Evaluation of Vectorization Potential of Graph500 on Intel's Xeon Phi

Milan Stanic*, Oscar Palomar*‡, Ivan Ratkovic*, Milovan Duric*, Osman Unsal*, Adrian Cristal*‡§, Mateo Valero*‡
* Computer Sciences, Barcelona Supercomputing Center, Barcelona, Spain
‡ Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, Barcelona, Spain
§ Artificial Intelligence Research Institute (IIIA), Centro Superior de Investigaciones Científicas (CSIC), Barcelona, Spain
Email: {first}.{last}@bsc.es

***Abstract*—Graph500 is a data intensive application for high performance computing and it is an increasingly important workload because graphs are a core part of most analytic applications. So far there is no work that examines if Graph500 is suitable for vectorization mostly due a lack of vector memory instructions for irregular memory accesses. The Xeon Phi is a massively parallel processor recently released by Intel with new features such as a wide 512-bit vector unit and vector scatter/gather instructions. Thus, the Xeon Phi allows for more efficient parallelization of Graph500 that is combined with vectorization.**

In this paper we vectorize Graph500 and analyze the impact of vectorization and prefetching on the Xeon Phi. We also show that the combination of parallelization, vectorization and prefetching yields a speedup of 27% over a parallel version with prefetching that does not leverage the vector capabilities of the Xeon Phi.

***Keywords*—Multi-Core Architectures and Support, Data Intensive Supercomputing, Large Scale Scientific Computing**

I. Introduction

The Graph500 [1] benchmark is a data intensive, high performance computing application. It complements TOP500, the performance evaluation metric used to rank supercomputers, and targets five graph-related business areas: Cybersecurity, Medical Informatics, Data Enrichment, Social Networks, and Symbolic Networks. Graph500 is used for performance ranking and has attracted attention in recent years. The Graph500 list ranks the first 500 supercomputers with highest performance running the Graph500 benchmark.

Graph500 implements three kernels: concurrent search, optimization (single source shortest path), and edge-oriented (maximal independent set). They access a single data structure representing a weighted, undirected graph. The main kernel of Graph500, Breadth First Search (BFS), contains a lot of irregular memory accesses. Most of SIMD ISAs do not provide support for indexed memory accesses and this lack has prevented vectorization of BFS.

The Xeon Phi is a recent massively parallel x86 microprocessor designed by Intel and is based on the Larrabee GPU [2]. Its new features, such as scatter/gather instructions, satisfy missing requirements that prevent vectorization of Graph500. It has a large number of cores and each core contains a wide 512-bit vector processing unit. It delivers substantial performance and has been designed for power efficiency when executing highly parallel applications that can benefit from parallelization and vectorization. These characteristics make the Xeon Phi an attractive processor for building supercomputers. The most powerful supercomputer according to the TOP500 list from November 2013[a], Tianhe-2 (MilkyWay-2), is built using Xeon Phi co-processors.

Currently there are two supercomputers on the Graph500 list that are built using the Xeon Phi. However, in their implementation of the Graph500 they just exploit the parallelization features of the Xeon Phi, following the work done by Saule et al. [3] and ignore the vector features offered by the Xeon Phi that allows for more efficient parallelization.

The major contributions of this paper are a vectorization of Graph500, an analysis of the peformance of vectorization for Graph500 and a study of the impact of prefetching on the performance of the Xeon Phi. The Xeon Phi cores are in-order, which makes it very sensitive to cache misses. We also provide two observations for Xeon Phi users: the initial results for vectorization can be misleading and prefetching is of maximum importance if data is not cache resident. We also prove that the combination of vectorization and parallelization is beneficial for workloads running on the Xeon Phi. All this allows for achieving higher performance than just using parallelization.

II. Parallel BFS Implementations

Breadth First Search (BFS) is the main kernel of Graph500. BFS begins at a random node called start node and inspects all its neighboring nodes. Then, for each of those neighbor nodes in turn, it inspects their neighbor nodes which were yet unvisited, and so on. In Graph500, there is freedom to change the algorithm, the implementation and the data structures used. In order to be able to compare the performance of BFS implementations across a variety of architectures, programming

The research leading to these results has received funding from the European Research Council under the European Unions 7th FP (FP/2007-2013) / ERC GA n. 321253. It has been partially funded by the Spanish Government (TIN2012-34557).

[a]http://top500.org/lists/2013/11/

978-1-4799-5313-4/14/$31.00 ©2014 IEEE

models, languages and frameworks, the performance metric TEPS (traversed edges per second) is used. It is defined as the ratio of the number of edges in the input graph to the execution time. In this section we describe the current default implementation that we have used in our experiments, as well as an overview of recently proposed alternative BFS implementations.

