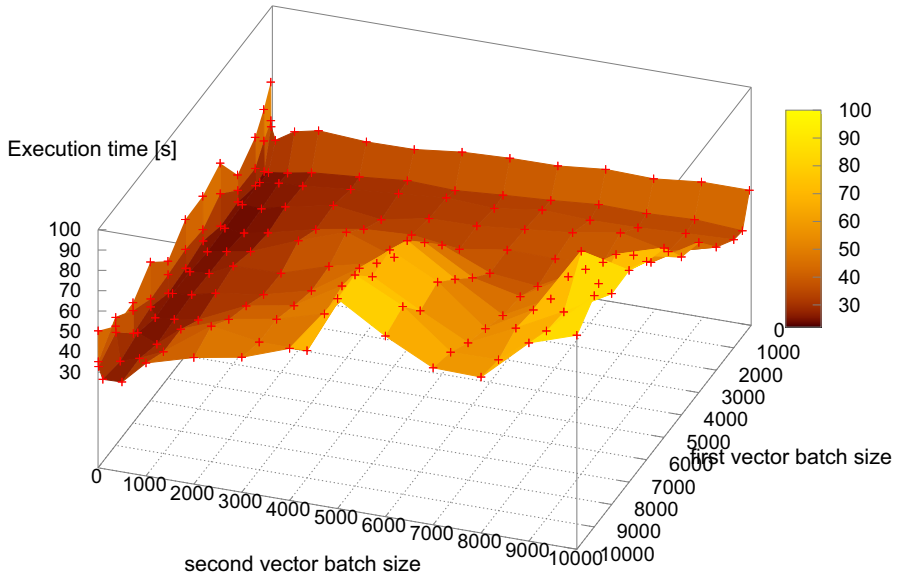

Fig. 6 host+2x mic—execution time (s) versus first vector batch size and second vector batch size, 10,000 vectors, `dim_size = 40,000`, dynamic batch adjustment, `batch_size_threshold = 78`

tests have revealed that the value `batch_size_threshold = 78` is still suitable for the improved version as evidenced by results shown in Fig. 8. Various settings of `batch_size_threshold` gave marginally different execution times though.

Various values of thread affinities set on Intel Xeon Phi cards as well as the impact of `MIC_USE_2MB_BUFFERS = 64k` are shown in Figure 7. It can be seen clearly that, while affinity settings do not impact performance much (note that 236 threads per Xeon Phi are used here), setting `MIC_USE_2MB_BUFFERS = 64k` brings a noticeable gain in performance.

Furthermore, we tested impact of dynamic adjustment of the second vector batch size as well as overlapping computations and communication. Best results of 10 runs for each configuration are reported in Fig. 8. These do have measurable impact albeit relatively small in the hybrid environment.

Then, the optimized code was tested firstly for host only and mic (Xeon Phi) only configurations in order to verify scalability and speed-ups. In these cases the number of computing threads was varied as shown in the X axis of the plots and set using the `num_threads(x)` clause to occupy the given number of logical processors on Xeons or a Xeon Phi. Figures 9 and 10 present execution times and speed-ups for these configurations for 10,000 vectors with the dimension size of 10,000. It can be noted that in work [25] parallel execution of the BiCGSTAB solver gave very similar speed-ups of up to around 90 for larger data sizes. In our case performance of the host CPU and the Xeon Phi were very similar. Peak double precision performance of Intel Xeon Phi 5000 series is given at just over 1 TFlop/s and is higher than theoretical performance of the CPUs used. However, real results may be different and may depend on how efficiently resources may be used by a given implementation of a specific problem.

