Breadth First Search Vectorization on the Intel Xeon Phi

Mireya Paredes, Graham Riley, and Mikel Luján
School of Computer Science, University of Manchester
Oxford Road, M13 9PL
Manchester, UK
paredesm@cs.man.ac.uk

ABSTRACT

Breadth First Search (BFS) is a building block for graph algorithms and has recently been used for large scale analysis of information in a variety of applications including social networks, graph databases and web searching. Due to its importance, a number of different parallel programming models and architectures have been exploited to optimize the BFS. However, due to the irregular memory access patterns and the unstructured nature of the large graphs, its efficient parallelization is a challenge. The Xeon Phi is a massively parallel architecture available as an *off-the-shelf* accelerator, which includes a powerful 512 bit vector unit with optimized scatter and gather functions. Given its potential benefits, work related to graph traversing on this architecture is an active area of research.

We present a set of experiments in which we explore architectural features of the Xeon Phi and how best to exploit them in a top-down BFS algorithm but the techniques can be applied to the current state-of-the-art hybrid, top-down plus bottom-up, algorithms.

We focus on the exploitation of the vector unit by developing an improved highly vectorized OpenMP parallel algorithm, using vector intrinsics, and understanding the use of data alignment and prefetching. In addition, we investigate the impact of hyperthreading and thread affinity on performance, a topic that appears under researched in the literature. As a result, we achieve what we believe is the fastest published top-down BFS algorithm on the version of Xeon Phi used in our experiments. The vectorized BFS top-down source code presented in this paper can be available on request as free-to-use software.

Keywords

graph algorithms; BFS; large scale; irregular; graph traversing; hyperthreading; thread affinity; prefetching

1. INTRODUCTION

CF'16, May 16 - 19, 2016, Como, Italy

ISBN 978-1-4503-4128-8/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2903150.2903180

Today, scientific experiments in many domains, and large organizations can generate huge amounts of data, e.g. social networks, health-care records and bio-informatics applications [8]. Graphs seem to be a good match for important and large dataset analysis as they can abstract real world networks. To process large graphs, different techniques have been applied, including parallel programming. The main challenge in the parallelization of graph processing is that large graphs are often unstructured and highly irregular and this limits their scalability and performance when executing on *off-the-shelf* systems [18].

The Breadth First Search (BFS) is a building block of graph algorithms. Despite its simplicity, its parallelization has been a real challenge but it remains a good candidate for acceleration. To date, different accelerators have been targeted to improve the performance of this algorithm such as GPGPUs [11] and FPGAs [28]. The Intel Xeon Phi, also known as MIC (Intel's Many Integrated Core Architecture), is a massively parallel *off-the-shelf* system consisting of up to 60 cores with 4-way simultaneous multi-threading (SMT) for a maximum of 240 logical cores [21]. As well as this thread-level parallelism, each core has a 512 bit vector unit allowing data-level parallelism to be extracted by using Single Instruction Multiple Data (SIMD) programming. We believe that by exploiting both of these forms of parallelism, the Xeon Phi is an interesting platform for exploring parallel implementations of graph algorithms.

The goal of this study is to demonstrate through experiments and analysis the impact of using the Xeon Phi architecture to accelerate BFS in a single Xeon Phi device, as a step towards the multi-device solutions that will be needed to tackle very large graph-based datasets. As a starting point, we took the description of the top-down BFS algorithm in [9] which is summarised in Section 3. Although this offers a good starting point on the Xeon Phi, there are still some architectural features that need to be well understood in order to be exploited. The contribution of this paper is twofold; first, we present the results of a series of experiments demonstrating the benefit of successfully exploiting architectural features of the Xeon Phi focusing on the vector unit (programmed via vector intrinsics), with data alignment and prefetching. In addition, we present the results of investigations into the impact of hyperthreading (Intel's term for SMT) and thread affinity when the Phi is underpopulated with threads.

The structure of the paper is as follows. The Xeon Phi architecture is presented in Section 2. We present the procedure of vectorizing the BFS algorithm on the Xeon Phi,

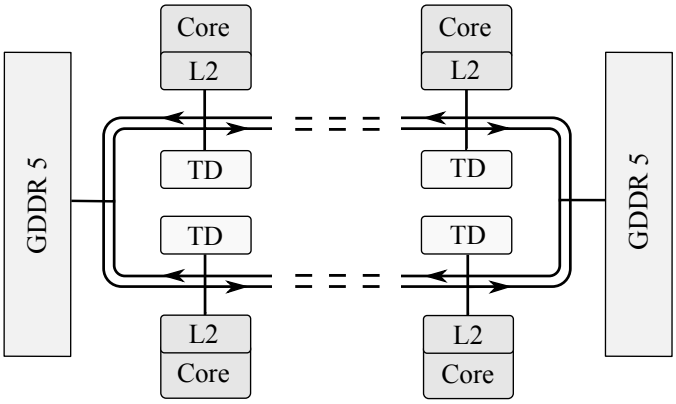


Figure 1: The Intel ® Xeon Phi Microarchitecture.

starting with the serial BFS algorithm in Section 3.1, followed by an initial parallel version in Section 3.2, which we call *non-simd* version. Next, we discuss the *simd* version that exploits the vector unit on the Xeon Phi in Section 4. We then present performance comparisons between the *non-simd* and *simd* versions and explore the impact of parallelism at two levels of granularity (OpenMP threading and the vector unit), achieving better results for the native BFS top-down algorithm on the Xeon Phi than those previously presented in [10] and [24]. In Section 4 we explore the impact of data prefetching and discuss the effect of thread affinity mapping, leading us to key optimizations, and motivating future work on the possibility of using helper threads to improve performance. The experimental setup and analysis of our results are shown in Sections 5 and 6. Related work is briefly discussed in Section 7 and conclusions and future work are discussed in Section 8.

2. THE XEON PHI ARCHITECTURE

The Intel ® Xeon Phi ™ coprocessor used in this work is composed of 60, 4 way-SMT Pentium-based cores [21] and a main memory of 8 GB. Each core contains a powerful 512-bits vector processing unit and a cache memory divided into L1 (32KB) and L2 (512KB) kept fully coherent by a global-distributed tag directory (TD), coordinated by the cache coherency MESI protocol. Cores are interconnected through a high-speed bi-directional ring bus [21] as it is shown in Figure 1. Maximum memory bandwidth is quoted as 320GB/s.