A. Current BFS Implementation

The default parallel implementation of BFS is queue-based with local per-thread next-level queues. Fig. 1 shows the pseudo-code of that implementation. It is a level synchronous BFS algorithm with two optimization techniques borrowed from [4]: *'test and test-and-set'* operation and use of local next-level queues. It means that the search of unvisited neighbor nodes (*neighbors*) is done in parallel (*partition* per thread) and there is synchronization before the algorithm starts to search their neighbors. The algorithm manages two sets of nodes: the visited (*parent*) set and the next-nodes (*next*) set. BFS starts searching from the *start* node. In each iteration, the algorithm visits all the nodes in the *next* set in parallel and for each node, the *'test and test-and-set'* operation (lines 8-9) is used to check if the neighbors have not been visited already (line 8). If this is true, the *parent* node is assigned to the parent set (line 9). Unvisited nodes are first stored in the per-thread local queue (*local_queue*) until they are bulk inserted into the global queue (lines 12, 20). Fig. 2 shows an example of the graph traversed using this implementation. The *start* node is A. In the first iteration, all its neighbors are visited (gray nodes - B, C, D) and added to the *next* set. In the next iteration, the neighbors of gray nodes are visited and only unvisited nodes (yellow nodes - E, F, G, H) are added to the *next* set.

```
1.  function breadth-first-search(vertices, source)
2.    next ← {source}
3.    parents ← [-1,-1,...,-1]
4.    parents[source] = 0
5.    while next ≠ {} do
6.      for v ∈ next.partition[tid] do
7.        for n ∈ neighbors[v] do
8.          if parents[n] = -1 then
9.            if parents[n].atomicSet(v) = 1 then
10.             local_queue[tid] <- n
11.             if local_queue[tid].isFull() then
12.               next.safeBulkPush(local_queue[tid])
13.               local_queue[tid] ← {}
14.             end if
15.           end if
16.         end if
17.       end for
18.     end for
19.     if local_queue[tid].notEmpty() then
20.       next.safeBulkPush(local_queue[tid])
21.       local_queue[tid] ← {}
22.     end if
23.   end while
24.   return parents
```

Figure 1. Pseudo-code for the Graph500 BFS algorithm.

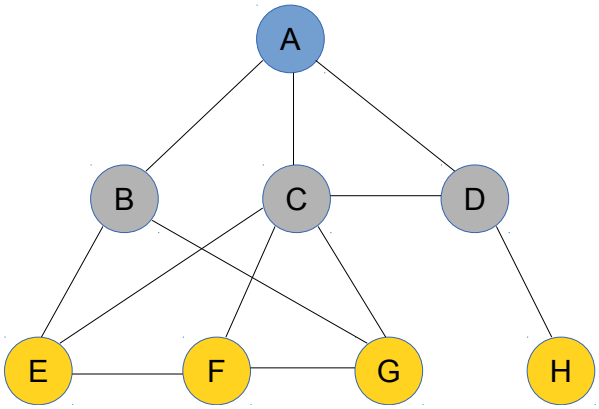

Figure 2. An example of graph traversed by the BFS algorithm.

B. Other BFS Implementations

In this subsection we describe alternative implementations of BFS.

1) Bitmap: A bitmap is used to compactly represent the visited set as an optimization to the default algorithm that has been discussed above. The main benefit is to reduce the size of the structure, thus reducing cache misses. Since this structure is the most frequently accessed data in the algorithm, the use of a bitmap in shared-memory machines with large last-level caches is effective.

2) Read-based: Read-based BFS is proposed in [5] and it also keeps the visited set as a bitmap. In contrast to the current BFS implementation, it keeps a node's level in the *next* set. During the search, if a node belongs to the current level, it means that its neighbors should be searched in the current iteration. For example, in Fig. 2, the blue node will have level 0, the gray nodes will have level 1 and the yellow nodes will have level 2. The advantages of this method are the elimination of queue overhead (it removes atomic instuctions and also saves cache and memory bandwidth) and the replacement of some indexed memory accesses with sequential memory accesses.

3) Bottom-Up: Bottom-up BFS [6] proposes searching in the opposite direction compared to previous methods. Instead of each node in the *next* set attempting to become the parent of *all* of its neighbors, each unvisited node attempts to find *any* parent among its neighbors. A neighbor can be a parent if the neighbor is a member of the *next* set. For example, in the first iteration for the graph in Fig. 2 only node A is in the *next* set. It means that only the gray nodes (B, C, D) can find a valid parent in the first iteration. This approach is more efficient when the *next* set is large. It reduces a lot the total number of edges examined.