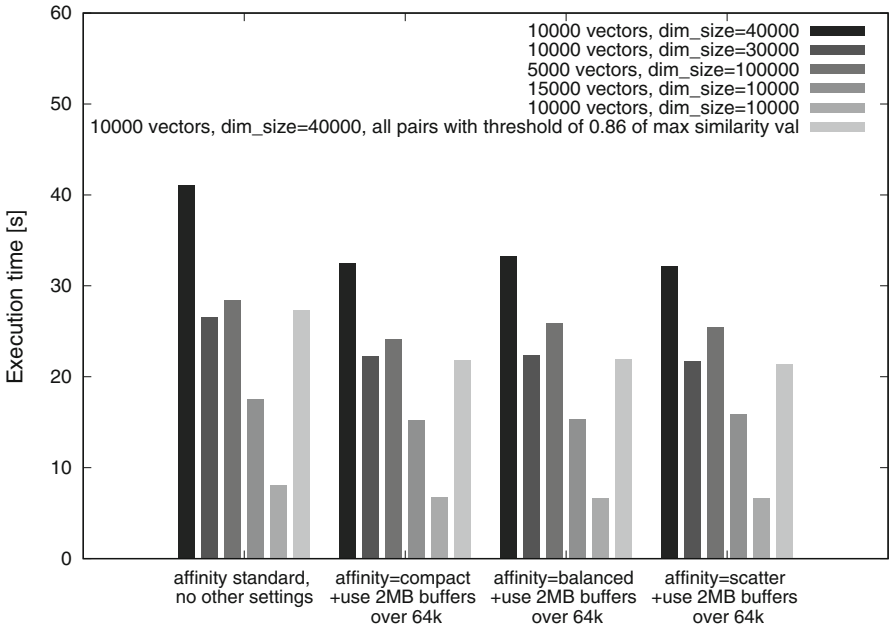

Fig. 7 Execution times for various settings including affinity on Intel Xeon Phi, host 2x CPU + 1x Xeon Phi offload, dynamic batch size `batch_size_threshold` = 78, second vector batch size = 100, first vector batch size = 2000

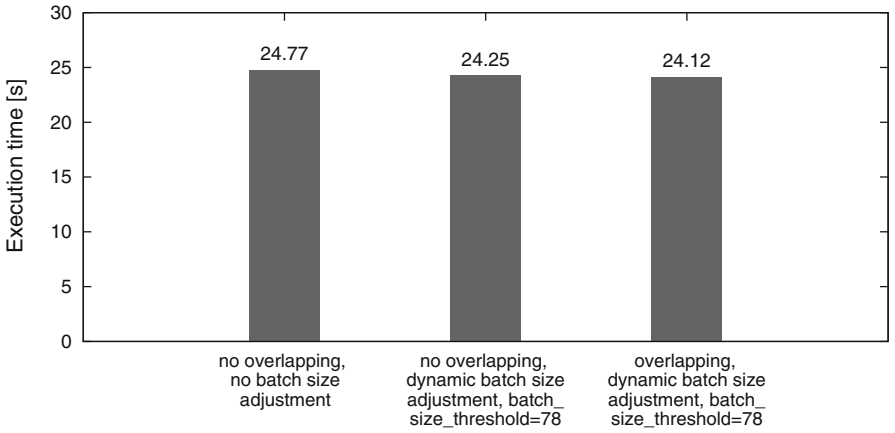

Fig. 8 Execution times with and without overlapping and thresholding, second vector batch size = 100, first vector batch size = 2000

Especially in the case of Xeon Phi, speed-up limitations and memory bottlenecks may affect performance. For instance, Fig. 10 shows that for the application tested parallel efficiency obtained for the host CPUs (speed-up of almost 26 for a total of 40 logical processors) compared to that for the Xeon Phi (speed-up of 95 for a total of 240 logical processors) contributes to similar execution times for the two. Power requirements for