The vector process unit (VPU) is composed of vector registers and 16-bit mask registers. Each vector register can process either 16 (32-bit) operations or 8 (64-bit) operations at a time. A vector mask consists of 16 bits that control the update of the vector elements. Only those elements whose bits are set to 1 are updated into the vector register, the ones with 0 value in the mask remaining unchanged. The Xeon Phi contains both hardware (HWP) and software (SWP) prefetching, which in some cases can help to reduce memory latency.

The Xeon Phi can be programmed to support vectorization at two levels: automatic and manual. In automatic vectorization the compiler identifies and optimizes all parts of the code that can be vectorized without the intervention of the programmer. However, there are some obstacles that can limit the vector unit utilization, such as non-contiguous memory accesses or data dependencies [12] . In such cases, manual vectorization can be used allowing the user to force the compiler to vectorize certain parts in the code. Manual vectorization can be set by using SIMD pragma directives supported in the compiler. The compiler also supports a wide range of *intrinsics* which allow a programmer low-level control of the vector unit.

3. BREADTH FIRST SEARCH

The Breadth First Search (BFS) algorithm is one of the building blocks for graph analysis algorithms including betweenness centrality, shortest path and connected components [4]. Given a graph G and a starting vertex s, the BFS systematically explores all the edges E of G to trace all the vertices V that can be reached from s [7], where $|V|$ is the number of vertices and $|E|$ is the number of edges. The output is a BFS spanning tree (*bfs tree* of all the vertices encountered from s.

The simplest sequential BFS algorithm consists in having a queue that contains a list of vertices waiting to be processed. Enqueue and dequeue operations on the queue can be implemented in constant time $\Theta(1)$ [16]. Thus, the running time of this serial BFS algorithm is $O(V + E)$. Despite the queue being simple and efficient, during parallelization, it has some drawbacks. The queue can be a bottleneck since it implies a vertex processing order because the dequeue operation typically takes out the vertex that has been added first. To address this problem vertices can be partitioned into layers. A layer consists of a set of all vertices with the same distance[1] from the source vertex. Processing vertices by layers avoids the order restriction imposed by the queue, allowing vertices to be explored in any order as long as they are in the same layer. However, each layer has to be processed in sequence; that is all vertices with distance k, (layer L_k) are processed before those in layer L_{k+1}.

Traversing the graph from the vertices in the top layer down to their children in the bottom layer is a conventional approach known as *top-down* whereas traversing from the vertices in the bottom layer to find their parent in the top layer is known as *bottom-up* approach. The *bottom-up* concept was introduced by [3] in a hybrid approach that explores the graph in both directions, first traversing the graph with the top-down and then swaping to the bottom-up approach in the middle layers to finally swap back to the top-down. Despite the fact that the hybrid algorithm has shown better results than the top-down approach, we focus our work in this paper on the conventional top-down BFS to demonstrate how vectorization techniques can be applied to effectively utilize Xeon Phi's vector unit. The same techniques can be applied to the bottom-up phase, which can lead to speed up the hybrid BFS algorithm. Figure 2 shows an example of the top-down BFS algorithm. The exploration starts from vertex 1 and reaches all the vertices in the three layers illustrated by a, b and c in Figure 2. Dotted lines represent edges linked with already explored vertices.

3.1 The Serial Top Down BFS algorithm

The implementation of the serial top-down BFS algorithm uses two lists to process vertices by layers. The first list is the input list and it contains all the vertices to be processed in the layer. The second list is the output list which, after processing the layer will be swapped with the input list for

[1]Distance is the number of edges in the shortest path between two vertices.

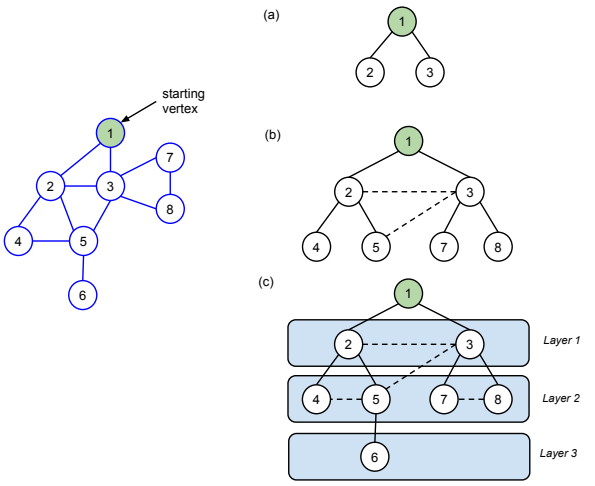


Figure 2: An example of the Top-Down Breadth First spanning tree.

the next layer. The result of the algorithm is a BFS spanning tree represented by a sequence of the predecessors (P) of the traversed vertices. When a vertex has been processed it is marked as *visited*, otherwise, it remains *non-visited*. Each vertex has an associated set of adjacent vertices, known as neighbor list. Only the non-visited vertices in the list are put into the output list to be processed in the next layer.

Algorithm 1 shows the pseudocode of the serial top-down BFS algorithm. The pseudocode uses four data structures: input list (*in*), output list (*out*), visited array (*visited*), and P, predecessor list; all data structures are initialized at the beginning. The visited array is used to mark vertices as *visited* during the exploration process. Initially all the vertices are set as *non-visited*. The predecessor array is used to store the output BFS spanning tree and is initialized with big number values, denoted by the symbol ∞ in line 2. In practice, ∞ can be an integer bigger than the number of vertices. The exploration starts when the input list *in* has at least one element (line 7). Thus, the starting vertex s is placed in the input list, visited array and in the predecessor array set as its own parent (lines 4-6).

In lines 7 to 17, every single vertex u in the input list *in* is explored. This exploration consists of checking each adjacent vertex v of u that has not been visited. If it is the case, then they are put into the output list *out*, marked as *visited* and the parent for the vertex v in the P array is set to u. By checking the non-visited vertices first, some extra work is avoided by not putting previously processed vertices in the list [1]. In line 16, the input and the output lists are swapped and the output list is cleared. The algorithm ends when all vertices reachable from the input vertex have been marked as visited in *out*. The output BFS spanning tree is the predecessors list P, which contains a record of the order that the vertices were explored.