4) Hybrid: Hybrid BFS [6] is the state-of-the-art BFS implementation and it is part of pre-released version of

Graph500 benchmark. It combines top-down and bottom-up approches because they are complementary. It uses the top-down approach when the *next* set is small and the bottom-up approach when the *next* set is large.

III. Vectorization of Graph500

We vectorized by hand two versions of BFS: sequential and parallel OpenMP versions based on the current implementation. Both versions are vectorized practically in the same way using intrinsics. The part of code that is actually vectorized is the loop where a node searches for its neighbors (lines 7-17 in Fig. 1). All neighbors of the node are loaded using vector load instructions (line 7). Line 8, where the algorithm checks if a neighbor is not visited, is vectorized using an indexed vector load from the *parent* set and comparing loaded values with -1. The neighbors that are loaded in the previous step are used as indices. The two previous steps are exactly the same for the sequential and parallel versions. The next step, where the *parent* and *next* sets have to be updated, is different. In the parallel version, unvisited neighbors are stored to a temporal array using the vector packstore instruction to store selected elements to consecutive memory locations using a vector mask. Then scalar atomic instructions are used to ensure that all the threads correctly update the unvisited nodes to the *parent* and *next* sets. This is done with scalar code. In the sequential version, the *parent* set is updated using an indexed vector store, while the *next* set is updated using the vector packstore instruction.

Each core in Xeon Phi [7] uses a short in-order pipeline and is capable of supporting 4 threads in hardware. It contains a vector processing unit (VPU) that implements a novel 512-bit SIMD instruction set. A mask register was added to allow predicated execution and it also supports gather and scatter instructions. The VPU can execute 8 operations per cycle with 64-bit data or 16 operations per cycle with 32-bit data.

As mentioned above, the Xeon Phi operates on 512-bit vectors. Our implementation of Graph500 uses 32-bit data, thus each Xeon Phi vector instruction operates on 16 elements at a time. This is known as the vector length. If the number of iterations in a loop is longer than the vector length, the strip-mining technique [8] is applied during the process of vectorization. Strip-mining is a technique that allows for operating on stripes of data that has length less than or equal to the vector length. If the number of iterations is not a multiple of the vector length there is a remaining part, called epilogue. The epilogue can be vectorized in the Xeon Phi using vector masked instructions or it can be left as scalar code. Please note that when the number of iterations is smaller than the vector length, the whole loop is considered the epilogue. Xeon Phi has support for gather vector memory instruction that has an index vector with elements of a maximum of 32 bits. Since elements that are loaded from memory and used as index vector, we are forced to use 32-bit elements for graph representation instead of 64-bit elements like the original implementation of Graph500.

IV. Methodology

This section presents our experimental setup. We measured the performance of our various vectorized BFS implementations and we compared them against the original Graph500 implementations. *Edgefactor* and *SCALE* are the main input parameters of the graph generator. *SCALE* is the log_2 of the number of nodes in the graph. This parameter determines the graph size and consequently the size of data structures needed to store it. *Edgefactor* is the ratio of the graph's edge count to its node count (i.e., half the average degree of a node in the graph). For higher *edgefactor*, the average size of the adjacency list is also higher. In particular, this parameter has a direct effect on the length of the vectorized loop and, consequently, on the amount of vectorized code when the epilogue is executed with scalar instructions.

We use an *edgefactor* of 8 and a *SCALE* of 23 in our experiments unless specified otherwise. This is the highest value of SCALE that can be used in our system. The graphs are stored in Compressed Sparse Row (CSR) format. It merges the adjacency lists of all nodes into a single array, with the initial location of each node's adjacency list stored in a separate array. We used Intel's compiler version 13.0.1 with the -O3 optimization level.

We run our experiments on a compute node that contains two Intel Xeon CPU E5-2670 @ 2.60GHz processors (8 cores/processor), 64 GB of RAM memory and two Intel Xeon Phi 5110P (60 cores and four hardware threads per core). In our experiments we use a single Xeon Phi processor.

As mentioned above, Traversed Edges per Second (TEPS) is used to benchmark performance and it represents the ratio of the number of edges in the input graph to the runtime. Graph500 performs 64 searches (starting from a random node) per run. We use the harmonic mean of TEPS of all 64 searches (*harm TEPS*).

For the sake of clarity, we label the experiments in the following way: the name of experiment has three parts; the first part indicates whether it is vectorized (*vect*) or sequential (*seq*) code; the second part tells if sequential prefetching is applied (*spf*), vectorized (*vpf*), both are combined (*vspf*) or there is no prefetching (*npf*); this is followed by a number that indicates if it is a single-thread execution (1) or multi-thread execution (n being the number of threads used). For example, *seq_npf_1* is a sequential single-thread execution without prefetching. Additional information is appended to the name in some cases.

