Bachelor Informatica

Universiteit van Amsterdam

Irregular algorithms on the Xeon Phi

Sander Lijbrink

June 17, 2015

Supervisor(s): Merijn Verstraaten (UvA)

Signed:

Abstract

The Xeon Phi is a coprocessor first released in 2012 by Intel. With x86 instruction set support, 60 cores and up to 2 teraflops of single-precision performance, the Xeon Phi seems promising and has gained wide interest. The world's fastest supercomputer to date, the Tianhe-2, features the Xeon Phi, so does the recently announced 180 petaflops supercomputer *Aurora*. Irregular algorithms such as graph processing algorithms are the core of many high-performance applications. Due to random memory access patterns and workload imbalance, this family of algorithms is challenging to implement on parallel hardware. This thesis investigates the performance of these algorithms on the Intel Xeon Phi 5110P compared to a dual-socket Xeon X5650 setup. We try to determine what kind of workloads can be run efficiently on the Xeon Phi. First, we show that the memory bandwidth of the Xeon Phi is significantly higher than the Xeon. We also show that the Xeon Phi performs better than the Xeon in matrix multiplication, a regular and vectorizable algorithm. To determine the impact of the Xeon Phi's architecture on irregular algorithms, we have implemented and evaluated two key graph algorithms, breadth-first search and PageRank. We conclude that memory-intensive algorithms such as BFS perform well on the Xeon Phi. However, compute-intensive algorithms such as PageRank suffer from the Xeon Phi's architectural trade-offs.

Contents

CHAPTER 1

Introduction

Modern science makes extensive use of computer simulation and with the rise of "big data", the demand for high-performance computing (HPC) has increased. To meet this demand, the use of *accelerators* such as Graphical Processing Units (GPUs) has gained popularity for massively parallel workloads. In 2012, Intel released the Xeon Phi coprocessor, a 60-core accelerator card with up to 2 teraflops of single-precision performance. In contrast to GPUs, the Xeon Phi features x86-64 compatibility and supports traditional parallel programming models such as OpenMP and MPI. This allows programmers to run existing C/C++/Fortran code on the Xeon Phi with little effort.

Due to its flexibility, high memory bandwidth and parallelism, the Intel Xeon Phi has attracted attention in HPC. The Tianhe-2, the fastest supercomputer in the world today, utilizes Xeon Phi nodes to reach a 33.86 petaflops[13]. Supercomputer maker Cray has announced the new 180 petaflops supercomputer *Aurora* for the US government, which will be based on next-generation Xeon Phi coprocessors and is scheduled for 2018.

The Xeon Phi is interesting because it combines high memory bandwidth with massive parallelism but at the same time offers the flexibility of conventional CPUs. It requires a host CPU and communicates over PCI-Express. It is well-suited to parallel computational tasks, which are common in HPC. The Xeon Phi can be used to offload tasks from the host CPU, much like a GPU. The Xeon Phi has cache-coherency which is not available in GPUs and a unique high-bandwidth ring interconnect for communication between each core.

Irregular algorithms are the core of many high-performance applications. Irregular algorithms are algorithms that involve irregular memory accesses and input-dependent workloads. Examples of these are graph algorithms, N-body simulations and sparse matrix operations. This family of algorithms is challenging to implement efficiently on parallel hardware, due to random memory access patterns and unpredictable computational workload.

Graph algorithms, such as breadth-first search, are largely dominated by memory accesses rather than computational work. The Xeon Phi's architecture, which trades single-core performance for parallelism and memory bandwidth, might be a good fit for these kind of tasks.
The main research question of this thesis is: "What is the impact of the Xeon Phi's architectural trade-offs on the performance of graph algorithms?". Given the memory-intensive nature of graph algorithms, it is also relevant to know if the Xeon Phi can deliver the memory performance it promises on paper. Therefore, we measure the memory bandwidth we can achieve in practice. We then try to determine what kind of workloads can be run efficiently on the Xeon Phi. To do this, we evaluate two key graph algorithms, breadth-first search(BFS) and PageRank.

CHAPTER 2

Intel Xeon Phi

2.1 Architecture

The Intel Xeon Phi is based on the *Many Integrated Core architecture* (MIC). This architecture uses previous concepts developed by Intel for the Larrabee many core architecture, as well as the Teraflops Research Chip and the Intel Single-chip Cloud computer. We will discuss one of the Xeon Phi variants, the Intel Xeon Phi 5110P. This is the model tested in this thesis. The 5110P features 60-cores at 1.053Ghz and 8GB GDDR5[8]. Each core is a fully functional x86-64 execution unit, with coherent L1 and L2 caches. A single core can handle 4 threads, which means the total number of threads the chip can handle efficiently is 240. The Xeon Phi is highly optimized for vector processing. Each core features a special Vector Processing Unit (VPU) with 512-bit SIMD instructions, as a replacement for the more commonly found Intel SSE, MMX and AVX instructions. With 512-bit SIMD instructions, vectors with up to 16 single and 8 double precision values can be processed in a single clock cycle.

Cores are interconnected with a high-speed bidirectional ring with up to 115 GB/sec bandwidth. The peak memory bandwidth is 320 GB/s.

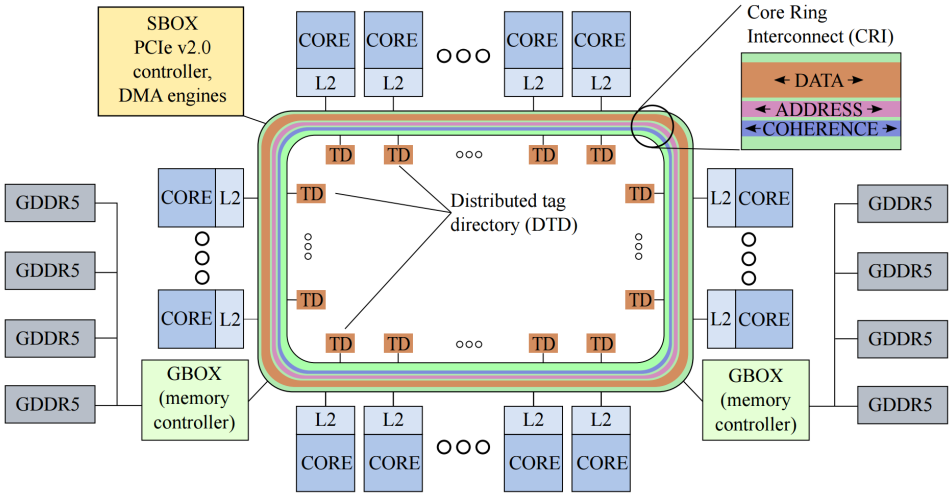

Figure 2.1: Intel Xeon Phi architecture overview [10, page 20]

Just like other accelerators, the Intel Xeon Phi coprocessor is connected to a host system with PCI Express. It runs a Linux-based operating system, allowing the user to access the coprocessor as a network node, with full TCP/IP capabilities. Multiple Xeon Phi's can be combined in a single system, with communication through the PCIe bus without intervention from the host.

We will not cover this use case in this thesis, but it is a useful way to use the Xeon Phi in clusters.

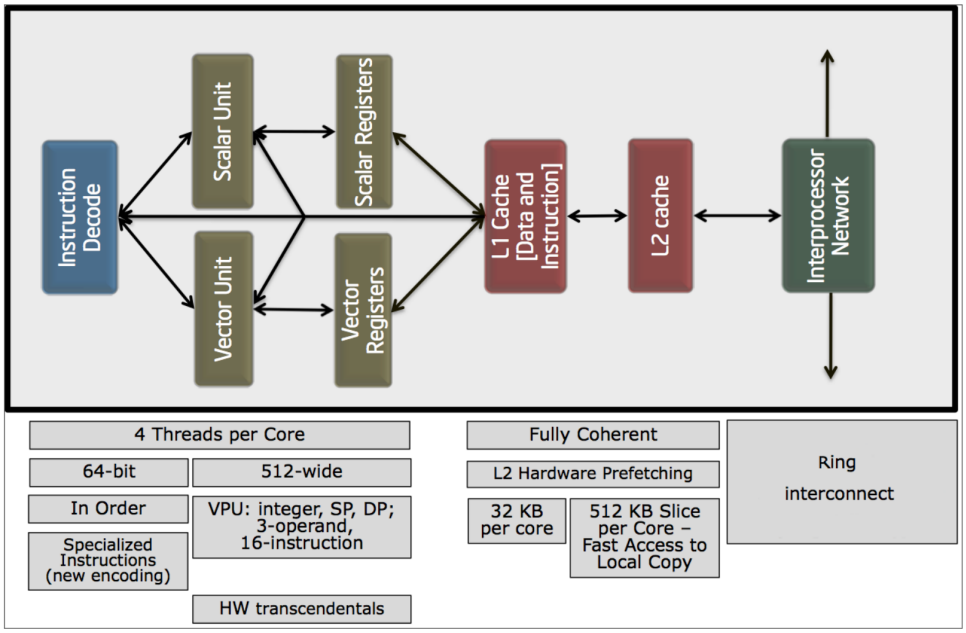

Figure 2.2: A diagram of the Xeon Phi core design [10, page 21]

Each of the 60 cores of the Xeon Phi is designed to support 4 threads. This means that to reach maximum efficiency on the Xeon Phi, we have to use 240 threads. We will demonstrate this with experiments. The Xeon Phi cores are less complex and powerful than those found in regular Xeon CPUs. The design is based on the Pentium processor codenamed P54C. It is a two-way superscalar architecture with an in-order pipeline. Modern Intel CPUs such as the X5650 are superscalar too, but also support *out-of-order* execution of instructions in the pipeline to improve performance even further.