Algorithm 1 Serial *Top-Down* BFS(G, s)

```
Initialize: in.init() out.init() vis.init()
 1: for all vertex u ∈ V(G) − s do
 2:     P[u] ← ∞
 3: end for
 4: in.add(s)
 5: vis.Set(s)
 6: P[s] = s
 7: while in ≠ 0 do
 8:     for all u ∈ in do
 9:         for all v ∈ Adj[u] do
10:             if vis.Test(v) = 0 then
11:                 vis.Set(v)
12:                 out.add(v)
13:                 P[v] = u
14:             end if
15:         end for
16:     end for
17:     swap(in, out)
        out ← 0
18: end while
```

3.2 Parallel Top-Down BFS

In the serial *top-down* BFS algorithm, there are two levels of parallelism that can be exploited. The first is a coarse-grain level in the outer loop for processing the input list; the second, finer-grain, is in the inner loop where the adjacency list is explored. Algorithm 2 shows the Parallel BFS by augmenting Algorithm 1 with parallel *for* loops. In practice, we aim to parallelize the outer loop using threads and the inner loop by exploiting the vector unit. The major changes to parallelize the algorithm are in the initialization of the lists, the visited and the predecessor array data structures, and in the outer and inner loops, lines 8 and 9, where all the vertices are explored in parallel.

Algorithm 2 Parallel Top-Down BFS(G, s)

```
Initialize: in.init() out.init() vis.init()
 1: for all parallel vertex u ∈ V(G) − s do
 2:     P[u] = ∞
 3: end for
 4: in.add(s)
 5: vis.Set(s)
 6: P[s] = s
 7: while in ≠ 0 do
 8:     for all parallel u ∈ in do
 9:         for all parallel v ∈ Adj[u] do
10:             if vis.Test(v) = 0 then
11:                 vis.Set(v)
12:                 out.add(v)
13:                 P[v] = u
14:             end if
15:         end for
16:     end for
17:     swap(in, out)
        out ← 0
18: end while
```

However, this parallel version presents a race condition between multiple threads. This condition happens in the exploration of the adjacency list when a vertex v is tested to verify if it has been *visited* previously. Multiple threads might test the same vertex at the same time. Figure 3 illustrates this race condition. Here, threads A and B are trying to update vertex 5, which is a child of both vertex 2 and 3. While this could end up in redundant work when the status of the vertex and the queue are updated, the major impact

is in the predecessor list, where the parent of vertex 5 can be set to either 2 or 3. However, this is called a benign race condition since the correctness of the algorithm is not affected. It means that different correct BFS spanning trees can be generated. It is possible to avoid this race condition by using an atomic operation such as *__sync_fetch_and_or*. However, we will see in the following section that another more critical race condition comes up when we introduce bitmap arrays as data structure.

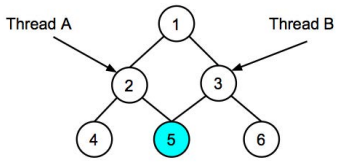


Figure 3: Example of data benign race condition.

3.3 Parallel Top-Down BFS without bit race conditions

In addition to the benign race condition described in Section 3.2, introducing bitmap arrays as data structures leads into a bit race condition that is solved by adding a restoration process to the parallel top-down BFS algorithm.

3.3.1 Data structures

Our parallel BFS implementation is based on the implementation, `bfs_replicated_csc`, given in the *Graph500* source code [27]. The graph is efficiently represented by a Compressed Sparse Row (CSR) matrix format, which is composed by two integer arrays: *rows* and *colstarts*. The *rows* array contains the adjacency list of every vertex and the *colstarts* stores the start and the end indexes of every vertex pointing to the *rows* array. An example of this data structure is illustrated in Figure 4. The use of the CSR is when the adjacency list is explored, line 9 in Algorithm 3 abstracts this step.

On the other hand, the data structures used for the input list, the output list and the visited arrays are bitmap arrays and an integer array for the predecessor array. A bitmap is a mapping from a set of items to bits with values either zero or one. By having vertices represented by a bitmap, the working set size can be reduced significantly [1]. For example, an array that holds 1,048,576 vertices represented by integers would require 4MB. By using bitmaps this memory storage can be reduced significantly to 131,072 bytes. Figure 5 illustrates the mapping between an array of integers to bits. The upper array is an array of integers, where every vertex corresponds to each index in the array. Values can be set to either zero or one and the length of the array is the total number of vertices. In this example, vertex 28 and

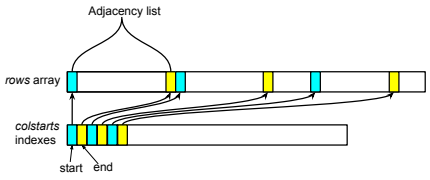


Figure 4: Compressed Sparse Row representation.

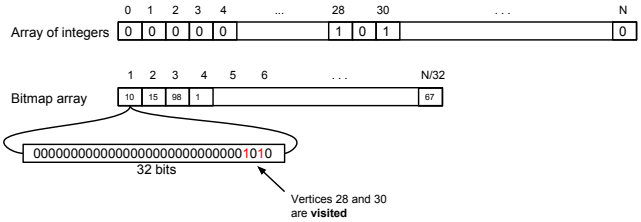


Figure 5: Example of an array of integers (32 bits) represented by a bitmap array.

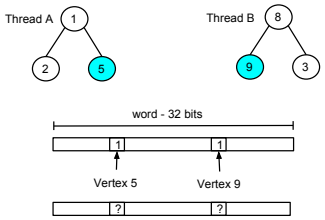


Figure 6: Example of the visited bitmap array race condition.

30 are set to one. The array at the bottom is the bitmap array and its length is the total number of vertices divided by the length, in bits, of an integer (32 bits). Every integer in the bitmap array represents the status (0, 1) of 32 vertices. Thus, the same vertices 28 and 30 are set to one but they are both located in the first integer of the array as it is illustrated.

3.3.2 Restoration process

A race condition is raised by the bitmap update operations when different threads try to update multiple bit values in the same word. An example of this is shown in Figure 6, where vertices 5 and 9 are updated by different threads but their location in the bit array is in the same word (32 bits integer). The bitmap race condition happens, in the adjacency list exploration, while setting the output and the visited bitmaps in lines 9 to 12 of the Algorithm 2. Thus, to overcome this problem a restoration process is applied as in [10] to both bitmap arrays afterwards, the visited and the output. That way, the restoration process helps to keep the output and the visited bitmaps consistent for processing the next layer.

The restoration consists of finding all the corrupted words in the output bitmap array that were updated, if a bit race condition happened the output array should have at least one bit set. Since the predecessor list is an array of integers, it does not present the bit race condition and is used to fix the corrupted output array. To do so, first we identify which vertices were updated in the predecessor list by setting a negative number, which is a subtraction of the number of nodes from the vertex's parent. Second, we iterate through the non zero words (32-bits integer) in the output bitmap array. In those words are the vertices that are corrupted and need to be fixed. We step through each of the 32 bits in the word to look for the corresponding vertices, in lines 20 and 21 of Algorithm 3, that have been set to a negative number in the predecessor list. These vertices are restored by setting their corresponding bit in the output bitmap and

adding back the number of nodes in the predecessor list. Finally, the visited bitmap array is updated consistently.