Two modes of execution are used to run experiments on the Xeon Phi: "offload" and "native" mode. In "offload" mode, an application runs on the host machine and the parts of the code that are specified to be executed on the Xeon Phi are "offloaded" during execution to the co-processor. Data has to be copied to the co-processor memory before and after "offloading". In our experiments in "offload" mode, the whole BFS kernel is executed on the Xeon Phi. In "native" mode,

the application is executed completely on the Xeon Phi.

V. Experimental Results

In the first part of our experiments, we evaluated our vectorized implementation of BFS against the sequential implementation for single-thread execution. In the second part, we focused on parallel OpenMP implementations. Our main goal was to check if there is any benefit of applying vectorization on Graph500 using the Xeon Phi.

A. Single-Thread Results

For the single-thread execution, we focused on three types of experiments. We compared results when only vectorization is applied, then we applied prefetching and finally we measured hardware counters using the PAPI library to better understand the results that we obtained in the previous experiments.

1) Manual vectorization and memory alignment: We compared our sequential implementations of Graph500 against the original sequential Compressed Sparse Row (CSR) implementation from the benchmark suite.

The CSR format is used to store the graph. It merges the adjacency lists of all nodes into a single array, and the offset to the initial location of each node's adjacency list is stored in a separate array. By storing the list consecutively, the lists are typically unaligned at the 512-bit boundary. This implies that two vector memory instructions are needed to load/store a 512-bit vector with a node's adjacency list in the Xeon Phi. We have implemented a padded version of this structure to enforce 512-bit aligned accesses, thus reducing the access to a single vector memory instruction.

Fig. 3 shows the speedup over the original implementation for different single-threaded versions of Graph500: sequential original CSR (*seq_npf_1*), a vectorized version with aligned accesses (*vect_npf_1+alig*), a vectorized version with unaligned accesses to the original (non-padded) data structure (*vect_npf_1+unalig*), and *vect_npf_1+unalig+epil*, which is the same but with a vectorized epilogue using masked operations. The epilogue is executed in scalar fashion in *vect_npf_1+alig* and *vect_npf_1+unalig* versions. The speedups are computed for *harmonic TEPS* over *seq_npf_1*.

In Fig. 3, one may notice that vectorized implementations yield small speedups, ranging from 3% for unaligned+epilogue up to 7% for unaligned. Another interesting point is that we did not get better results when aligned memory accesses are enforced. The main reason is the increased cache miss rate due to the use of a larger structure to store the input graph. We also run experiments using the "native" mode of execution and we almost got the same results for the sequential and aligned versions. We experimented with an *edgefactor* of 16 to increase the lenght of the vectorized loop. However, we saw only a 3.5% speedup for the unaligned version. Subsequent experiments assume vectorized code with unaligned memory accesses and scalar epilogue.

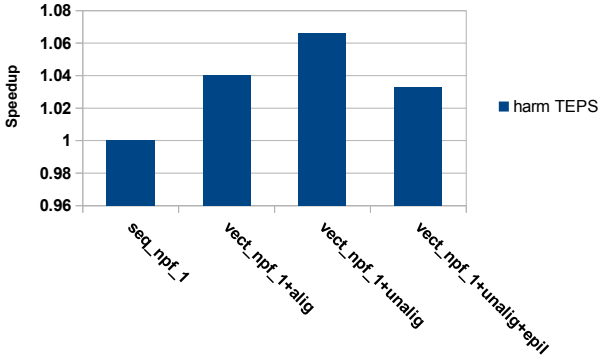

Figure 3. Results for different implementations using single-thread run.

2) Vectorization Prefetching vs Sequential Prefetching: Since the results of vectorization were not satisfying, we decided to apply prefetching to the original sequential code and our implementation. The Intel compiler has good automatic prefetching capabilities but in our experiments it does not provide significant speedup. The main reason is the abundance of indexed memory accesses, which the compiler does not prefetch automatically. Therefore, we decided to manually insert prefetching intrinsics. We used scalar prefetch instructions for the sequential version and vector gather prefetch instructions for our vectorized implementation with unaligned accesses to memory. The results are presented only for the "native" mode but the results for "offload" are very similar.

For sequential prefetching, we experimented with six different prefetch distances. Fig. 4 (a) shows results for these experiments. The y-axis presents the speedups over the original implementation (*seq_npf_1*) and the x-axis shows experiments with different prefetch distances. Scalar prefetching allows for significant speedups. A prefetch distance of two provides the maximum speedup, 3.01x.

