Karlsruhe Institute of Technology

Parallel Graph Algorithms on the Xeon Phi Coprocessor

Master Thesis of

Dennis Felsing

At the Department of Informatics
Institute of Theoretical Computer Science
Parallel Computing Group

Reviewer:	Juniorprof. Dr. Henning Meyerhenke
Advisor:	Moritz von Looz

Duration: 2015-02-20 – 2015-08-19

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association www.kit.edu

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text, and have followed the rules of the KIT for upholding good scientific practise.

Karlsruhe, 2015-08-19

. . . .

(**Dennis Felsing**)

Abstract

Complex networks have received interest in a wide area of applications, ranging from road networks over hyperlink connections in the world wide web to interactions between people. Advanced algorithms are required for the generation as well as visualization of such graphs.
In this work two graph algorithms, one for graph generation, the other for graph visualization, are studied exemplarily. We detail the work of adapting and porting the algorithms to the Intel Xeon Phi coprocessor architecture. Problems in porting real software projects and used libraries are encountered and solved. Memory allocations turned out to be a major problem for the graph generation algorithm. The limited memory of the Xeon Phi forced us to offload chunks of the data from the host system to the Xeon Phi, which impeded performance, eliminating any significant speedup.
The data sets consisting of at most 365 000 edges for the graph visualization algorithm fit into the Xeon Phi's memory easily, which simplified the porting process significantly. We achieve a speedup for sparse graphs over the host system containing two 8-core Intel Xeon (Sandy Bridge) processors. While the hot inner loop by itself can utilize the 512-bit vector instructions of the Xeon Phi, the benefit disappears when embedded in the more complicated full program.

Zusammenfassung

Komplexe Netzwerke finden in vielen Bereichen Anwendung, von Straßennetzen über Hyperlink-Verbindungen im World Wide Web bis zu Interaktionen zwischen Personen. Moderne und komplexe Algorithmen werden benötigt um derartige Graphen sowohl zu generieren als auch zu visualisieren.
In dieser Masterarbeit werden zwei Graphalgorithmen exemplarisch analysiert, einer zur Graphgenerierung, der andere zur Graphvisualisierung. Auf den Prozess der Adaption und Portierung der Algorithmen auf den Xeon Phi Koprozessor gehen wir dabei genauer ein. Probleme treten bei der Portierung existierender Softwareprojekte und ihrer benutzten Libraries auf und wir beschreiben deren Behebung.

Acknowledgements

I am grateful to Moritz von Looz and Juniorprof. Henning Meyerhenke for the chance to work on this master thesis with the Xeon Phi coprocessor, and advising and supporting me throughout the time I worked on it.

My deepest gratitude goes to my parents for supporting me throughout my life.

Finally I'm thankful to my friends. Maybe one of them will read this one day.