Although the restoration process implies extra work, it solves the bit race condition without having to use atomic operations while keeping correctness and even more important allow us to use it in the vectorization process described in Section 4. Algorithm 3 shows the complete pseudocode of the BFS algorithm containing the restoration process to cope with the bitmap race condition, lines 15 to 30. However, the benign race condition described in 3.2 still remains since we avoid making use of atomic operations for further vectorization. Bitmap operations used by the visited, input and output arrays are: $InitBitmap()$, $SetBit(n)$, $GetBit(n)$, $TestBit(n)$ and $bit2vertex(n)$, where n is the bit position.

Algorithm 3 Parallel BFS without bit race conditions.

Initialize: $in.InitBitmap()$ $out.InitBitmap()$ $vis.InitBitmap()$
1: **for all parallel** vertex $u \in V(G) - s$ **do**
2: $P[u] = \infty$
3: **end for**
4: $in.SetBit(s)$
5: $vis.SetBit(s)$
6: $P[s] = s$
7: **while** $in \neq 0$ **do**
8: **for all parallel** $u \in in$ **do**
9: **for all parallel** $v \in Adj[u]$ **do**
10: **if** $v \notin (vis.TestBit(v)$ OR $out.TestBit(v))$ **then**
11: $out.SetBit(v)$
12: $P[v] = u - nodes$
13: **end if**
14: **end for**
15: //Restoration process
16: **for all parallel** $w \in out$ **do**
17: // w is a word in out bitmap
18: **if** $w \neq 0$ **then**
19: // iterate through every bit in w
20: **for all** $b \in w$ **do**
21: vertex = bit2vertex(b)
22: **if** $P[vertex] < 0$ **then**
23: $out.SetBit(vertex)$
24: $vis.SetBit(vertex)$
25: $P[vertex] = P[vertex] + nodes$
26: **end if**
27: **end for**
28: **end if**
29: **end for**
30: **end for**
31: $swap(in, out)$
$out \leftarrow 0$
32: **end while**

4. BFS VECTORIZATION

Our vectorized BFS algorithm is based on the parallel BFS algorithm without using atomic operations presented in Algorithm 3. Basically, it avoids atomic operations by using an extra step to restore possible missing values in the output queue due to a data race condition and the lack of atomic operations at bit level. This atomicity freedom allows to vectorize the algorithm straight away because atomic bit operations are not part of the instruction set architecture (ISA) in the Intel compiler[20].

There are two potential parts to be vectorized in the algorithm: the adjacency list exploration and the restoration process. In the adjacency list exploration, each element in the list is explored sequentially. By using the vector unit, instead of exploring one element at a time, it could be possible for one thread, to explore 16 (32-bits integers) vertices simultaneously. The vectorization of the adjacency list exploration involves three main SIMD steps. Firstly, a sequence of vertices in the adjacency list are loaded into the vector unit, which can hold 16 (32-bits) vertices. Secondly, all the loaded vertices are filtered by using the visited array and the output queue bitmaps to find the ones that have not been visited yet either in previous layers (visited array) or in the current layer (output queue). Finally, the result is set back to the predecessor array P and the output queue *out*. Figure 8 shows an example of the vectorization of the adjacency list exploration in a vector register of 512 bits wide. The specific values in the visited and output queue bitmap arrays are loaded into the vector unit by using the SIMD *gather* instructions. The *scatter* and the *gather* operations are two instructions that the Xeon Phi use to deal with non-contiguous memory loads to the vector register and data stores back to memory. Both operations receive, as argument, a list of indexes to be scattered/gathered in the array. Figure 7 illustrates both instructions to update the visited array. Since the visited array is a bitmap array an index transformation is necessary to load the bit word (32 bits) specific to a vertex. Then, the vector unit can do bit shifting operations to get the bit value of the vertex.

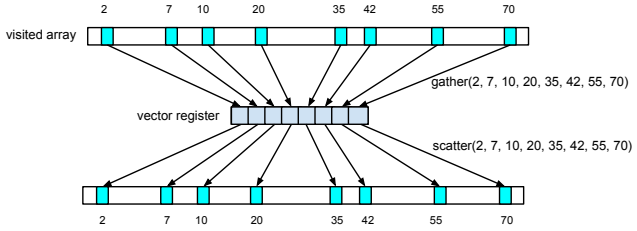


Figure 7: *Gather* and *scatter* operations, to load non-contiguous visited array.

Once the specific values of the visited array and the output queue are loaded into the vector unit, a filtering process is applied. To find all the vertices that have not been either visited previously (upper layers) or put into the output queue recently (current layer), two logical operations (OR and NOT) are used to create a vector mask. The *OR* operation will find the union of the vertices that have already been visited and the ones that have been put into the output queue. By applying the *NOT* logical operation, we can find all the vertices that have not been visited or set into the output queue. Afterwards, the result is scattered into the predecessor array and the output queue, only those indexes that have one as bit value in the mask are updated.

On the other hand, the vectorization of the restoration process adds an extra step to the parallel Algorithm 3 and consists of repairing the output queue and the visited array based on the P array, which remains consistent. To vectorize it, there are some details to take into account. Firstly, there is a difference between the output queue and the P array representation. While the output queue is a bitmap array, P is an array of integers. Thus, during stepping through each of the words (32 bits length) of the output queue, 32 vertices are expected to be processed, however the vector unit can only process up to 16 elements at a time. To cope with that, we split the word in two: the low part and the

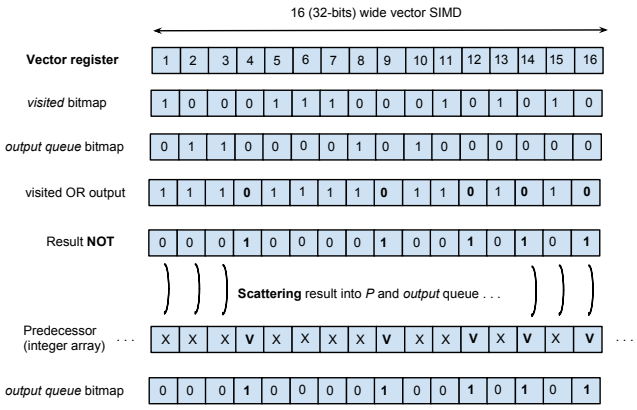


Figure 8: Example of vectorizing the adjacency list exploration.

high part. Then, the restoration process is divided into two sections, the first one repairs the vertices that are located in the low part and the second section repairs vertices located in the high part. Source code of the SIMD implementation is presented at the end of Section 4.2.