Although these cores are significantly less powerful than Xeon cores, they feature a *Vector Processing Unit* with wide 512-bit registers for SIMD instructions. Intel has chosen to remove the SSE/AVX instructions that are normally found in Intel processors (up to 256-bit) for vector processing. The VPU also features a new specialized circuit called Intel Extended Math Unit (EMU), which implements certain transcendental operations such as reciprocal, square root and exponent. The VPU units are also capable of Fused Multiply-Add (FMA) and Fused Multiply-Subtract(FMS), which doubles the amount of theoretical vector operations per clock cycle from 16 to 32 (single precision). Because of these vector-related modifications, the Xeon Phi's instruction set is not binary compatible with standard x86-64 binaries. This means that code needs to be recompiled for the Xeon Phi. In table 2.1 the specifications of 5110P are shown, next to these

	Intel Xeon Phi 5110P	**Intel Xeon X5650**
Number of cores	60	6
Number of threads	240	12
Instruction set	64-bit	64-bit
Lithography	22 nm	32 nm
Base frequency	1.053 Ghz	2.66 GHz
Random access memory	8 GB GDDR5	Up to 288 GB DDR3
Max memory bandwidth	320 GB/s	32 GB/s
Max performance (DP)	1011 GFlops	124.8 GFlops
TDP	225 W	95 W

Table 2.1: Comparison of specifications[8, 9]

of the Xeon X5650. The X5650 CPUs are widely used in the *DAS4* cluster. We will compare the performance of the 5110P to a dual-socket setup of X5650 CPUs, so with 12 cores instead of 6. A remarkable difference in these specification is the theoretical maximum memory bandwidth, which is much higher(320 GB/s) on the Xeon Phi. This directly follows from the architecture, because the X5650 uses DDR3 RAM, while the 5110P features GDDR5. DDR3 RAM is optimized for low-latency access, while GDDR5 is optimized for high bandwidth at the expense of higher latency. This suggests that the Xeon Phi is beneficial to use for bandwidth-intensive algorithms.

A significant difference between the memory architecture of the Xeon Phi and GPUs is the cache coherency. Cache coherency means that changes in cache memory are synchronized with other cores that keep other copies of this data in their cache. The Xeon Phi 5110P has 30 MB of shared L2 cache, divided over 60 cores. This memory is kept coherent by the hardware ring interconnect. GPU architectures typically are not cache coherent. The lack of cache coherency in GPU architectures allows for greater scalability with simpler hardware and less overhead, because no data needs to be synchronized. On the other hand, the programming models for these architectures are often more restrictive to the developer. Fully coherent caches are found in regular CPUs, which require a more complex architecture but allow for more flexible programming models such as OpenMP.

The Xeon Phi is promising on paper, with high memory bandwidth and two teraflops of theoretical computational power. We will quantify the impact of these specifications in real-world situations, specifically with graph algorithms. When looking at the architecture, we can see some downsides. Because of the 60 cores at a relatively low 1.053 GHz frequency, we expect the synchronization overhead to be higher. Cache misses are more expensive than on the regular Xeon CPUs. It is clear that the single-core performance is relatively low, so to achieve good performance, it is essential to parallelize code as much as possible. However, many applications are not designed to be parallelized up to 240 threads. The well-known principle named *Amdahl's Law* comes to mind, which basically says that the speedup of a program using multiple cores in parallel computing is limited by the time needed for the sequential fraction of the program. Even more so because of the low single-core performance on the Xeon Phi, the sequential fraction of the program could be a bottleneck. The graph processing algorithms that we have implemented have a negligible sequential fraction, so we do not have to worry about this in our experiments.

2.2 Programming models

2.2.1 Programming languages

Because the instruction set architecture (ISA) of the Intel Xeon Phi is based on a subset of x86-64, traditional programming models can be used to program for the Xeon Phi. GPUs need to be programmed with tools such as CUDA/OpenCL, which are less flexible. For the Xeon Phi, existing knowledge and optimizations for regular CPUs can be reused on the Xeon Phi[6]. However, the Xeon Phi is not binary compatible with x86-64, so a special compiler is needed. Therefore the Xeon Phi is not compatible with all of the languages and tools used today. At the moment the Xeon Phi only works with the Intel compilers which support C, C++ and Fortran (`icc` and `icpc`).

2.2.2 Execution modes

The Xeon Phi communicates over PCI Express with the host CPU. It runs a Linux-based operating system, rather than a driver-based model often used for PCI Express connected cards. With a Linux host system, the Xeon Phi can be used in three different execution mode:

Offload mode
The program starts execution on the host system. During execution, regions of code can be *offloaded* to the Xeon Phi with custom compiler directives, supported by the Intel compiler.

This results in data being copied from the host to the Xeon Phi. It is also possible to use the coprocessor in parallel with host execution.

Native mode
A program executable is compiled for the Xeon Phi. The program is first copied to the Xeon Phi and then runs on the coprocessor independent of the host system. In this mode the coprocessor can be viewed as a node in the network.

Symmetric mode
Similar to native mode, but in this case the program runs on both the host and coprocessor concurrently. They can communicate over an inter-node messaging protocol such as MPI.

2.2.3 OpenCL

The Open Compute Language (OpenCL) is a framework for implementing parallel applications. It is similar to NVIDIA's CUDA. OpenCL is normally used for GPU programming, but Intel offers full support of OpenCL on the Xeon Phi. This is attractive to developers because existing GPU code written in OpenCL can be used without modifications and CUDA code can ported to OpenCL for the Xeon Phi with only modest effort.

2.2.4 OpenMP

The most commonly used tool for multithreading is OpenMP (Open Multi-Processing). It is considered the de facto industry standard for shared-memory multithreading in HPC. Code written with OpenMP can be executed on regular CPUs and Xeon Phi's with little to no modifications. We have used this technique for implementing the benchmarks.

2.2.5 Open MPI

Open MPI is a popular Message Passing Interface (MPI) implementation, supported by Intel for the Xeon Phi. This library is used in many of today's Linux clusters. OpenMP and Open MPI can be used together, where MPI is used for communication between nodes and OpenMP is used to parallelize code on a shared-memory single machine. By supporting Open MPI together with OpenMP and OpenCL, Intel basically supports all major programming models that are used in HPC today.

CHAPTER 3

Impact of architecture on algorithms

3.1 Benchmarking the Xeon Phi

We will describe a number of algorithms that we have used to analyze the performance of the Intel Xeon Phi 5110P and Intel Xeon X5650 setup. We have implemented these algorithms in C++ with OpenMP. We use the Xeon Phi in native mode for the experiments, as this is the most straightforward way to run code on the Xeon Phi and we are only interested in the performance of the Xeon Phi card itself compared to the regular Xeon. All code is compiled with the Intel C++ compiler `icpc`, version 2013.3.163.
These compiler flags are used:
`-std=c++11 -O3 -openmp`

It is important to note that the Intel C++ compiler will automatically try to vectorize loop constructs in the code, which is supported for both the Xeon and Xeon Phi. Unfortunately, the compiler can only vectorize simple loops that do not contain complex pointer dereferences. To compile an executable for native execution on the Xeon Phi, the flag `-mmic` is added, without any code changes. Being able to compile code for Xeon and Xeon Phi without much effort is one of the main selling points of the Xeon Phi. The results of our experiments will be shown in next chapter.