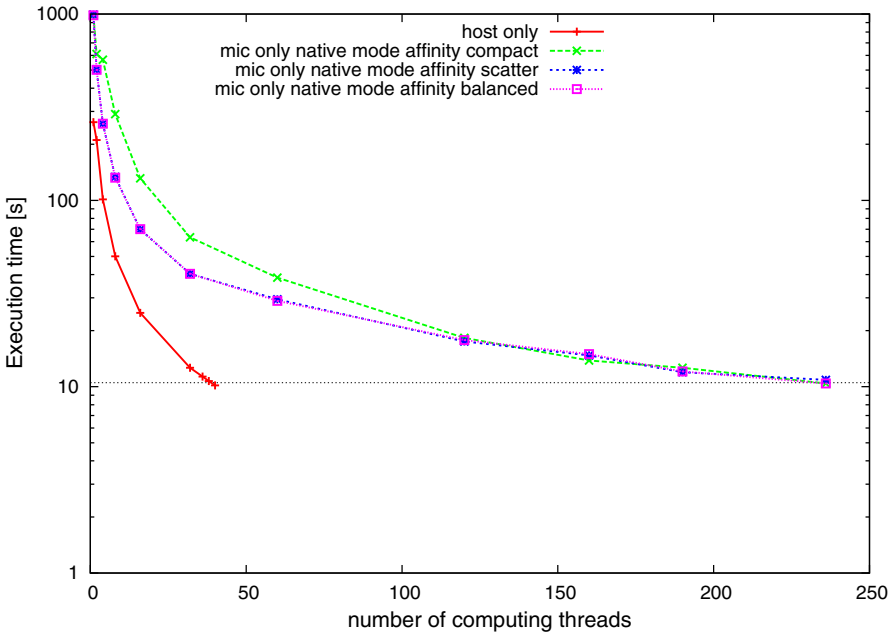

Fig. 9 Host only and mic only—execution time (s) versus number of computing threads, 10,000 vectors, `dim_size = 10,000`, dynamic batch adjustment, `batch_size_threshold = 78`

the CPUs are $2 \cdot 115$W which is slightly higher than given for Intel Xeon Phi 5110P at 225W which might be a relative benefit for the latter.

Interestingly, the improvements resulted in around 3 times faster code than the initial approach for 10,000 vectors and the dimension size of 10,000 shown in Sect. 3.1. This is supported by the results shown in Figs. 3 and 4 in which execution times for very small first vector data size are considerably larger. Then the other optimizations contributed to additional gains.

Finally, the optimized code was tested for various configurations on the host (2 CPUs), Xeon Phi in the native mode (without copying from/to the host) as well as in hybrid environments using 2 host CPUs and 1 or 2 Xeon Phi cards. In the latter configuration $20 + 20$ logical processors on the host and 480 logical processors on the Xeon Phi cards are available. From the results shown in Fig. 11 it can be seen that for various combinations of the number of vectors and dimension sizes, codes scale well with a very visible improvement in execution times between configurations.

Furthermore, we performed scalability tests across configurations for a modified code which finds pairs of vectors for which similarity exceeds a certain threshold of 0.86 of the maximum similarity for the previous runs. Specifically, after the tiled loop that for a given first vector goes through a second vector, there is a conditional cutting off further computations after the threshold has been achieved. Such version is more difficult to balance as various vectors may require various computational times. As shown in Fig. 12, compared to previous results, this version still scales well through the configurations up to the host and 2 Xeon Phi cards. Results for various affinities on the Xeon Phi are shown in Fig. 7.