4.1 Which layers are vectorized?

Large and sparse graphs often present the *small-world* graphs properties of having small diameter and skewed degree distribution [5]. We use RMAT, a synthetic graph generator, to create our input *small-world* large graphs. The RMAT graph size is defined by two input values: the $SCALE$ and the *edgefactor*. The total of vertices in the graph is calculated by 2^{SCALE} and the number of edges generated by $2^{SCALE} * edgefactor$, including self-loops and repeated edges. Graph structure is crucial because it can help to improve performance by exploiting the architecture resources efficiently, such as the vector unit on the Xeon Phi. Table 1 shows the number of input vertices, the number of edges and the traversed vertices by the BFS top-down algorithm per layer, for a graph of 1,048,576 million vertices (RMAT graph with $SCALE$ 20 and edgefactor 16), choosing the starting vertex randomly. As it can be seen, the number of input vertices per layer increase along with the number of layer until the middle layer is reached and then they start to decrease. The diameter of the graph is 7, which is reflected in having 7 traversed layers. On the other hand, the number of edges per layer varies according to the *edgefactor*, which is used to distribute the edges per vertex. Both graph characteristics, diameter and vertex degree are key to decide what is the best way to improve threads workload imbalance and to increase the vector unit usage. Assuming that most of the vertices are traversed in the first layers, we used the vectorized SIMD BFS top-down algorithm only for the first two layers and the parallel top-down presented in Algorithm 2 for the rest of the layers.

4.2 Xeon Phi optimizations

To optimize the vectorization of the BFS top-down algorithm, there are some crucial factors such as cache-oriented memory address alignment, loop vectorization, masking and prefetching [12].

Table 1: Traversed vertices per layer for a 1,048,575 million vertices graph, $SCALE$ 20 and *edgefactor* 16.

Layer	Vertices	Edges	Traversed vertices
0	1	12	12
1	12	21,892	18,122
2	18,122	13,547,462	540,575
3	540,575	17,626,910	100,874
4	100,874	150,698	486
5	486	490	4
6	2	2	0

Intrinsic functions.

Despite the promise of automatic vectorization, this is not straightforward for algorithms with irregular data access patterns, such as the BFS. Some explicit manual code transformations are needed to assist the compiler. An explicit way to use the vector unit is through *intrinsic functions*, these are a set of assembly functions that allow complete control over the vector unit. We used specialized intrinsic functions which are part of the Intel AVX-512 instruction set.

Data alignment.

Data alignment specifications assist the compiler operation in the creation of objects on specific byte boundaries [13]. The optimal data alignment for the Xeon Phi is on 64 byte boundaries. To align the *rows* array, we used the intrinsic function *mm_malloc*. This data alignment allows the vector unit to have more efficient access to the memory. However, due to the way the *rows* array is built, it might lead to some *less-than-full-vector* loop vectorization inefficiencies, such as the *peel* and the *remainder* loops [25], caused non-aligned boundary accesses.

Peel and remainder loops.

The *peel* loop is the sequence of contiguous elements for which the start index in the array does not match with a aligned boundary of the array. On the contrary, the remainder loop is the sequence of the last elements, the tail, that do not fit on an aligned boundary. Both cases imply an extra processing step because they cannot be computed as a complete 16 elements vector. There are different approaches to cope with the implications of having *less-than-full-vector* loops including *padding*, sequential processing of *peel* and *remainder* loops, and the use of vector masks.

Prefetching.

Prefetching is a technique that helps hide memory latency. The Xeon Phi has a heuristic to choose between hardware or software prefetching. Due to the irregular data access patterns that the BFS algorithm shows, software prefetching is necessary. In addition, working with large graph sizes might result in having large numbers of L2 cache misses, and thus poor performance. Prefetch *distance* is a metric that hints to the compiler the right number of cycles to load data ahead of it use. Finding the right distance is crucial to gain performance [19]. A recommended distance should be one similar to memory latency [2]. Rather than calculate a precise distance, and based on the work in [14], in the adjacency list exploration we prefetch the rows array for the vertices that will be processed in the next iteration. Also, we use prefetching intrinsic functions for every data load/store

to the vector unit by setting a hint that indicates which memory will be prefetched to either L1 cache (_MM_HINT_T0) or to L2 cache(_MM_HINT_T1).

The vectorization of the adjacency list exploration consists of splitting the list into chunks of 16 (32 bit) elements. The peel and the remainder loops are considered special cases because vertices need to be filtered according to the precalculated mask. Listing 1 shows the source code of the vectorization of the adjacency list for a *full-vector*, using optimizations such as alignment, masks and prefetching. The code consists of three main steps described previously in the algorithm: loading the adjacency list, filtering non-visited vertices and setting the values back to memory. However, an intermediate operation is needed due to the discrepancy between the indexes of the 16 input vertex list (32-bit integers) and the bitmap arrays (bits) in the step 2: filtering the unvisited vertices in the visited array and the output queue. This step is implemented by getting the word and the bit offset of each element in the adjacency list. The words vector is used to gather all the words to be updated from the visited array and the output queue. The bit offset vector is used to create a mask to filter the specific bit values by shifting it to the left. The words and the bit offsets are used to generate a vector mask that allows to filter the visited and the output queue bitmap arrays respectively.

```
/* 1.- Load adjacency list to the register */
__m512i vneig = _mm512_load_epi32(&rows[index * 16);

/* Getting word and bit offset */
__m512i vword = _mm512_div_epi32(vneig, _mm512_set1_epi32(
     BITS_PER_WORD));
__m512i vbits = _mm512_rem_epi32(vneig, _mm512_set1_epi32(
     BITS_PER_WORD));

/* Gathering words from visited bitmap array */
_mm512_prefetch_i32gather_ps(vword, explored->start, sizeof(
     word_t), _MM_HINT_T0);
_mm512_prefetch_i32gather_ps(vword, queue->start, sizeof(word_t)
     , _MM_HINT_T0);

__m512i vis_words = _mm512_i32gather_epi32(vword, explored->
     start, sizeof(word_t));
__m512i out_words = _mm512_i32gather_epi32(vword, queue->start,
     sizeof(word_t));

/* Shifting 1 to the left indexes position in the vneig array */
__m512i bits = _mm512_sllv_epi32(_mm512_set1_epi32(1), vbits);

__mask16 mask = _mm512_knot(_mm512_kor(_mm512_test_epi32_mask(
     vis_words, bits), _mm512_test_epi32_mask(out_words, bits)))
     ;

/*3.- Scattering P (bfs_tree) and output queue */
_mm512_mask_prefetch_i32scatter_ps(bfs_tree, mask, vneig, sizeof
     (word_t), _MM_HINT_T0);
_mm512_mask_i32scatter_epi32(bfs_tree, mask, vneig,
     _mm512_set1_epi32(vertex - nodes), sizeof(word_t));

/* Setting the output queue */
//Adding to the output queue word the new bit values depending on
     the filtered mask.
__m512i new_values = _mm512_mask_or_epi32(_mm512_set1_epi32(0),
     mask, out_words, bits);

_mm512_mask_prefetch_i32scatter_ps(queue->start, mask, vword,
     sizeof(word_t), _MM_HINT_T0);
_mm512_mask_i32scatter_epi32(queue->start, mask, vword,
     new_values, sizeof(word_t));
```