3.2 Memory bandwidth

It is interesting to see if the high memory bandwidth of the Xeon Phi helps to improve performance of graph algorithms. First we want to determine how much memory bandwidth we can achieve in practice. We allocate an array of 250 million integers (N), which equals 1 GB of memory. This is sufficiently large to use for the experiments as it far exceeds the total CPU cache size of both systems.
The integers in the array are accessed in these orders:

- Sequential
  Read/write the integers in ascending order: indices 0, 1, .. N - 1.
- Interleaved
  Read/write the integers in interleaved order: indices 0, 2, 4, 6, 8, followed by 1, 3, 5, 7, etc.
- Random
  Read/write the integers in random order. The indices consist of a randomly shuffled array containing the N (unique) indices.

We expect sequential memory access to be the fastest, followed by interleaved and random. Due to cache misses, random memory access will be a lot slower. In general, reading data is faster than writing. To achieve a high bandwidth on multi-core systems, we need to read/write with

multiple threads concurrently, which is why we run this experiment with a different number of threads.

3.3 Matrix multiplication

Matrix multiplication is used in many applications and can be optimized very well for multicore systems, because of the parallelism and vectorization that it allows for. In contrast to BFS and PageRank, matrix multiplication of dense matrices is not irregular. It should give a different perspective on the performance of the Xeon Phi 5110P in comparison to the Xeon X5650. The experiment is done in this way:

- We measure the time it took for multiplying two square matrices. Each data point represents the average of 10 executions.
- We have used three different matrix sizes: 3000x3000, 5000x5000 and 7000x7000. The matrices contain random single-precision floats. A matrix is stored in a 1D-array.
- For multiplying the matrices, we have used the function `cblas_sgemm(...)` provided by the Intel Math Kernel Library (MKL). MKL is highly optimized for our Xeon and Xeon Phi systems[7]. It implements multithreading and vector instructions for maximum performance.

We expect the Xeon Phi to perform well here, as matrix multiplication can fully exploit the *Vector Processing Units* and massive parallelism provided by the Xeon Phi. Matrix multiplication should be able to use the full 240 threads of the Xeon Phi and the 512-bit wide SIMD registers, whereas the Xeon X5650 handles only 24 threads (12 cores with HyperThreading) and has 128-bit SIMD registers.

3.4 Impact on algorithms

Irregular algorithms are often difficult to run efficiently on massively parallel hardware, because of the following characteristics:

- They often exhibit irregular memory access patterns, which are known to cause poor performance.
- The computational effort can be input dependent or dynamically varying. Therefore the distribution of work among threads cannot be planned in advance.

We will look at two (irregular) graph processing algorithms, breadth-first search (BFS) and PageRank. Both algorithms depend on a lot of memory accesses. These algorithms are relevant because graph processing is the core of many workloads in high-performance computing and is often used as a factor to judge performance of parallel systems[12].

Graphs are represented by edges and vertices. It is important to use an efficient data structure to store the graphs. Most graphs are sparse because the number of edges is much smaller than the maximum number of edges, that is an edges between every node. Storing a sparse graph as an *adjacency matrix* will have significant memory overhead because most of the entries in the matrix will be zero. We want to store the non-zero elements only, so we need an efficient sparse matrix format for our graphs. For the implementations of BFS and PageRank, we use the *Compressed Sparse Row* (CSR) format, which is an efficient format for storing sparse matrices if no mutability is required.

Figure 3.1 shows a graph with 5 vertices and 8 edges with the corresponding CSR representation. Two arrays are allocated. One array of size $|V| + 1$ is used store the offsets for accessing the neighbors which are stored in the second array of size $|E|$. All vertices are numbered starting from 0. For example, if we want to find the neighbors for vertex 2, we look up the first offset

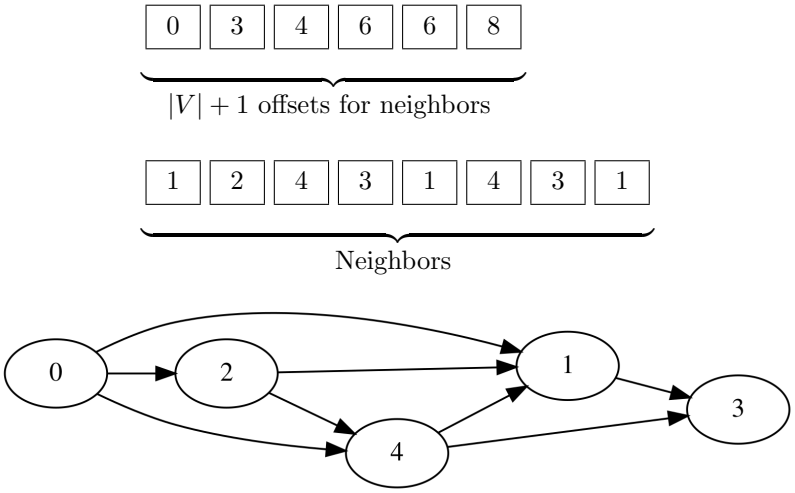

Figure 3.1: A simple graph and the corresponding CSR representation

which is 4 (at index 2), and the value at the subsequent index, this value is 6. We know that vertex 2 has $6 - 4 = 2$ neighbors located at indices 4 and 5. Using this representation, we can iterate over the vertices and neighbors of the graph efficiently while having a lower memory footprint compared to an adjacency matrix.

3.5 Breadth-first search (BFS)

Breadth-first search is a fundamental graph traversal algorithm. It traverses the nodes of a graph per level, starting at a chosen root node. It visits each neighbor node first before moving to the next level. BFS is important because it is a building-block for many applications. For example, it is commonly used to find the shortest paths in unweighted graphs. It is also used to test a graph for *bipartiteness* and for finding the *maximum flow* in a flow network.

There are multiple ways to implement BFS. Our implementation is a *hybrid* approach based on the paper *Direction-Optimizing Breadth-First Search* [1]. This is a combination of top-down and bottom-up BFS which will be explained below. We first look at the conventional *top-down* approach. The algorithm starts at a selected source node, and in each step of the algorithm, the frontier expands to the next level, visiting all vertices at depth d before moving to depth $d+1$. The frontier, i.e., the current level of vertices that is explored, is determined by an integer representing the depth of the vertex in the graph. This is an alternative to the queue-based approach, where a queue data structure is used to store the current frontier vertices. Using a queue in a multithreaded BFS was slower, because of the overhead introduced by accessing the queue concurrently.

During a step, each vertex in the frontier checks all of its neighbors to find out if any of them are unvisited. Each unvisited neighbor is added to the frontier by setting its depth value and marking it as visited. To parallelize this algorithm, the outer loop over all vertices is parallelized. The algorithm needs to be *level-synchronous*, in order for BFS to work correctly, i.e., threads can only work concurrently on a single level in the graph. The pseudocode for our implementation of a top-down BFS step is shown below (Algorithm 1). Note that in the parallel implementation of a top-down BFS step, marking a vertex visited is implemented by an *atomic* operation. This is required because multiple threads might be checking the same neighbor vertices, if neighbors are shared among vertices. To correctly update the frontier, each vertex should only be visited exactly once.
A large part of the work in top-down BFS is checking all edges in the frontier to find new unvisited vertices. If the frontier becomes large, the number of neighbors to examine is also much larger. In this case, the top-down approach can become inefficient and the bottom-up approach

```
for each vertex v in graph do in parallel
    if v is at current depth then
        for each neighbor n of v do
            if n is not visited then
                mark n visited;
                set depth of n to current depth + 1;
            end
        end
    end
end
```

Algorithm 1: Top-down breadth-first search step

```
for each vertex v in graph do in parallel
    if v is not visited then
        for each parent p of v do
            if p is at current depth then
                mark v visited;
                set depth of v to current depth + 1;
                break;
            end
        end
    end
end
```

Algorithm 2: Bottom-up breadth-first search step

comes into play.
In bottom-up BFS, we search for the next frontier vertices in the reverse direction, by using the reverse graph. Each unvisited vertex in the graph checks if one of the corresponding parents is contained in the frontier. The vertex is then added to the new frontier. This is efficient because once a vertex has found a parent that is also in the frontier, it does not need to check the rest of its parents. Another advantage of bottom-up BFS is that it eliminates the need for an atomic operation in a parallel implementation. There cannot be multiple threads writing to a vertex in bottom up, because a vertex only writes to itself.