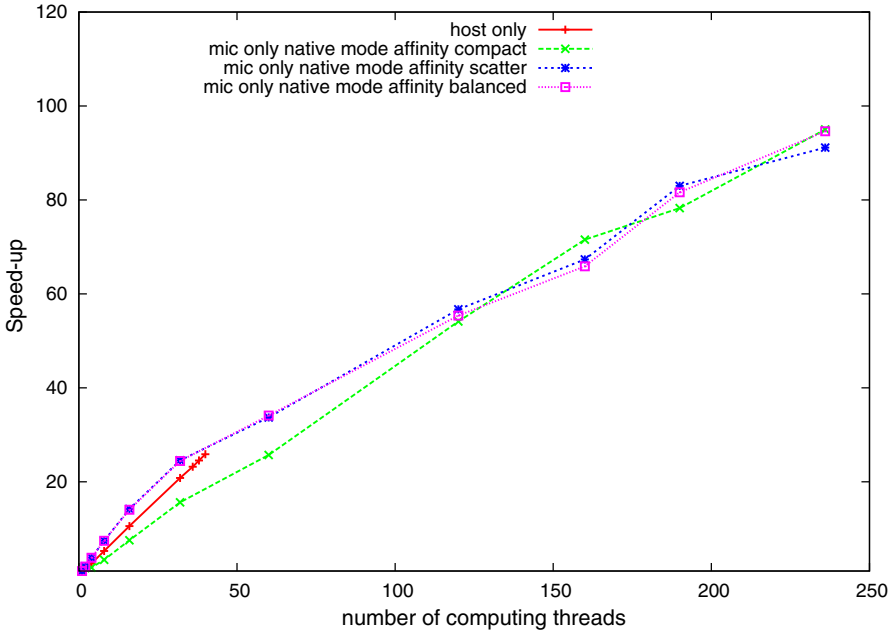

Fig. 10 host only and mic only—speed-up versus number of computing threads, 10,000 vectors, `dim_size = 10,000`, dynamic batch adjustment, `batch_size_threshold = 78`

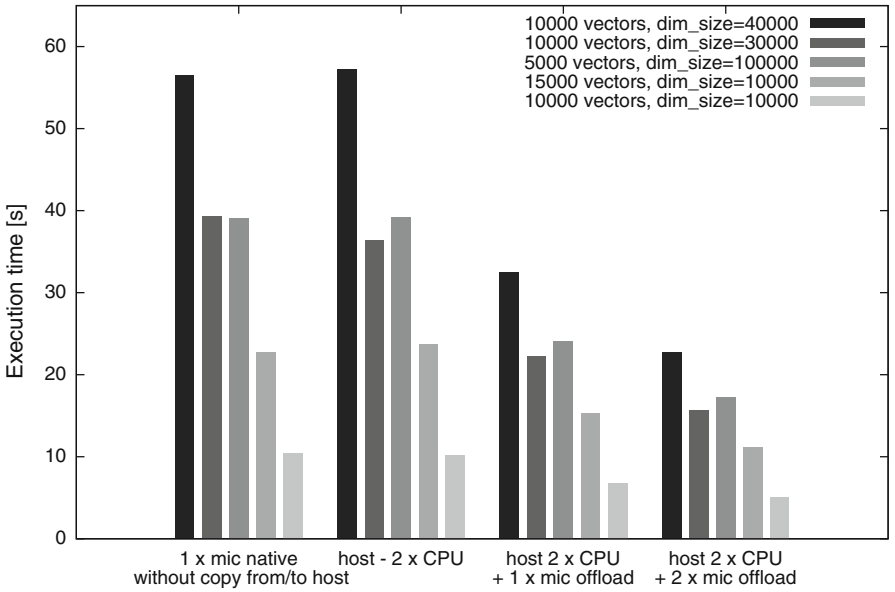

Fig. 11 Execution times for various configurations, dynamic batch size `batch_size_threshold = 78`, second vector batch size = 100, first vector batch size = 2000

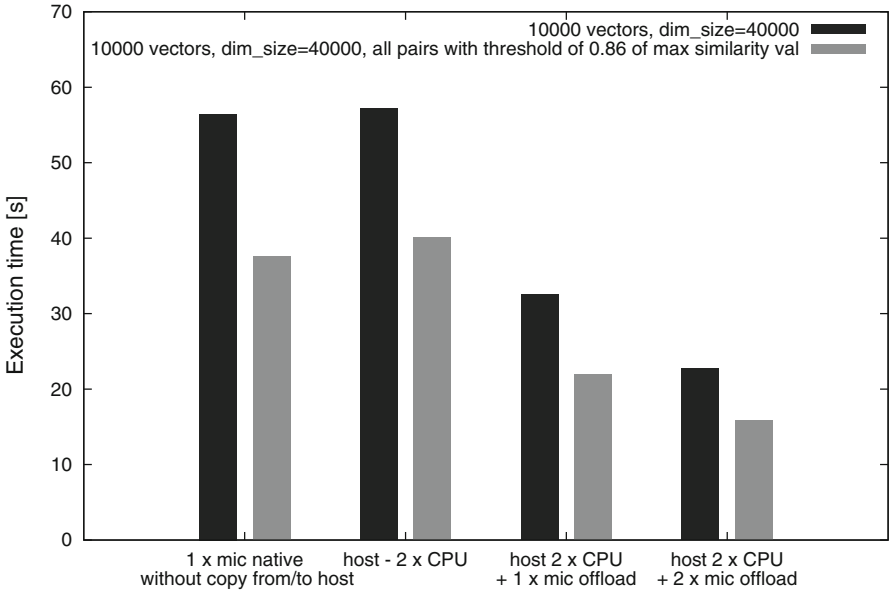

Fig. 12 Execution times for all pairs and all pairs exceeding threshold, dynamic batch size `batch_size_threshold` = 78, second vector batch size = 100, first vector batch size = 2000