We implemented six variants for vectorized prefetching. Prefetching is applied to the vectorized *for* loop in line 7 of Fig. 1. In the first variant (*n_iter+n_node*), we prefetched the next iteration of the vectorized loop (neighbor's nodes) for the current node in the *next* set or the first iteration of the next node in the *next* set if we are in the last iteration of the vectorized *for* loop. In the second variant (*2_its*), we prefetched the next two iterations if possible. In order to implement prefetching, we need to load the index vector for the gather vector prefetch instruction. We separated the vector load and the vector gather prefetch instructions in the third (*splited_ld_gth*) and fourth (*splited_ld_gth_II*) variants and placed them in different places of the code. The fifth variant (*2nd_iter_L2*) is similar to the second one, the only difference is that the second iteration is prefetched to the L2 cache. The last variant (*vect_vspf_1*) combines the first variant for vectorized code and sequential prefetching for the epilogue. As explained above, the epilogue is implemented with scalar instructions.

Results for prefetching in the vectorized code are presented in Fig. 4 (b). Speedups are computed over the vectorized

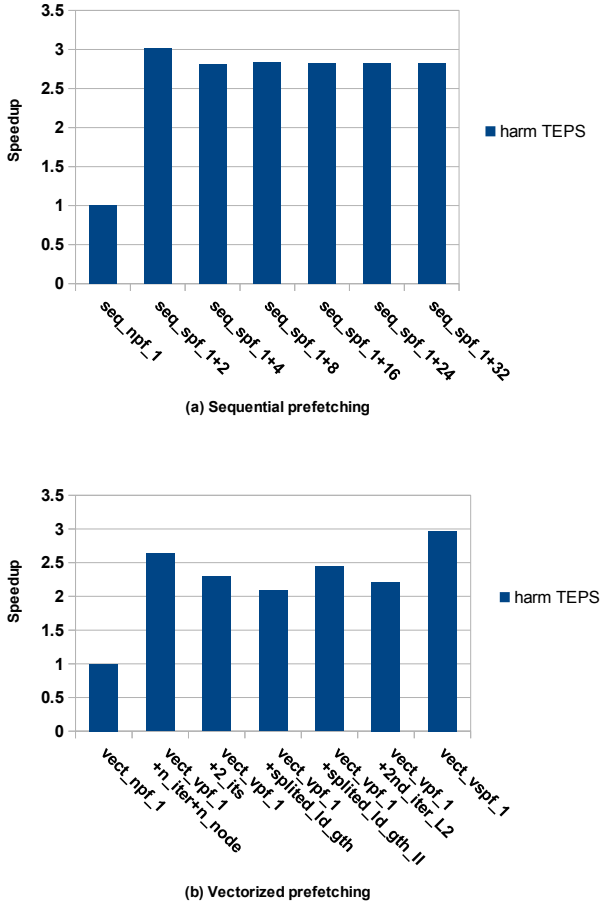

Figure 4. Results with prefetching for single-thread run.

implementation without prefetching. Prefetch again provides significant speedups, up to 2.99x for the sixth variant that combines sequential and vector prefetching. It is interesting to notice that the best results are obtained when sequential prefetching is included. The speedup is 1.07x over the best vector-only prefetching variant (first variant). The combination of both prefetching (*vspf*) schemes in the vectorized version provides a speedup of 1.04x over the best sequential variant. The next section aims to explain these results by analyzing measurements obtained with hardware counters.

3) PAPI Profiling Results: For further analysis of the results that we obtained using vectorization and vector prefetching, we used PAPI [9] hardware counters. We run experiments for five different implementations: sequential, vectorized, sequential with prefetching, vectorized with vector prefetching and vectorized version with combined sequential and vector prefetching. The experiments were run using the "native" mode of execution. Table I summarizes the results that we collected with PAPI counters, while aggregating the results for 64 BFS calls. The first column lists the different implementations of BFS. The second column presents the percentage of vector instructions executed in the benchmark. Third and fourth columns are L1 and L2 cache miss rates, respectively. The fifth column is the number of vector instructions executed. Finally, the last column is the number of execution cycles.

Table I shows that the results for the original and vectorized versions are similar. Provided that the epilogue is executed with scalar instructions, one may guess that there is a small amount of vector instructions and there are no benefits of vectorization, but it is actually not the case. 42.71% of all executed instructions are vector and the main reason why we observe similar performance for sequential and vectorized versions is cache behaviour. The use of gather/scatter instructions increases the L1 cache miss rate and therefore reduces the benefit of vectorization.