The top-down approach checks all neighbors of every vertex in the frontier, while the bottom-up approach checks at most all the parents of every unvisited vertex. The bottom-up approach is only advantageous if the frontier is relatively large, in which case it will reduce the edge examinations compared to the top-down. If the frontier is small, it will be slower than top-down BFS. Therefore these two approaches are combined in the hybrid implementation. In our implementation, the algorithm chooses either a top-down or bottom-up step in every iteration based on the current frontier size. The decision is made with a threshold value.
We have compared our final implementation on a range of graphs(see table 3.1 below) on the Xeon Phi and Xeon setup. The results are shown in next chapter.

3.6 PageRank

The second irregular algorithm that we have implemented is PageRank. It is a ranking algorithm for graphs developed by Larry Page and Sergey Brin, the founders of Google[14]. Given a graph, the algorithm computes a score for each vertex to represent its importance. The algorithm is based on the idea of a "random web surfer", i.e., someone who randomly clicks on the links of web pages. Pages with a high probability to be viewed by the web surfer will get a higher score. The sum of all scores should be equal to 1.0 (or 100.0%). The output can be considered a probability distribution to represent the likelihood that a person will arrive at the vertex (page) in the graph.

Table 3.1: An overview of graphs used in our experiments

Graph name	**#Vertices**	**#Edges**	**Diameter**	**Type**
as-skitter	1,696,415	11,095,298	25	Autonomous system
web-google	875,713	5,105,039	21	Web graph
web-Stanford	281,903	2,312,497	674	Web graph
amazon0505	410,236	3,356,824	20	Product co-purchasing network
wiki-Talk	2,394,385	5,021,410	9	Communication network
roadnet-CA	1,965,206	2,766,607	849	Road network
roadNet-PA	1,088,092	1,541,898	786	Road network
roadNet-TX	1,379,917	1,921,660	1054	Road network
degree6-1M	1M	6M	-	Synthetic random
mesh1000x1000	1M	4M	1000	Synthetic

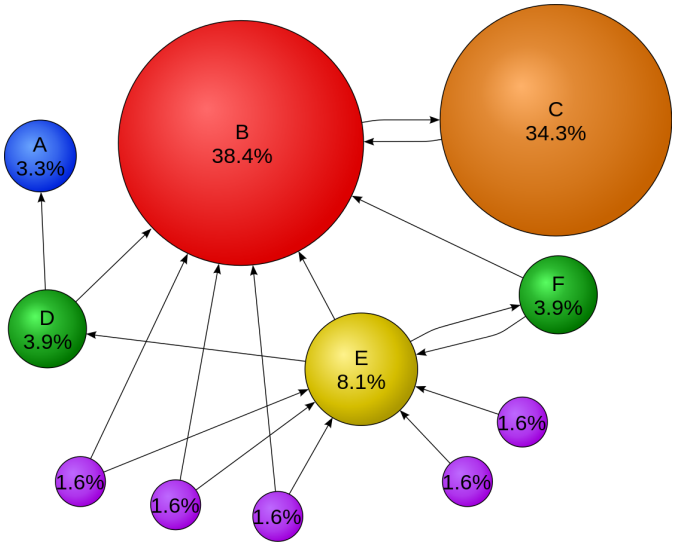

Figure 3.2: Example of a PageRank result[17]

PageRank can be implemented relatively simple as an iterative algorithm. Every iteration the PageRank value of each vertex will be updated. After a number of iterations the values converge to a stable value between 0 and 1. There are two ways of computing the PageRank values, push-based and pull-based. In the push-based implementation, every vertex "pushes" its current PageRank value to the outgoing edges. In the pull-based implementation, every vertex "pulls" the PageRank value from their incoming edges. We have implemented both of these variants. They are very similar, which is why we will only describe the pull-based variant below. The pull-based PageRank algorithm can be described in three steps:

1. Initialization: all vertices are initialized with a PageRank value of $1/|V|$ where $|V|$ equals the number of vertices in the graph. This represents the initial probability of picking a vertex from all vertices in the graph. After this step we run the following two steps a predefined number of times, until the PageRank distribution has stabilized.

2. Rank computation: each vertex computes the PageRank value as the sum of the PageRank values of the vertices from incoming edges. In this step, the vertex "pulls" the current PageRank value from all of its neighbors. Note that we use the reverse CSR graph representation to access these vertices.

3. Consolidation: we iterate over each vertex again, and apply a dampening factor to each PageRank value. This dampening factor d is a factor between 0 and 1, and is introduced to include the probability that a web surfer randomly stops following links and just moves to a new page (vertex). We use $d = 0.85$ in our implementation as this is the most commonly

chosen value. Furthermore, to make sure that the PageRank is still a valid distribution with a sum of 1.0, we need to add $(1.0-d)/|V|$ to each PageRank value. The final operation in this step is to divide each PageRank value by the number of outgoing edges. By doing this, the PageRank value is evenly distributed among the neighbors.

```
for vertex in reverse graph do in parallel
    vertex.pagerank = (1/|V|) / vertex.degree;
end
for N iterations do
    for vertex in reverse graph do in parallel
        vertex.newpagerank = 0;
        for each neighbor of vertex do
            vertex.newpagerank += neighbor.pagerank;
        end
    end
    for vertex in reverse graph do in parallel
        vertex.pagerank = applyDampening(vertex.newpagerank);
        if not last iteration then
            vertex.pagerank = vertex.pagerank / vertex.degree;
        end
    end
end
```

Algorithm 3: Pseudocode for our parallel pull-based PageRank implementation

The downside of the push-based approach is that atomic operations are needed when updating the PageRank values. This is necessary as there can be multiple threads writing to the state of the same vertex, to distribute the PageRank values. We do not need any atomic operations in pull-based implementation, which is why we expect this one to perform better. Atomic operations cause memory synchronization overhead resulting in lower performance. We have benchmarked our pull-based PageRank implementation as well our push-based PageRank implementation on the graphs shown in table 3.1.

CHAPTER 4

Results and discussion

4.1 Memory bandwidth

For this experiment we have measured the time it took to read or write an array in the orders described in this previous chapter, with 4, 8, 12, 16, 24, 60, 120, 180 and 240 threads. The results are converted to gigabytes per second (10^9 bytes/s). Every data point represents the average of 10 runs. The error bars in graphs represent the standard deviations of the measurements. We measured sequential, interleaved and random read performance on the Xeon Phi 5110P and Xeon X5650. Figure 4.1 shows that the Xeon Phi reaches the maximum random-read performance of 2.2 GB/s with 240 threads. This is almost a 2x speedup compared to the Xeon X5650 setup.

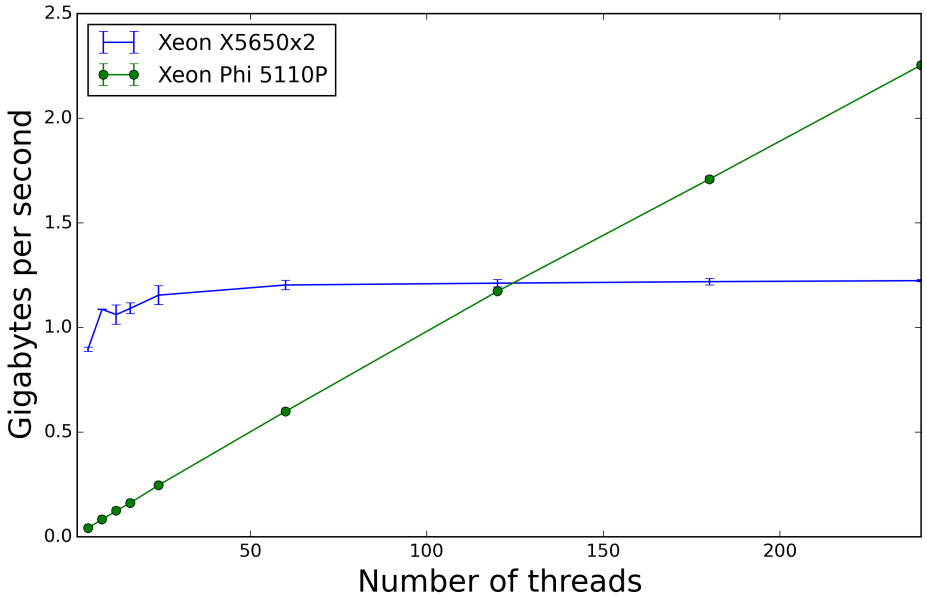

Figure 4.1: Read performance in random order

The sequential bandwidth is much higher, as can be seen in figure 4.2 on the next page. The 5110P reaches a sequential read bandwidth of 78 GB/s in our best case, with an average of only 51 GB/s. Although this is significantly below the 320 GB/s theoretical maximum, the Xeon Phi still has a large advantage compared to the Xeon X5650 dual socket setup.