Listing 1: Adjacency list exploration using SIMD intrinsic functions.

Figure 9 shows different results for the BFS top-down native algorithm on the Xeon Phi. The experiments involved three implementations, including the SIMD without optimizations (SIMD - no opt), the combined (SIMD + parallel) BFS algorithm plus alignment and masks optimizations and the one with prefetching. As it can be seen performance was

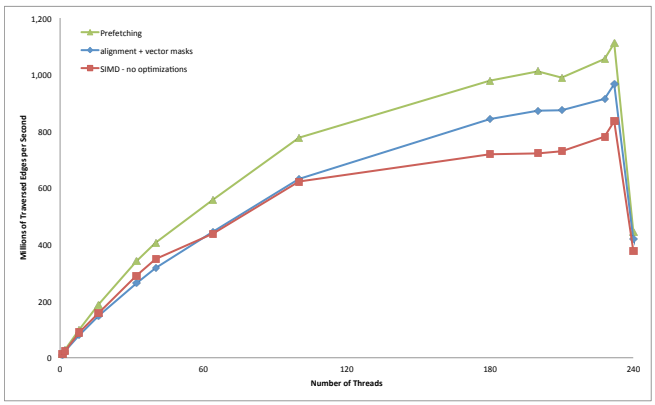


Figure 9: BFS experimental optimizations results for a graph of *SCALE* 20 and *edgefactor*=16.

increased after applying prefetching. Future work will target improving prefetching by finding a good prefetch distance.

Thread affinity.

Thread affinity is a feature in hyper-threading architectures that enables the pinning of threads to specific *logical cores* on the same physical core, enabling the user to affect the use of shared resources in a core (e.g. cache and memory bandwidth). The Xeon Phi allows up to 4 hardware threads per core and provides three strategies for controlling thread affinity: compact, scatter and balanced. *Compact* affinity assigns free threads as close as possible to the thread contexts in a core (i.e. to logical cores on the same physical core); *scatter* distributes the threads as widely as possible across the entire set of physical cores, using one logical core per physical core for each thread placed up to the number of physical cores and then cycling through the physical cores again. Finally, *balanced* is similar to scatter but places threads with adjacent thread ids on the same core. Regardless of the affinity strategy used, if the number of threads is equal to the number of logical cores available, the affinity is equivalent to compact (i.e. four threads per physical core) and resource sharing is at a maximum.

Since the effect of thread affinity on performance is application dependant (for example, sharing cached data between thread may be beneficial but sharing memory bandwidth may not, depending on the application), we ran some experiments to determine which strategy works best for our BFS algorithm. By experimenting we found that balanced affinity was generally better and it was used in the results presented in Figure 9. However, we followed the methodology presented in [22] to demonstrate the effect of affinity on the BFS algorithm, we ran a 48 thread version manually controlling the affinity to achieve one, two, three and four threads per core (1T/core, 2T/core, 3T/core and 4T/core), thus using 48 cores with one thread per core down to 12 cores with four threads per core. The thread affinity is controlled through the environment variable *KMP_AFFINITY*.

5. EXPERIMENTAL SETUP

5.1 Hardware platform

We evaluate the *non-SIMD* version presented in Algorithm 2 and the vectorized version, *simd*, of the BFS top-down algorithm on the Intel Xeon Phi. We used OpenMP as a thread parallel platform and Linux as execution platform [2]. We compiled our code with the Intel C++ compiler (version 14.0.0) with the optimization flag *-02* and *-fopenmp* [3]. We used the intrinsic functions which give access to the AVX-512 instructions set. We used Linux as platform.

5.2 Input graph

Our implementation uses different modules of the Graph500 benchmark [26], including the graph generator, the BFS path validator, the experimental design and the ability to run our parallel BFS implementation. Firstly, the graph generator creates synthetic scalable large *Kronecker* graphs [17] and is based on the R-MAT random graph model [5]. These graphs aim to naturally generate graphs with common real network properties in order to be able to analyse them. The graph size is defined by the *SCALE* and the *edgefactor* values. The total number of vertices in the graph is calculated by 2^{SCALE} and the number of edges generated by $2^{SCALE} * edgefactor$ * 2 (the factor of 2 reflects the fact that the edges are bidirectional). The generator uses four initiator parameters (*A, B, C* and *D*), which are the probabilities which determine how edges are distributed in the adjacency matrix representing the graph. We used the standard set of parameters defined by Graph500 (A=0.57, B=0.19, C=0.19 and D=0.05).

5.3 Implementation details

The input graph is efficiently represented by a Compressed Sparse Row (CSR) matrix format. The validation method, which checks the correctness of the algorithm, consists of five check results that do not intend to get a full check of the generated output (the bfs spanning tree) but just provide a 'soft' check of the output. The experimental design comprises 64 BFS executions each with a randomly chosen different starting vertex. The time of each execution is measured in seconds. After the completion of the executions, statistics, including time and Traversed Edges Per Second (TEPS), are collected. TEPS is a *Graph500* performance metric used to compare other BFS implementations on different architectures. However, it became apparent that out of the 64 BFS iterations, some of the starting points are unconnected, resulting in zero TEPS values for those iterations. This then results in having a harmonic mean, as calculated by Graph500, higher than the maximum number of TEPS. Some groups filter out such unconnected nodes in their experiments. Our results show the harmonic mean of the TEPS across the 64 executions without filtering in order to compare fairly with other implementations such as [10] and [3].