4 Summary and Future Work

In the paper, parallelization of computing similarity measures between large vectors with OpenMP was benchmarked and optimized in a hybrid Intel Xeon/Xeon Phi system. Various optimizations were tested for performance in a real hybrid environment with multicore CPUs and Intel Xeon Phi coprocessors. Specifically, these included partitioning vectors into batches of size variable in time, testing for various thread affinities on Xeon Phi, overlapping computations and communication and Xeon Phi specific settings. Optimizations were presented for load balancing, loop tiling etc. Scalability of the code was demonstrated from Intel Xeon Phi only or host only environments to hybrid host + 1x Xeon Phi and host + 2x Xeon Phi cards. Future work includes tackling more advanced versions of the all pair search algorithms for which similarity values exceed a given threshold on the hybrid system. Additionally, we are going to incorporate and test Intel Xeon Phi coprocessors in KernelHive [33]—a system for parallelization of computations among heterogeneous clusters with CPUs and GPUs.

Acknowledgements The author wishes to thank Intel Technology Poland for provision of Xeon/Xeon Phi equipment and literature.

References

1. Alabduljalil, M.A., Tang, X., Yang, T.: Optimizing parallel algorithms for all pairs similarity search. In: Leonardi, S., Panconesi, A., Ferragina, P., Gionis, A. (eds.) Sixth ACM International Conference on Web Search and Data Mining, WSDM 2013, Rome, Italy, 4–8 February 2013, pp. 203–212. ACM (2013). doi:10.1145/2433396.2433422
2. Awekar, A., Samatova, N.F.: Fast matching for all pairs similarity search. In: IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, vol. 1, pp. 295–300 (2009). doi:10.1109/WI-IAT.2009.52
3. Barker, J., Bowden, J.: Manycore parallelism through openmp - high-performance scientific computing with xeon phi. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) OpenMP in the Era of Low Power Devices and Accelerators—9th International Workshop on OpenMP, IWOMP 2013, Canberra, ACT, Australia, 16–18 September 2013. Proceedings, Lecture Notes in Computer Science, vol. 8122, pp. 45–57. Springer (2013). doi:10.1007/978-3-642-40698-0_4
4. Barth, M., Byckling, M., Ilieva, N., Saarinen, S., Schliephake, M., Weinberg, V.: Best practice guide intel xeon phi. Partnership for Advanced Computing in Europe. http://www.prace-ri.eu/best-practice-guide-intel-xeon-phi-html/ (2014)
5. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: Proceedings of the 16th International Conference on World Wide Web, WWW '07, pp. 131–140. ACM, New York, NY, USA (2007). doi:10.1145/1242572.1242591
6. Cepeda, S.: Optimization and performance tuning for intel coprocessors, part 2: Understanding and using hardware events. Intel Developer Zone. https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding (2012)
7. 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, pp. 38–44 (2012)
8. Czarnul, P.: A model, design, and implementation of an efficient multithreaded workflow execution engine with data streaming, caching, and storage constraints. J. Supercomput. **63**(3), 919–945 (2012). doi:10.1007/s11227-012-0837-z
9. Czarnul, P.: Integration of Services into Workflow Applications. Chapman & Hall/CRC Computer and Information Science Series. Taylor & Francis. ISBN 978-1-49-870646-9. https://www.crcpress.com/Integration-of-Services-into-Workflow-Applications/Czarnul/p/book/9781498706469. (2015)
10. Czarnul, P.: Parallelization of divide-and-conquer applications on intel xeon phi with an openmp based framework. In: Swiatek, J., Borzemski, L., Grzech, A., Wilimowska, Z. (eds.) Information Systems Architecture and Technology: Proceedings of 36th International Conference on Information Systems Architecture and Technology—ISAT 2015—Part III, Karpacz, Poland, 20–22 September 2015, Advances in Intelligent Systems and Computing, vol. 431, pp. 99–111. Springer (2015). doi:10.1007/978-3-319-28564-1_9
11. Czarnul, P., Rosciszewski, P., Matuszek, M.R., Szymanski, J.: Simulation of parallel similarity measure computations for large data sets. In: 2nd IEEE International Conference on Cybernetics, CYBCONF 2015, Gdynia, Poland, 24–26 June 2015, pp. 472–477. IEEE (2015). doi:10.1109/CYBConf.2015.7175980
12. Davis, K.: Effective use of the intel compiler's offload features. Intel Developer Zone. https://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features. (2013)
13. De Francisci, G., Lucchese, C., Baraglia, R.: Scaling out all pairs similarity search with mapreduce. In: Large-Scale Distributed Systems for Information Retrieval, p. 27 (2010)
14. Fang, J., Sips, H., Zhang, L., Xu, C., Che, Y., Varbanescu, A.L.: Test-driving intel xeon phi. In: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ICPE '14, pp. 137–148. ACM, New York, NY, USA (2014). doi:10.1145/2568088.2576799
15. Green, R.W.: Openmp* thread affinity control. Intel Developer Zone. https://software.intel.com/en-us/articles/openmp-thread-affinity-control (2012)
16. Lam, H.T., Dung, D.V., Perego, R., Silvestri, F.: An incremental prefix filtering approach for the all pairs similarity search problem. In: Han, W., Srivastava, D., Yu, G., Yu, H., Huang, Z.H. (eds.) Advances in Web Technologies and Applications, Proceedings of the 12th Asia-Pacific Web Conference, APWeb 2010, Busan, Korea, 6–8 April 2010, pp. 188–194. IEEE Computer Society (2010). doi:10.1109/APWeb.2010.30