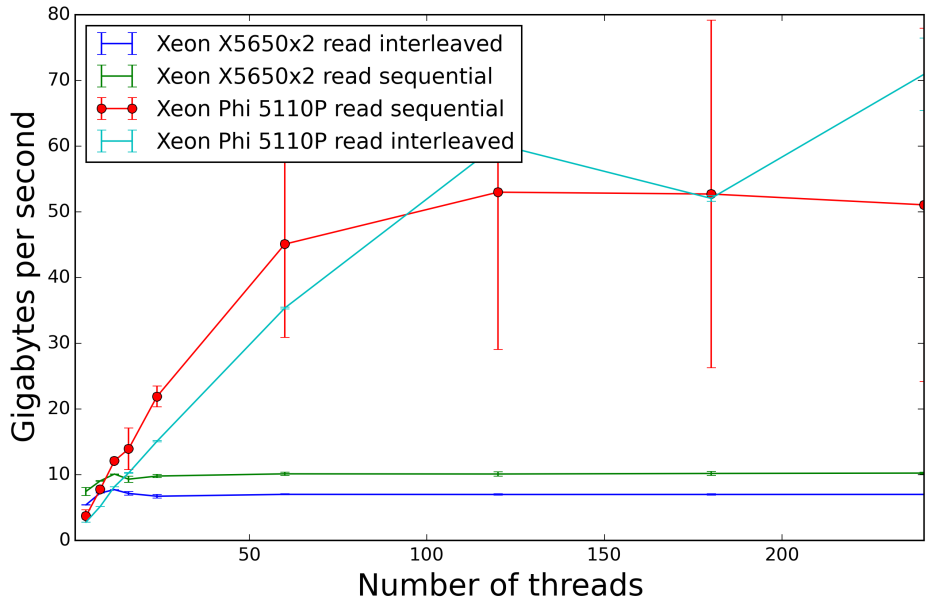

Figure 4.2: Read performance in interleaved and sequential order

We observe a much larger standard deviation in the measurements of sequential read on the Xeon Phi(the red line) compared to the other measurements. Figure 4.2 also shows that on the Xeon Phi, the interleaved memory bandwidth performance is surprisingly slightly higher than the sequential performance, on average. The write benchmarks show similar results. The Xeon Phi is again much faster than the X5650 setup, but overall slower than in the read benchmark. The Xeon X5650 reaches a maximum random write performance with 24 threads a 0.8 GB/s, The Xeon Phi reaches 1.9 GB/s with 240 threads.

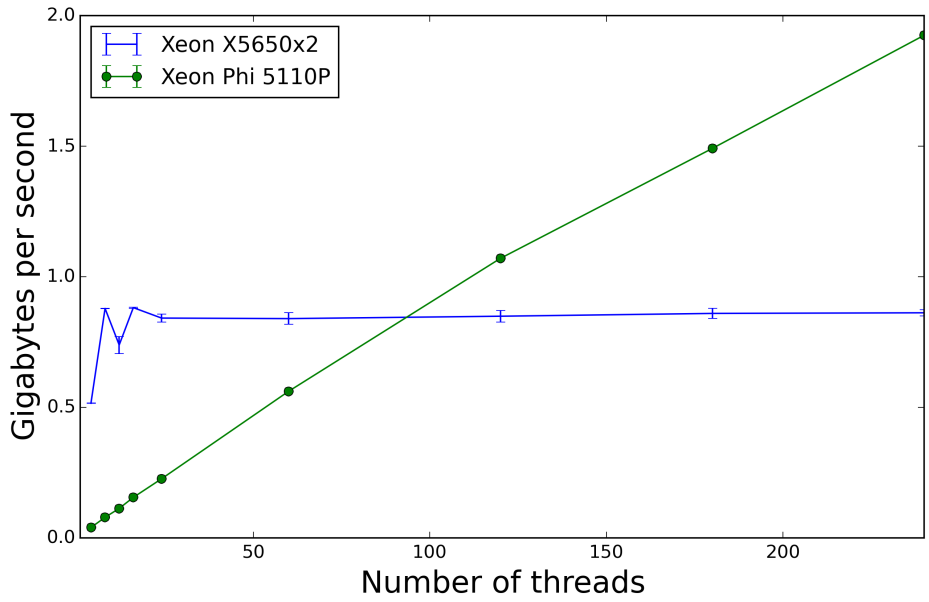

Figure 4.3: Write performance in random order

In the sequential/interleaved write benchmarks(figure 4.4), we see results similar to the read benchmarks but with a lower bandwidth. The Xeon Phi achieves 22 GB/s peak sequential write bandwidth, still higher than the X5650 at 8 GB/s, but the relative difference is smaller than in the read benchmarks. Table 4.1 shows a summary of the average bandwidth measurements at the optimal thread count for both systems. Overall, our results are in line with prior work by

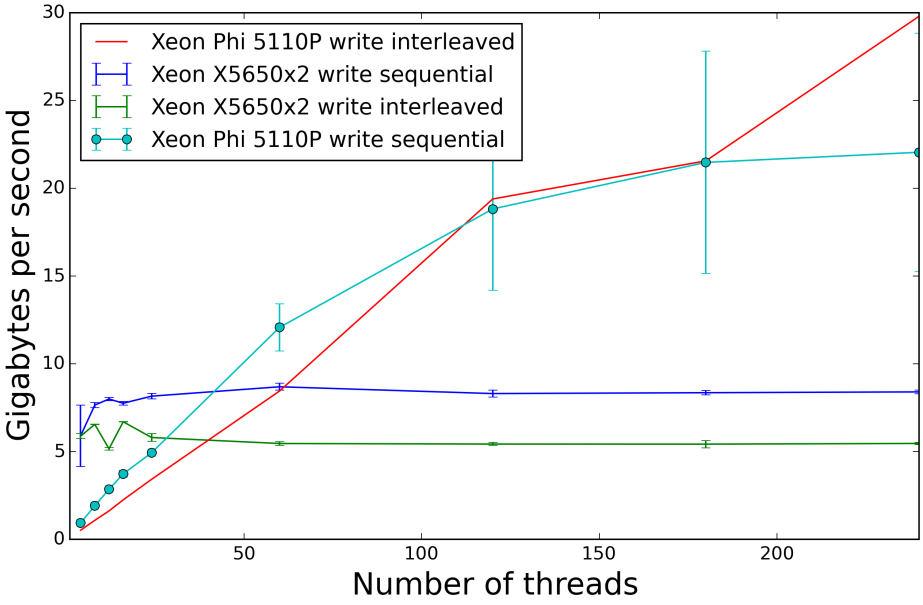

Figure 4.4: Write performance in interleaved and sequential order

	Xeon Phi 5110P	**Xeon X5650**
Random read	2.25 GB/s	1.15 GB/s
Random write	1.92 GB/s	0.84 GB/s
Interleaved read	70.92 GB/s	6.70 GB/s
Interleaved write	29.79 GB/s	5.79 GB/s
Sequential read	51.05 GB/s	9.77 GB/s
Sequential write	22.04 GB/s	8.15 GB/s

Table 4.1: Average measured bandwidth with 240 threads(5110P) and 24 threads(X5650)

Ana Varbanescu et al[4]. The achieved bandwidth is below the theoretical peak of 320 GB/s, but still significantly higher than the Xeon setup. It is clear that we need to use 240 threads on the Xeon Phi to obtain the highest bandwidth. More threads are able to send more memory requests to the memory controller, which helps to saturate the memory channels and ring interconnect.

According to our results, the Xeon Phi has a significant advantage in sequential and random memory bandwidth compared to the Xeon setup. Memory-intensive graph algorithms should benefit from this.