Result presentation: The experimental results reported result from a sequence of sets of 64 executions (one for each selected start vertex) in which we varying the following parameters: the number of threads and the graph *SCALE* factor (the edgefactor is fixed at 16). The number of threads were chosen as: 1, 2, 8, 16, 32, 40, 64, 100, 180, 200, 210, 228, 232 and 240 (the maximum number of one thread per logical core). The *SCALE* were set to 18, 19 and 20.

[2]The Linux operating system allows access to the OpenMP library with flag *-fopenmp*.

[3]Higher optimisation levels did not improve performance

Table 2: Performance *SIMD* version by setting thread affinity for a graph size of SCALE=20 and edgefactor=16.

#Threads	Thread Affinity	Cores	TEPS
48	1T/C	48	**4.69E+08**
	2T/C	24	2.67E+08
	3T/C	16	1.89E+08
	4T/C	12	1.42E+08

6. RESULTS AND ANALYSIS

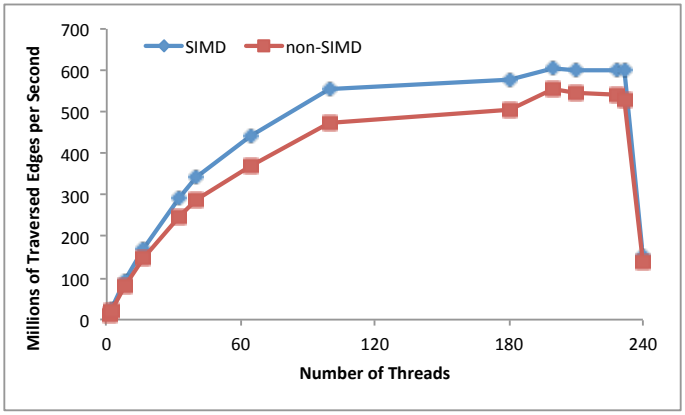
6.1 SIMD version versus non-SIMD version

In Section 4.2 we present results illustrating the SIMD optimizations. Figure 10 shows the experimental results for our *non-simd* version and for the *simd* optimized version, which are described in Algorithm 2, for three different graphs sizes, SCALE (18, 19 and 20), and edgefactor 16. Both versions present similar scalability but the simd version is around 200 MTEPS faster than the non-simd one. However, as the number of threads increases the rate of increase in TEPS decreases. This is a result of hyprethreading and is discussed in Section 6.2. Finally, the variation in performance between 200 and 236 threads is due to workload imbalance during the exploration of the vertices in each layer since, as the number of threads increases, the chances of vertices processed by a thread having an uneven number of adjacent vertices increases. The results show that our maximum number of TEPS is above 1 gigatep. This is higher than that publishedf in [10], which gives approximately 800 MTEPS, for their native BFS algorithm for the same graph size - the highest top-down performance figure on a similar Xeon Phi that we have found in the literature.

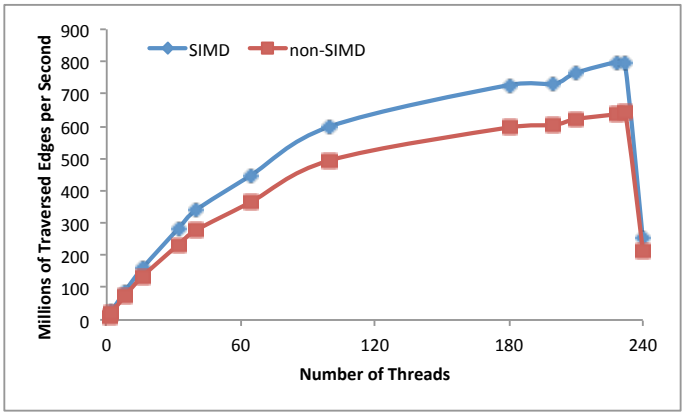
6.2 Thread affinity

Table 2 shows the result of running with 48 threads but varying the number of threads pinned per physical core manually. These results show the detrimental effects of overpopulating the cores for the BFS. The TEPS obtained with one and two threads per core are significantly higher than the TEPS with three and four threads per core. This reduction in the TEPS rate as the number of threads per core increase is the key driver to the changes in slope observed in Figure 10c occuring around 60, 120 and 180 cores when the number of threads per core has to increase as more threads are used. At these points, each thread's exclusive access to cache space decreases as does its share of memory bandwidth.Despite these changes in slope, the performance of the both the simd and non-simd BFS top-down algorithm continues to scale. So, by using fully populated cores (59), each one with the maximum number of threads, the number of TEPS for 236 logical threads is the fastest. Beyond 236 thread, threads are placed on the final core, which is reserved for the operating system on the Phi, resulting in a dramatic fall in performance.

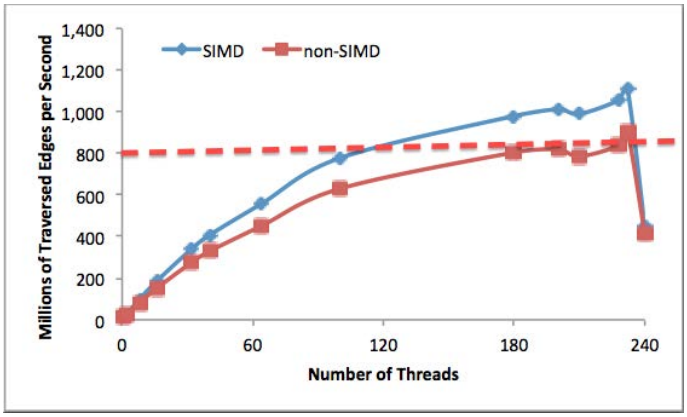
In the future, it might be possible to exploit this behaviour by under-populating cores with threads performing BFS and to make use of so-called *helper* threads running on a core to assist with, for example, prefetching to help hide memory latency [15].



(a) $SCALE = 18$ and $edgefactor = 16$.



(b) $SCALE = 19$ and $edgefactor = 16$.



(c) $SCALE = 20$ and $edgefactor = 16$.

Figure 10: BFS *non-SIMD* and SIMD experimental results for $SCALE$ values of 18, 19, 20 and $edgefactor = 16$. In Figure (c) the dashed line represents the best MTEPS reported in [10].