As can be expected, the use of prefetch instructions (rows three and four) decreases a lot L1 and L2 cache miss rates and this is the main reason for the significant speedups shown in Fig. 4. However, another interesing point is that L1 and L2 cache miss rates for the vectorized version with prefetching are again higher than L1 and L2 cache miss rates for sequential prefetching. This is the reason why experiments with sequential prefetching achieve higher speedups in Fig. 4. L2 cache miss rate is always higher than L1 cache miss rate because the data set is not L2 cache resident.

The combination of vector and scalar prefetching provides better cache utilization, allowing the vectorized version to outperform the best sequential version by 4%.

B. Results for OpenMP Implementation

The Xeon Phi provides the best performance when both, parallelization and vectorization, are applied together. In this section we explore the effect of vectorization on the CSR version of Graph500 parallelized using OpenMP.

1) Manual vs. Automatic Vectorization: Fig. 5 presents results for three different parallel versions of Graph500. *seq_npf_n* is the original parallel version, *seq_npf_n-vect* is the same but with auto-vectorization disabled and *vect_npf_n* is our vectorized version of parallel Graph500. All three version are compiled using the -O3 optimization level without prefetching. The y-axis presents measured performance in harmonic TEPS and the x-axis represents the number of threads used (*n*).

The results show that vectorization does not have significant impact on the measured performance. The difference between *seq_npf_n-vect* and *seq_npf_n* is also negligible because the compiler is not able to vectorize any part of BFS code. While vectorization seems inefficient, the increased number of threads provides better performance. For example, the parallel vectorized version with 40 threads in "offload" mode is 14 times faster than the best single-thread implementation. The results scale well for "native" mode of execution while they saturate for "offload" mode if we use more than 160 threads. The main reason for better results and scalability in "native" mode is substantial data transfer overhead before and after "offloading".

TABLE I
OBTAINED RESULTS USING HARDWARE COUNTERS.

Implementation	% of vector instructions	L1 cache miss rate	L2 cache miss rate	# of instructions	# of cycles
Sequential	0.07	44.98	89.34	11.3×10^8	47.8×10^9
Vectorized	42.71	66.78	89.99	6.36×10^8	47.5×10^9
Seq_prefetching	0.01	1.96	15.87	30.1×10^8	12.0×10^9
Vect_prefetching	53.78	12.68	47.14	9.83×10^8	15.9×10^9
Seq_vect_prefetching	41.55	4.54	19.34	12.7×10^8	11.8×10^9

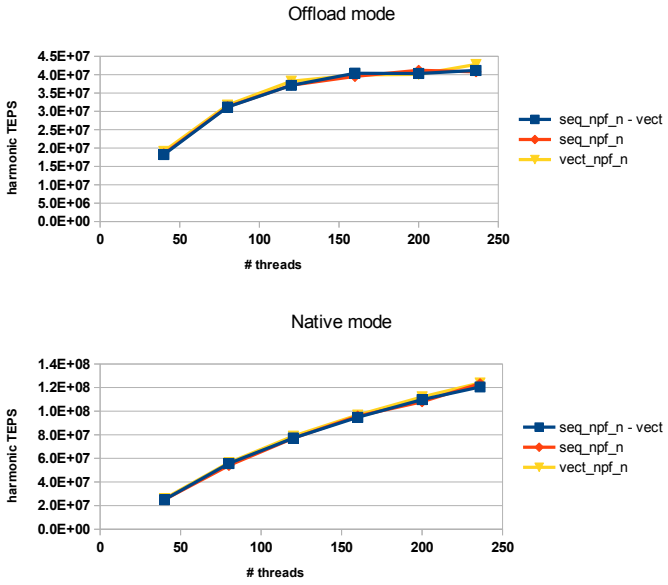

Figure 5. Results for hand-written vectorization, auto-vectorization and no vectorization.

2) Vectorization Prefetching vs Sequential Prefetching: We further explore the effects of prefetching on the parallel version of Graph500. We applied prefetching on the original OpenMP implementation using sequential prefetch instructions while we used vector scatter/gather prefetch instructions for our implemenation. We experimented with several different prefetch distances for sequential prefetching as well as six different variants for the vectorized version. These are the same prefetching schemes presented in subsection V-A. All experiments were performed using 160 threads.

Fig. 6 shows the speedup for the original OpenMP implementation with sequential prefetch instructions that use different prefetch distances. The speedup is computed over the original OpenMP implementation. Prefetching increases the performance of Graph500 for all prefetch distances in both modes, "offload" and "native". For example, the highest speedups are 2.07x and 2.36x for "offload" and "native" modes respectively, for a prefetch distance of eight.