4.2 Matrix multiplication

We have benchmarked matrix multiplication with three different matrix sizes and each experiment was repeated five times. Our results in figure 4.5 show that matrix multiplication scales well up to 12 threads for the Xeon X5650 and 60 threads on the Xeon Phi. It is also clear that the Xeon Phi is much faster. Figure 4.6 shows the performance for the different matrix sizes

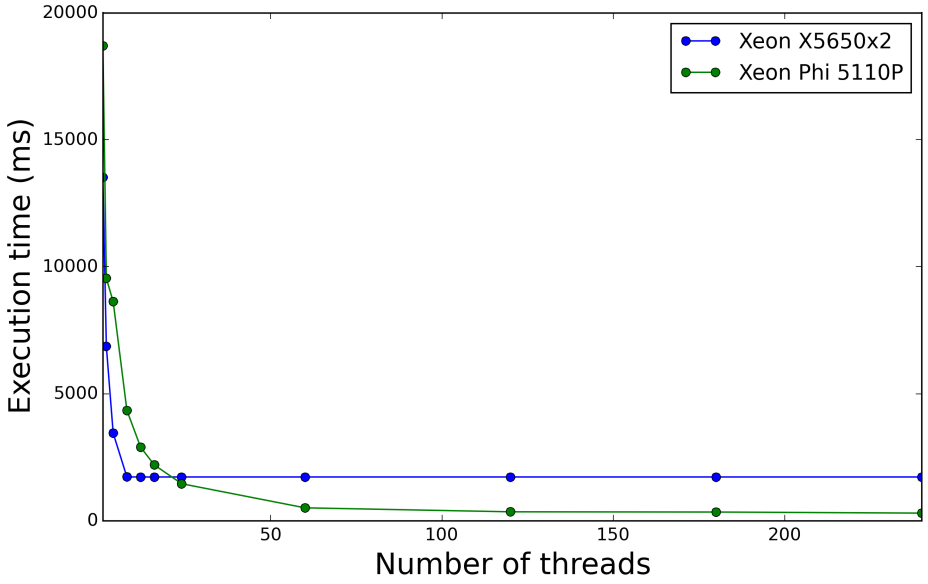

Figure 4.5: Matrix multiplication with 5000x5000 matrices

with the optimal number of threads for both systems. The standard deviation, if large enough, is shown as error bars in figure 4.6. The Xeon Phi performance is impressive here, being about 5 times faster than the Xeon. The Xeon Phi proves to be powerful with a vectorizable and regular algorithm such as matrix multiplication. Although the Xeon Phi delivers impressive results in

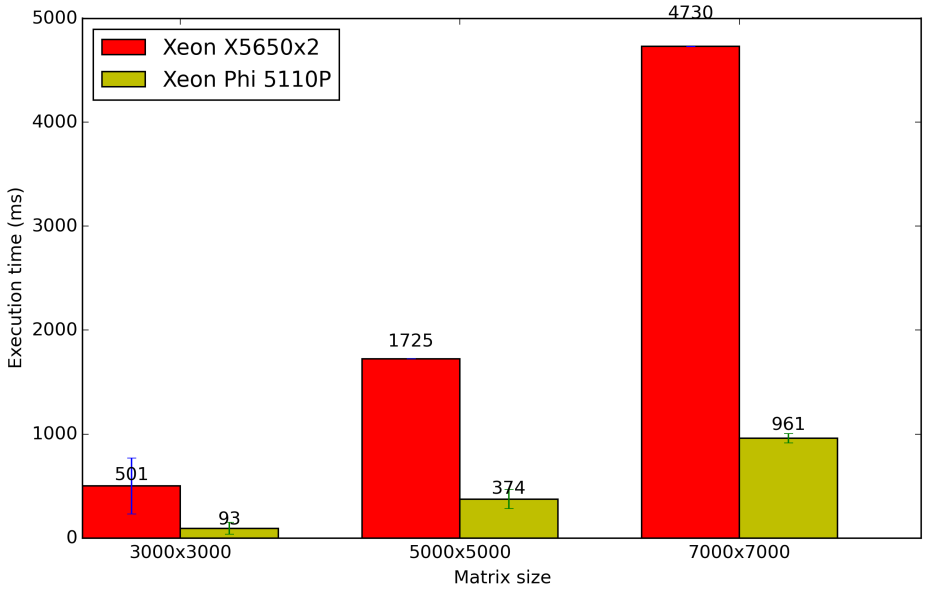

Figure 4.6: Matrix multiplication with Intel MKL

matrix multiplication, graph processing algorithms such as breadth-first search and PageRank

will not benefit from this, as these algorithms are irregular and not vectorized.

4.3 Breadth-first search (BFS)

First, to determine if our BFS implementation scales well across threads, we have measured the execution time with a different number of threads, to be exact 1, 2, 4, 8, 12, 16, 24, 60, 120, 180 and 240. For this initial test we have used a randomly generated graph with 131K edges and 8K vertices. The results are shown in figure 4.7. We can conclude that our code scales up to the number of physical cores in our test systems, 12 threads for the X5650 dual-socket setup and 60 for the Xeon Phi. From that point adding more threads only decreases the performance.
To gain insight into the performance of the Xeon Phi 5110P compared to the X5650, we benchmarked our BFS implementation with the optimal number of threads for both systems and with a range of graphs (see table 3.1). We did a full BFS traversal 100 times by picking each of the

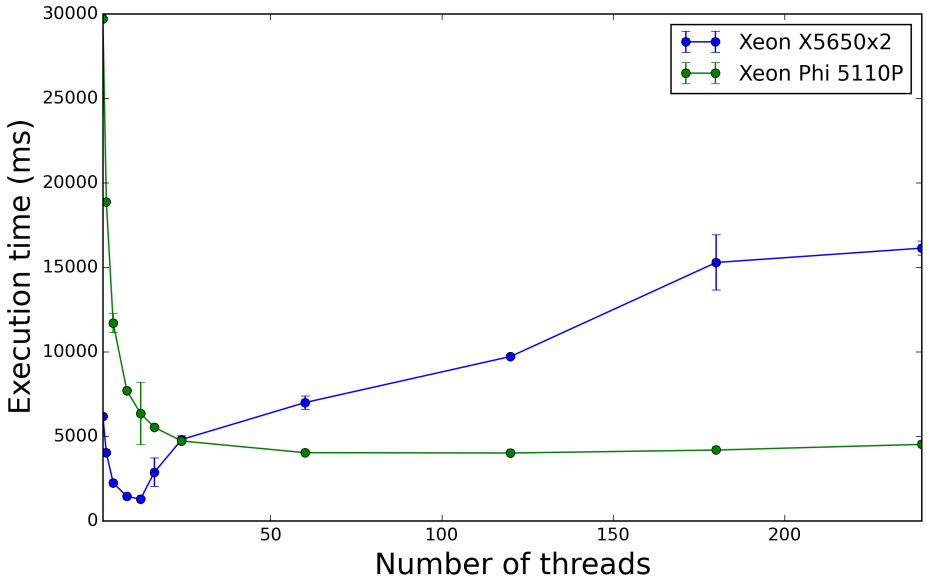

Figure 4.7: Scaling of our BFS implementation on RMAT generated graph with 131K edges, 8K vertices, with increasing number of threads

first 100 nodes of the graph as the root vertex. This process is repeated 10 times and the average execution time is shown in figures 4.8 and 4.9. The standard deviation in this experiment was negligibly small.
Figure 4.8 shows that the Xeon Phi is faster in 4 out of 5 cases. These include small-world graphs such as *web-Google* and *web-Stanford*. With high-diameter graphs such as *roadNet-CA*, *roadNet-PA* and *roadNet-TX*, the Xeon Phi consistently outperforms the Xeon (see figure 4.9).

It is important to note that BFS has a low compute-to-memory access ratio. It involves minimal vertex processing. BFS mostly consists of memory lookups without significant computational work. As a result, the performance will be largely dependent on (random) memory access performance. Based on our previous memory benchmarks, we know that the Xeon Phi does reasonably well in this regard. We think this is the main reason that the Xeon Phi is able to outperform the Xeon in BFS.

4.4 PageRank

We have benchmarked both our PageRank implementations, each measurement is repeated 10 times and averaged. Our pull-based variant is significantly faster than push-based, as we expected. This implementation also scales well across many threads, as can be seen in figure 4.10.