7. RELATED WORK

This section briefly presents related work targeting mainly the top-down BFS on the Xeon Phi. The Intel MIC Xeon Phi is a relative recent parallel architecture released in 2012. An early work related with graph algorithms on the Xeon Phi was reported [23]. However, they port a BFS parallel algorithm without actually using the vector unit of the Xeon Phi. Other, and most recent, work which is more related with ours is [9] and [10]. In [9] they present the vectorization of the top-down BFS algorithm, whereas in [10], a complete hybrid (top-down and bottom-up) heterogeneous BFS algorithm is presented. However, few details about their use of vectorization are presented. [24] also explored the top-down BFS on the Xeon Phi but with a traditional, queue-based, algorithm that uses atomic updates. Despite their exploration related with the use of the vector unit and prefetching, their results are much lower than ours. We implemented our BFS top-down algorithm using SIMD instructions by extending their work, which lead us to obtain a faster vectorized implementation. Another approach to exploiting the Xeon Phi is presented in a recently published system for heterogeneous graph processing [6] allows the utilization of the Xeon Phi in combination with a multi-core CPU through the use of a simple programming interface. Although it is a good start towards heterogeneous systems working on graphs, it is not made clear in this paper how fast the resulting implementation is in comparison with previous BFS implementations on the Xeon Phi.

8. CONCLUSIONS

In this paper, we revisited the vectorization and performance issues of the top-down BFS algorithm on the Xeon Phi. In particular, we studied the BFS top-down algorithm without (bit-wise) race conditions to improve vectorization. The contributions of the paper are, first, the development of an improved OpenMP parallel, highly vectorized SIMD version of the BFS using vector intrinsics and successfully exploiting data alignment and prefetching. This new implementation achieves a higher number of TEPS than previous published results for the same type of Xeon Phi. The second contribution is an investigation into the impact of the thread affinity mapping and hyperthreading on performance on an underpopulated system, a topic under researched in the literature. This work suggests future possibilities to take advantage of under-populating the cores and using the spare capacity to improve latency hiding through the use of helper threads, for example, to complement prefetching.

In the future, we plan to explore the benefit of our vectorization techniques beyond their use in native mode on the Xeon Phi, including targeting offload mode, GPGPUs and in the context of SSE and AVX SIMD technologies on multicore CPU-based systems. In addition, we are working on a version of the state-of-the-art hybrid BFS algorithm.

Acknowledgment

This research was conducted with support from the UK Engineering and Physical Sciences Research Council (EPSRC) PAMELA EP/K008730/1, and AnyScale Apps EP/L000725/1. M. Paredes is funded by a National Council for Science and Technology of Mexico PhD Scholarship. Mikel Luján is supported by a Royal Society University Research Fellowship.

9. REFERENCES

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable Graph Exploration on Multicore Processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[2] A.-H. Badawy, A. Aggarwal, D. Yeung, and C.-W. Tseng. The efficacy of software prefetching and locality optimizations on future memory systems, 2004.

[3] 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, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[4] M. Capota, T. Hegeman, A. Iosup, A. Prat, O. Erling, and P. A. Boncz. Graphalytics: A Big Data Benchmark For Graph-Processing Platforms. In *Proceedings of the International Workshop on Graph Data Management Experiences and Systems (GRADES, 2015)*, May 2015.

[5] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *In SDM*, 2004.

[6] L. Chen, X. Huo, B. Ren, S. Jain, and G. Agrawal. Efficient and simplified parallel graph processing over CPU and MIC. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 819–828, 2015.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 2009.

[8] D. Ediger, K. Jiang, J. Riedy, and D. Bader. GraphCT: Multithreaded Algorithms for Massive Graph Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 99, 2012.

[9] T. Gao, Y. Lu, and G. Suo. Using MIC to Accelerate a Typical Data-Intensive Application: The Breadth-first Search. In *IPDPS Workshops*, pages 1117–1125. IEEE, 2013.

[10] T. Gao, Y. Lu, B. Zhang, and G. Suo. Using the intel many integrated core to accelerate graph traversal. *IJHPCA*, 28(3):255–266, 2014.

[11] 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, pages 78–88, Washington, DC, USA, 2011. IEEE Computer Society.

[12] Intel Corporation. *A Guide to Vectorization with Intel® C++ Compilers*. 2012.

[13] J. Jeffers and J. Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Elsevier Waltham (Mass.), Amsterdam, Boston (Mass.), Heidelberg..., et al., 2013.

[14] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *PVLDB*, 8(6):642–653, 2015.

[15] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads. *SIGPLAN Not.*, 46(3):393–404, Mar. 2011.

[16] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 303–314, New York, NY, USA, 2010. ACM.

[17] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.*, 11:985–1042, Mar. 2010.

[18] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[19] S. Mehta, Z. Fang, A. Zhai, and P.-C. Yew. Multi-stage coordinated prefetching for present-day processors. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 73–82, New York, NY, USA, 2014. ACM.

[20] R. Rahman. *Intel Xeon Phi Coprocessor Instruction Set Architecture Reference Manual*. 2012.

[21] R. Rahman. *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, Berkely, CA, USA, 1st edition, 2013.

[22] A. Rodchenko, A. Nisbet, A. Pop, and M. Luján. Effective barrier synchronization on intel xeon phi coprocessor. In *Euro-Par 2015: Parallel Processing - 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings*, pages 588–600, 2015.

[23] E. Saule and Ü. V. Çatalyürek. An early evaluation of the scalability of graph algorithms on the intel MIC architecture. In *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPS 2012, Shanghai, China, May 21-25, 2012*, pages 1629–1639, 2012.

[24] M. Stanic, O. Palomar, I. Ratkovic, M. Duric, O. S. Unsal, A. Cristal, and M. Valero. Evaluation of vectorization potential of graph500 on intel's xeon phi. In *International Conference on High Performance Computing & Simulation, HPCS 2014, Bologna, Italy, 21-25 July, 2014*, pages 47–54, 2014.

[25] X. Tian, H. Saito, S. Preis, E. N. Garcia, S. Kozhukhov, M. Masten, A. G. Cherkasov, and N. Panchenko. Practical SIMD Vectorization Techniques for Intel Xeon Phi Coprocessors. In *IPDPS Workshops*, pages 1149–1158. IEEE, 2013.

[26] J. Willcock. Graph 500 Benchmark, nov 2012. Presentation at Super Computing 2012.

[27] J. Willcock and A. Lumsdaine. Graph 500 project on Gitorious. https://www.gitorious.org/graph500/graph500.

[28] D. M. Yaman Umuroglu and M. Jahre. Hybrid breadth first search on a single-chip fpga-cpu heterogeneous platform. *FPL2015*, 2015.