The results for vectorized prefetching are presented in Fig. 7. The y-axis presents speedups computed over our vectorized OpenMP implementation and the x-axis shows our different prefetching implementations. Again prefetching increases performance significantly and the best results are obtained for the combined vector and sequential prefetching scheme.

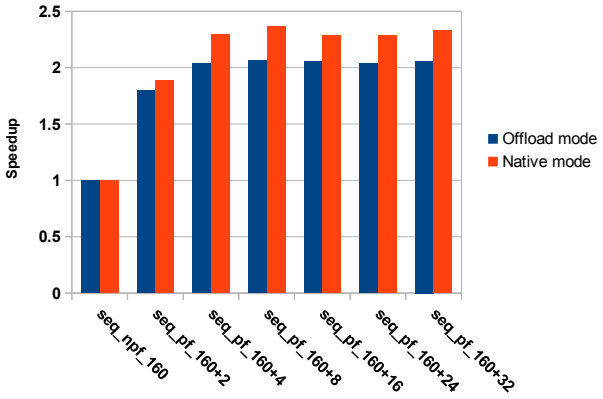

Figure 6. Results for Omp version with sequential prefetching.

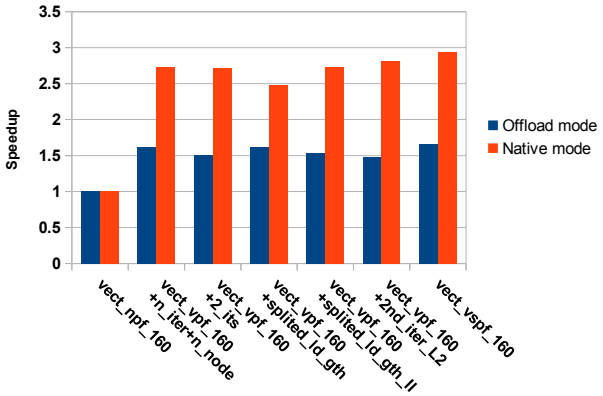

Figure 7. Results for vectorized version with gather/scatter prefetching.

Speedups are 1.66x and 2.94x for "offload" and "native" modes, respectively.

Our implementation outperforms the best implementation from Fig. 6 providing 10% and 27% higher harmonic TEPS in "offload" and "native" modes respectively. The Xeon Phi provides the best performance when both, parallelization and vectorization are applied and Graph500 clearly can benefit from it.

3) Scalability: Fig. 8 shows the results for different number of threads for the OpenMP implementation with sequential prefetching and the vectorized version with combined vector-sequential prefetching. It can be seen that "native" mode performance scales better than the results for the "offload" mode due to substantial data transfer overhead in the "offload" mode. In the "offload" mode, the harmonic TEPS seem to

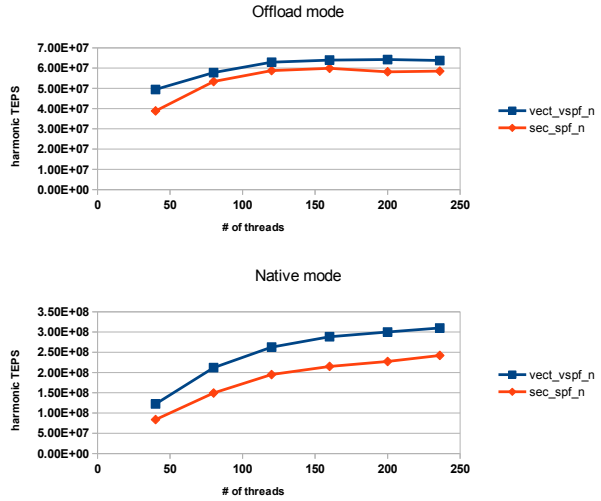

Figure 8. Results for prefetching using different number of threads.

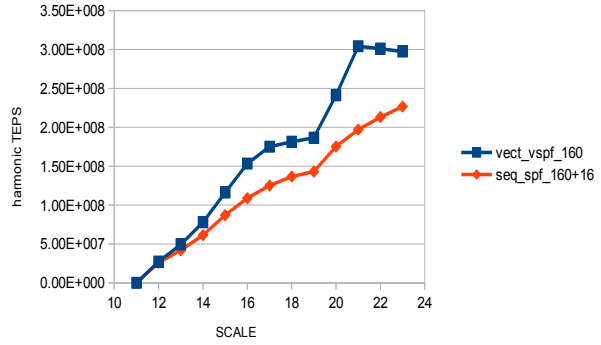

Figure 9. Impact of *SCALE* on performance in native mode.

saturate after 120 threads, while performance still grows, albeit slower, for "native" execution. It is also noticeable that the vectorized version consistently outperforms the sequential version, for any number of threads and both execution modes.