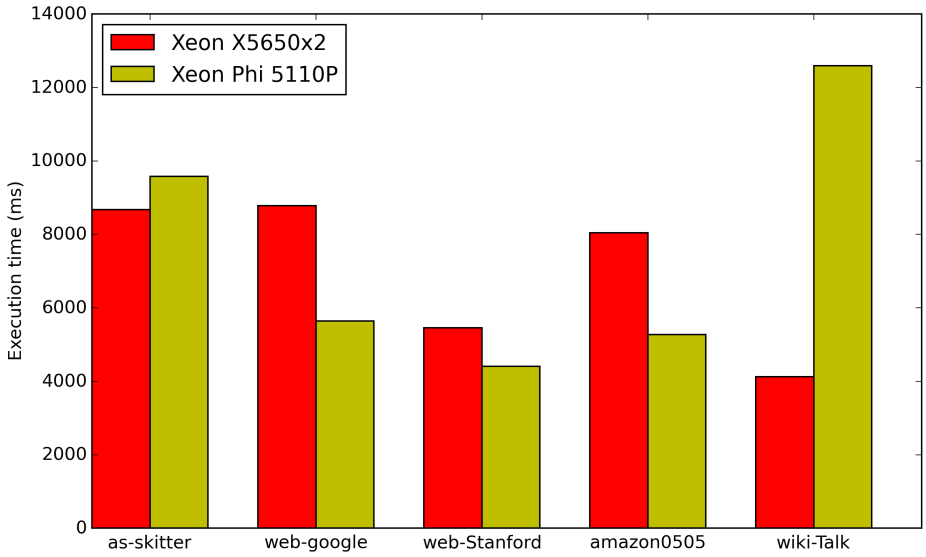

Figure 4.8: BFS performance on a range of real-world graphs

Nevertheless, the traditional Xeon was faster than the Xeon Phi in all our PageRank benchmarks, both with push and pull-based implementations. Figure 4.11 and 4.12 show the normalized execution times of our push-based and pull-based implementation, respectively. In our best case, the Xeon Phi was able to reach 69% of the Xeon performance, but was slower than that in most cases. We have also experimented with the road infrastructure graphs, but the results are similar and in favor of the Xeon.

In contrast to BFS, the Xeon Phi is 1.45 to 5.8 times slower than the Xeon in PageRank performance. Taking a look at the PageRank algorithm, we notice that PageRank is more compute-intensive than BFS. Every iteration in PageRank traverses all vertices, and for every vertex, the sum of all neighbor PageRank values is computed. In terms of reads and writes, we need at least N reads and N writes per vertex for PageRank, where N is the number of neighbors. With BFS we only need to write if a vertex should be added to the frontier. The Xeon Phi might suffer from memory contention, as multiple threads are accessing common neighbors. We suppose PageRank cannot take advantage of the increased memory bandwidth and parallelism of the Xeon Phi, and suffers from the slower, in-order core architecture. Overall, the Xeon Phi's architectural trade-offs cause worse PageRank performance than we initially expected.

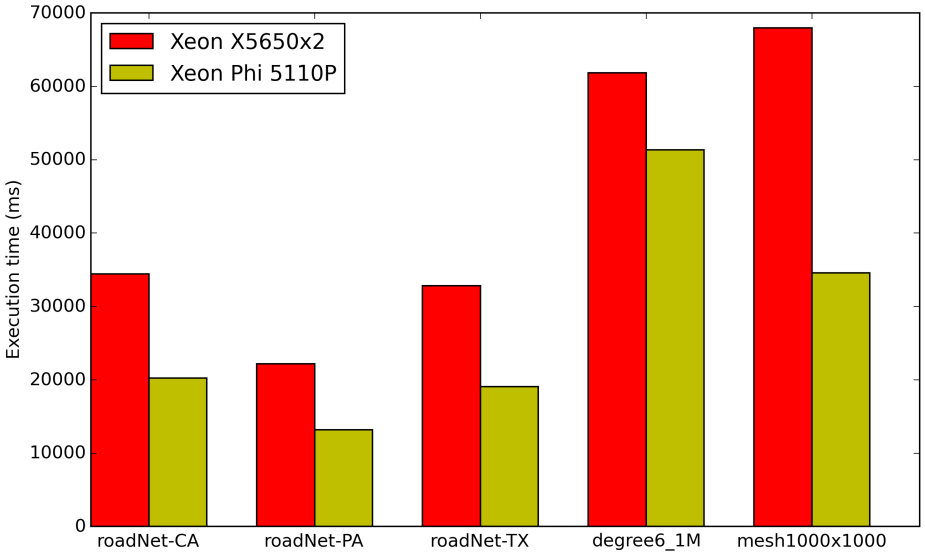

Figure 4.9: BFS performance on high-diameter, low-degree graphs

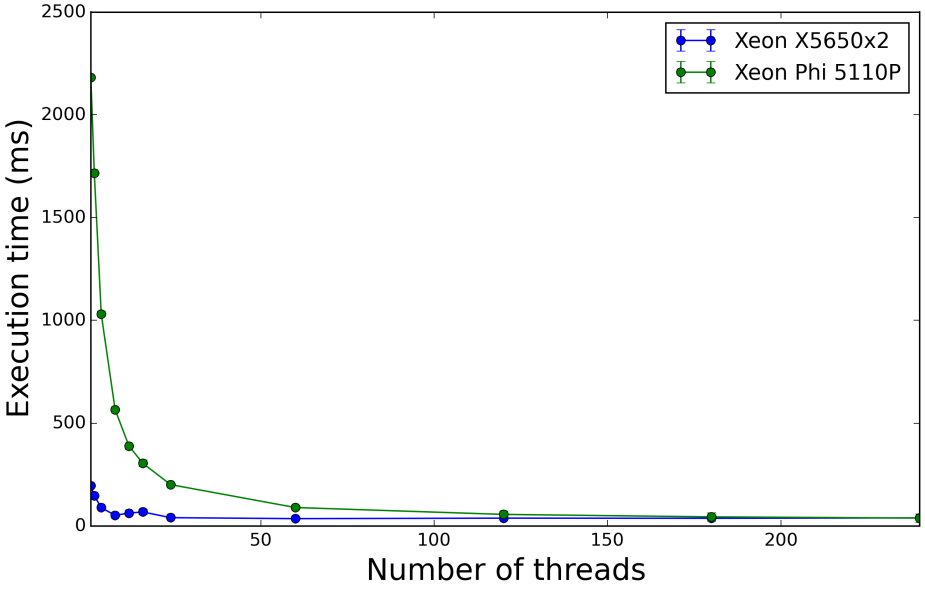

Figure 4.10: Parallel Pull-based PageRank implementation on randomly generated graph with 250K edges, 50K vertices, with different number of threads