17. Leung, K.C., Eyers, D., Tang, X., Mills, S., Huang, Z.: Investigating large-scale feature matching using the intel xeon phi coprocessor. In: 2013 28th International Conference of Image and Vision Computing New Zealand (IVCNZ), pp. 148–153 (2013). doi:10.1109/IVCNZ.2013.6727007
18. Mabotuwana, T., Lee, M.C., Cohen-Solal, E.V.: An ontology-based similarity measure for biomedical data application to radiology reports. J. Biomed. Inform. **46**(5), 857–868 (2013). doi:10.1016/j.jbi.2013.06.013. http://www.sciencedirect.com/science/article/pii/S1532046413000889
19. Mackay, D.: Optimization and performance tuning for intel coprocessors - part 1: Optimization essentials. Intel Developer Zone. https://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-1-optimization (2012)
20. Masci, F.: Benchmarking the intel xeon phi coprocessor. http://web.ipac.caltech.edu/staff/fmasci/home/miscscience/MIC_benchmarking_2013.pdf (2013)
21. Memeti, S., Pllana, S.: Accelerating DNA sequence analysis using intel xeon phi. CoRR **abs/1506.08612**, arxiv:1506.08612 (2015)
22. Michaela, M., Byckling, M., Ilieva, N., Saarinen, S., Schliephake, M., Weinberg, V.: Best practice guide intel xeon phi v1.1. PRACE, 7 Capacities. http://www.prace-project.eu/IMG/pdf/Best-Practice-Guide-Intel-Xeon-Phi.pdf (2014)
23. Movchan, A., Zymbler, M.: Time series subsequence similarity search under dynamic time warping distance on the intel many-core accelerators. In: Amato, G., Connor, R., Falchi, F., Gennaro, C. (eds.) Similarity Search and Applications, Lecture Notes in Computer Science, vol. 9371, pp. 295–306. Springer International Publishing (2015). doi:10.1007/978-3-319-25087-8_28
24. Pantel, P., Crestan, E., Borkovsky, A., Popescu, A.M., Vyas, V.: Web-scale distributional similarity and entity set expansion. In: Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing, EMNLP '09, vol. 2, pp. 938–947. Association for Computational Linguistics, Stroudsburg, PA, USA. http://dl.acm.org/citation.cfm?id=1699571.1699635 (2009)
25. Petkova, P., Grancharov, D., Markov, S., Georgiev, G., Lilkova, E., Ilieva, N., Litov, L.: Massively parallel poisson equation solver for hybrid intel xeon xeon phi hpc systems. PRACE, white paper. http://www.prace-ri.eu/IMG/pdf/wp143.pdf
26. Potluri, S., Hamidouche, K., Bureddy, D., Panda, D.: Mvapich2-mic: A high performance mpi library for xeon phi clusters with infiniband. In: Extreme Scaling Workshop (XSW), pp. 25–32 (2013). doi:10.1109/XSW.2013.8
27. Potluri, S., Tomko, K., Bureddy, D., Panda, D.K.: Intra-mic mpi communication using mvapich2: Early experience. In: TACC-Intel Highly Parallel Computing Symposium. Austin, TX, USA. https://www.tacc.utexas.edu/documents/13601/7f745047-5b63-44ac-aa7b-fb32cf0c4c05 (2012)