4) Scale and Edgefactor: Finally, we experimented with different values for *SCALE* and *edgefactor*. We compared again the OpenMP implementation with sequential prefetching and the vectorized version with combined vector-sequential prefetching. We used a fixed number of threads (160) for all experiments and run them in "native" mode.

Fig. 9 shows results for different *SCALE* values. The y-axis presents measured performance in harmonic TEPS and the x-axis shows different values for *SCALE*. It can be seen that the vectorized version consistently outperforms the sequential version. The vectorized version achieves the highest performance when *SCALE* is 21 and performance saturates for higher numbers. For *SCALE* lower than 14 the vectorized version achieves nearly no speedup and even has slowdown, for a *SCALE* of 11. This *SCALE* is extremely small since Graph500 aims to represent applications with very large graphs. In this case, the generated graph is small enough to fit into the L1 cache and be cache resident.

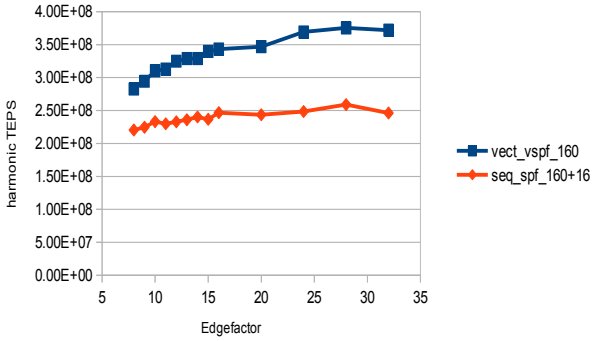

Figure 10. Impact of *edgefactor* on performance in native mode.

Fig. 10 shows results for different *edgefactor* values. The y-axis presents measured performance in harmonic TEPS and the x-axis shows different values for *edgefactor*. The vectorized version again outperforms the sequential version. As it is expected, the vectorized version benefits from a higher *edgefactor* while this gain is smaller for the sequential version. To quantify this, we have measured how often nodes are processed with vector instructions. For an *edgefactor* of 8, the vectorized loop processes 87% of the nodes, while the rest is executed in the scalar epilogue. For *edgefactor* of 32, 94.2% of the nodes are processed with vector instructions due to the increased vector lengths.

VI. Conclusion

In this paper we evaluated the vectorization potential of the Graph500 on the Xeon Phi. Applying vectorization on a single-threaded implementation provides negligible improvement and a hasty conclusion would be that Graph500 can not benefit of vector instructions. This conclusion is misleading because we achieve higher speedups when prefetching is combined with vectorization. Prefetching is more important for Graph500 running on the Xeon Phi, because the data is not cache resident. The combination of parallelization with vectorization and prefetching is very important for achieving higher performance. We achieve 27% of speedup for the best vectorized version with applied vector prefetch instructions over the best scalar implementation with sequential prefetching in "native" mode.

In the future, we will try to vectorize other implementations of Graph500. We will address issues that may be challenging for vectorization, e.g. the usage of bitmap structures to track unvisited nodes.

References

[1] R. C. Murphy, K. B. Wheele, B. W. Barret, J. A. Ang: "Introducing the graph 500." Cray Users Group (CUG) (2010).

[2] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan: Larrabee: a many-core x86 architecture for visual computing. In SIGGRAPH 08, pages 18:118:15, 2008.

[3] E. Saule and Ü. Catalyurek: An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture. In Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW '12). IEEE Computer Society, Washington, DC, USA, 1629-1639.

[4] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader: Scalable Graph Exploration on Multicore Processors. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10). IEEE Computer Society, Washington, DC, USA, 1-11.

[5] S. Hong, T. Oguntebi, and K. Olukotun: Efficient Parallel Graph Exploration on Multi-Core CPU and GPU. In Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11). IEEE Computer Society, Washington, DC, USA.

[6] S. Beamer, K. Asanović, and D. Patterson: Direction-optimizing breadth-first search. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12). IEEE Computer Society Press, Los Alamitos, CA, USA, , Article 12 , 10 pages.

[7] Intel: Intel Xeon Phi TM Coprocessor Instruction Set Architecture Reference Manual, 2012.

[8] J. L. Hennessy and D. A. Patterson: Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers Inc., 4 edition, 2006. Appendix F.

[9] S. Browne, P. J. Mucci, C. Deane, G. Ho: Papi: A portable interface to hardware performance counters. In Proc. of Department of Defense HPCMP Users Group Conf., 1999.