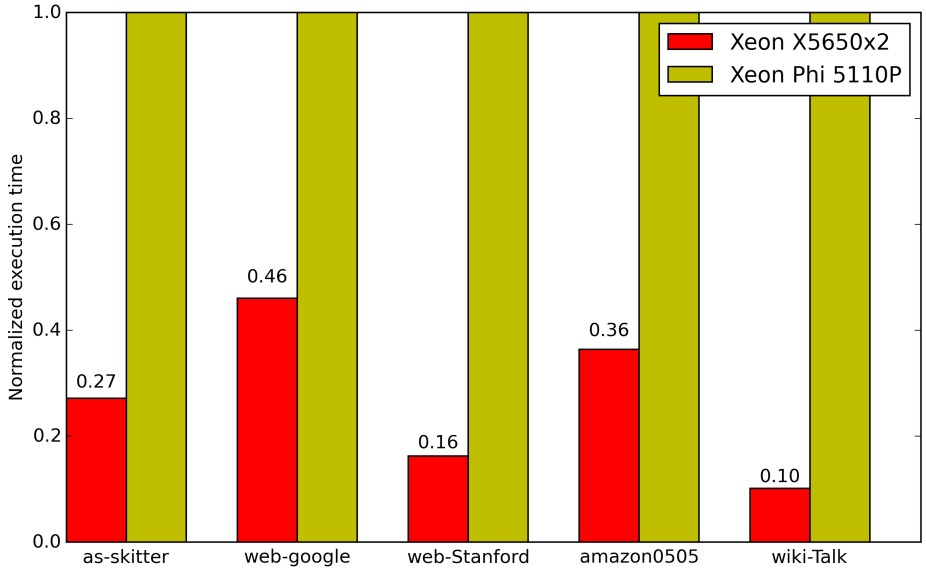

Figure 4.11: Parallel Push-based PageRank performance on real-world graphs

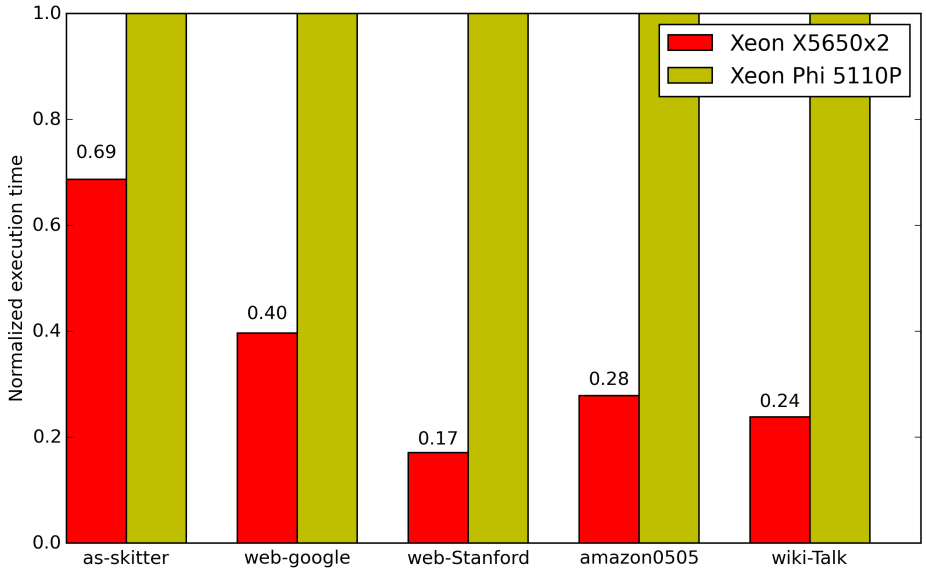

Figure 4.12: Parallel Pull-based PageRank performance on real-world graphs

CHAPTER 5

Related work

Prior studies by Ana Varbanescu et al.[3, 4] have investigated the performance of the Intel Xeon Phi by using microbenchmarks to measure the peak performance, as well as a medical imaging application as a real-world case study. They conclude that it takes lots of effort on parallelization and optimizations to achieve a high performance on the Xeon Phi. Their results are in line with ours, but their work does not consider the irregular algorithms discussed in this thesis.

Saule et al.[15] have extensively reviewed the performance of sparse matrix multiplication kernels on the Xeon Phi. They have benchmarked the Xeon Phi performance for multiplication of a sparse matrix with a dense vector. Their experiments show that the Xeon Phi's sparse kernel performance is even better than that of modern Xeon CPUs and GPUs. However, sparse matrix multiplication is compute-intensive, which is different from our memory-intensive benchmarks.

In [16], the authors compare the performance of the Xeon Phi, GPU and CPU on image analysis algorithms. Their research involves a number of algorithms that are used in image analysis. They conclude that GPUs are significantly faster for accessing irregular data than the Xeon Phi. However, the GPU required a custom CUDA implementation, whereas the Xeon Phi could reuse the same Xeon code.

Chen et al.[2] have developed a framework for exploiting SIMD instructions in graph processing on the Xeon Phi. Their experiments also include PageRank and BFS implementations, but their analysis focuses on the framework performance and graph partitioning strategies, rather than the performance of the Xeon Phi itself.

A paper by Intel[5] demonstrates the potential of the Xeon Phi in dense linear algebra operation, which involve mostly regular algorithms, in contrast to the graph algorithms discussed in this work.

CHAPTER 6

Conclusion

In this thesis, we have analyzed the performance of irregular algorithms on the Intel Xeon Phi. We have looked at the potential architectural advantages of the Xeon Phi compared to competing products such as conventional Xeon CPUs and GPU accelerators. To gain insight in practical performance, we implemented benchmarks in C++ with OpenMP for the Xeon and Xeon Phi. First, we determined if the Xeon Phi's specifications can be achieved in practice. We measured the memory bandwidth of the Xeon Phi. We looked at matrix multiplication, which fully utilizes the parallelism and vector instructions of the Xeon Phi, to assess the best case computational performance. Finally, we implemented and evaluated two key graph algorithms on the Xeon Phi, BFS and PageRank.

We can confirm that the Xeon Phi holds a significant advantage in memory bandwidth over the Xeon X5650 dual-socket setup. Although we could not reach the theoretical maximum of 320 GB/s, we measured a 7 times higher peak bandwidth on the Xeon Phi compared to the Xeon. We have shown that the Xeon Phi is able to perform particularly well in matrix multiplication, a regular and vectorizable algorithm. The Xeon Phi achieved a 5 times speedup compared to the Xeon.

Due to random memory access patterns and workload imbalance, graph algorithms are harder to run efficiently on the Xeon Phi, as demonstrated by our breadth-first search and PageRank benchmarks. In all cases, the PageRank implementations performed worse on the Xeon Phi. With breadth-first search our results are different and the Xeon Phi is faster in most cases. With small-world web graphs as well as with a set of high-diameter graphs, the Xeon Phi had the edge over the Xeon.

Based on our results, we conclude that the Xeon Phi is able to perform well with workloads that have a low compute-to-memory access ratio such as BFS. We suppose the Xeon Phi's architectural trade-offs are unfavourable with algorithms where the non-vectorizable computational phase dominates, such as PageRank. These algorithms are more sensitive to single-core performance, which is a weakness of the Xeon Phi. The underwhelming performance in PageRank could also be caused by the higher cost of cache misses on the Xeon Phi, as the PageRank algorithm needs significantly more reads and writes per vertex than BFS. However, a more exhaustive review of algorithms is required to verify this statement.

In future work, it would be useful to examine the performance with applications that combine irregular data, vectorization and parallelism, such as collaborative filtering[11]. Furthermore, to extend our results, it would be valuable to include GPU implementations in our comparison. Although GPUs do not offer the flexibility of the Intel Xeon Phi, they have become popular in high-performance computing.

Bibliography

[1] Scott Beamer, Krste Asanović, and David Patterson. "Direction-optimizing breadth-first search". In: *Scientific Programming* 21.3-4 (2013), pp. 137–148.

[2] Linchuan Chen et al. "Efficient and simplified parallel graph processing over CPU and MIC". In: *Proceedings of the 2015 IEEE International Parallel & Distributed Processing Symposium, IPDPS*. Vol. 15.

[3] Jianbin Fang et al. "An empirical study of Intel Xeon Phi". In: *arXiv preprint arXiv:1310.5842* (2013).

[4] Jianbin Fang et al. "Test-driving Intel Xeon Phi". In: *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*. ACM. 2014, pp. 137–148.

[5] Alexander Heinecke et al. "Design and implementation of the Linpack benchmark for single and multi-node systems based on Intel® Xeon Phi coprocessor". In: *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE. 2013, pp. 126–137.

[6] Intel. *An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors*. 2012. URL: `http://download.intel.com/newsroom/kits/xeon/phi/pdfs/overview-programming-intel-xeon-intel-xeon-phi-coprocessors.pdf`.

[7] Intel. *Intel Math Kernel Library (MKL)*. `https://software.intel.com/en-us/intel-mkl`. [Online]. 2015.

[8] Intel. *Intel Xeon Phi Coprocessor 5110P*. `http://ark.intel.com/nl/products/71992`. [Online]. 2015.

[9] Intel. *Intel Xeon Processor X5650*. `http://ark.intel.com/nl/products/47922`. [Online]. 2015.

[10] Vadim Karpusenko and Colfax International Andrey Vladimirov. *Parallel Programming and Optimization with Intel Xeon Phi Coprocessors*. `http://www.umbc.edu/hpcf/user-resources/Colfax-training.pdf`. [Online]. 2014.

[11] Yehuda Koren, Robert Bell, and Chris Volinsky. "Matrix factorization techniques for recommender systems". In: *Computer* 8 (2009), pp. 30–37.

[12] The Graph 500 List. *Graph 500 Benchmark 1*. `http://www.graph500.org/specifications`. [Online]. 2015.

[13] Top 500 supercomputer list. *Tianhe-2 (National Supercomputer Center in Guangzhou)*. `http://www.top500.org/system/177999`. [Online; accessed 27-May-2015]. 2015.

[14] Lawrence Page et al. *The PageRank citation ranking: bringing order to the Web*. 1999.

[15] Erik Saule, Kamer Kaya, and Ümit V Çatalyürek. "Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi". In: *Parallel Processing and Applied Mathematics*. Springer, 2014, pp. 559–570.

[16] George Teodoro et al. "Comparative Performance Analysis of Intel Xeon Phi, GPU, and CPU". In: *arXiv preprint arXiv:1311.0378* (2013).

[17] Wikipedia. *Numeric examples of PageRanks in a small system*. 2009. URL: `https://commons.wikimedia.org/wiki/File:PageRanks-Example.svg`.