28. Potluri, S., Venkatesh, A., Bureddy, D., Kandalla, K.C., Panda, D.K.: Efficient intra-node communication on intel-mic clusters. In: 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013, Delft, Netherlands, 13–16 May 2013, pp. 128–135. IEEE Computer Society (2013). doi:10.1109/CCGrid.2013.86
29. Pushpa, C., Girish, S., Nitin, S., Thriveni, J., Venugopal, K., Patnaik, L.: Computing semantic similarity measure between words using web search engine. In: Wyld, D.C., Nagamalai, D., Meghanathan, N. (eds.) Third International Conference on Computer Science, Engineering and Applications (ICCSEA 2013), pp. 135–142. Delhi, India (2013). ISBN: 978-1-921987-13-7. doi:10.5121/csit.2013.3514
30. Reinders, J.: An overview of programming for intel xeon processors and intel xeon phi coprocessors. Intel Developer Zone. https://software.intel.com/en-us/articles/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi-coprocessors (2012)
31. Rodriguez-Serrano, J.A., Perronnin, F., Llados, J., Sanchez, G.: A similarity measure between vector sequences with application to handwritten word image retrieval. In: IEEE Conference on Computer Vision and Pattern Recognition, 2009. CVPR 2009, pp. 1722–1729 (2009). doi:10.1109/CVPR.2009.5206783
32. Rosales, C.: Porting to the intel xeon phi: opportunities and challenges. In: Extreme Scaling Workshop (XSCALE13) (2013)
33. Rosciszewski, P., Czarnul, P., Lewandowski, R., Schally-Kacprzak, M.: Kernelhive: a new workflow-based framework for multilevel high performance computing using clusters and workstations with CPUs and GPUs. Concurr. Comput. Pract. Exp. **28**(9), 2586–2607 (2016). doi:10.1002/cpe.3719
34. Sahami, M., Heilman, T.D.: A web-based kernel function for measuring the similarity of short text snippets. In: Proceedings of the 15th International Conference on World Wide Web, WWW '06, pp. 377–386. ACM, New York, NY, USA (2006). doi:10.1145/1135777.1135834

35. Saule, E., Kaya, K., Çatalyürek, Ü.V.: Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. CoRR **abs/1302.1078**, arxiv:1302.1078 (2013)
36. Szymanski, J.: Mining relations between wikipedia categories. In: Networked Digital Technologies—Second International Conference, NDT 2010, Prague, Czech Republic, July 7–9, 2010. Proceedings, Part II, pp. 248–255 (2010)
37. Szymanski, J.: Comparative analysis of text representation methods using classification. Cybern. Syst. **45**(2), 180–199 (2014)
38. Vladimirov, A., Asai, R., Karpusenko, V.: Parallel Programming and Optimization with Intel Xeon Phi Coprocessors. Colfax International (2015). ISBN 978-0-9885234-0-1
39. Zadeh, R.B., Goel, A.: Dimension independent similarity computation. J. Mach. Learn. Res. **14**(1), 1605–1626 (2013). http://dl.acm.org/citation.cfm?id=2567715