Contents

CHAPTER 1

Introduction

1.1. Motivation

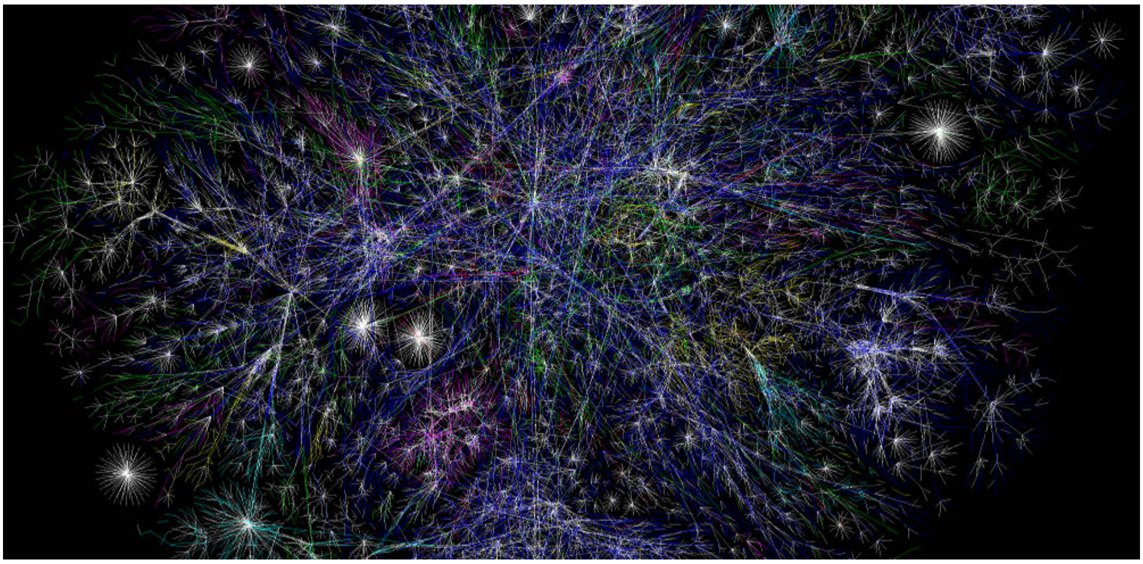


Figure 1.1.: Graph representing a part of the Internet [1]

Graphs and graph algorithms have become increasingly relevant today. With more data surrounding us every day, it is of interest in a wide range of areas to abstract this data and make sense of it.

Many relationships can be represented as complex networks. For example the relationships between people form complex social networks, while the interlinked structure of hyperlinks has become the namesake of the World Wide Web (see Figure 1.1). Among others, complex networks are used in research in the areas of statistical physics, protein interactions, cell biology and brain topology [2, 3, 4]. These complex networks can be worked with, studied and understood using graph algorithms.

Graph Generation

When working with large networks based on real data, a number of problems can occur:

For privacy and confidentiality reasons it is often not possible to share the data that has been used in research, especially when it concerns actual people. Additionally, the sheer size of some data sets can be a hindrance to sharing them with other researchers and making them widely available. This can be solved by providing a graph generator in combination with a set of parameters, which can be used to create realistic complex networks for the specific use case. Ideally such graph generators provide high performance, so huge graphs can be created on the fly and used directly to independently verify some research.

While real-world data is the gold standard for realistic graphs, there still is a great use for generating complex networks which are merely similar to reality, but are not based on any real data. Generating graphs has the additional benefit of being able to scale the complex network and experimentally test the scalability of the algorithm that is being researched for smaller and bigger graphs than can be obtained from real-world data sets.

Graph Drawing

Drawing graphs is the problem of representing them in a pictorial layout, making their features more easily perceptible for humans [5]. Furthermore graph drawing can be used as a preliminary step in other applications such as graph partitioning [6].

A common approach to graph drawing is the simulation of physical processes such as the forces of springs within all vertices of a graph, called a *full stress model*. This model can be used for smaller graphs to achieve a graph drawing of high quality. While it offers good results for graphs with fewer than 10000 vertices, its performance does not scale to big graphs. Instead we are interested in a solution that can be parallelized better, while still approximating the low stress metric of the full stress model.

Computation

While data sets are continually growing larger, performance improvements of the general computation architecture have slowed down in the last 10 years. Until then the simplest way of gaining performance has been the continual increase in clock frequency of the processor. The stagnation of CPU clock frequency in the last decade is shown in Figure 1.2.

Instead of increasing the clock frequency of a single processor, nowadays multiple processing cores are used. While the increases in frequency sped up existing programs without any changes to them, the parallelization of computation requires software to be designed using parallel programming techniques.

Modern GPUs (Graphical Processing Units) consist of thousands of simple compute units that can be used to solve general purpose problems in parallel. While CPUs have been slowing down in their performance increases, GPUs are becoming massively more parallel with each iteration. The general purpose programming of a GPU requires specific programming technologies and models such as CUDA or OpenCL, which makes a GPU more difficult to program for than a CPU.

A parallel alternative to GPUs is offered in the form of the Intel Xeon Phi coprocessor, which can be installed as an accelerator card in the same way as a GPU. Instead

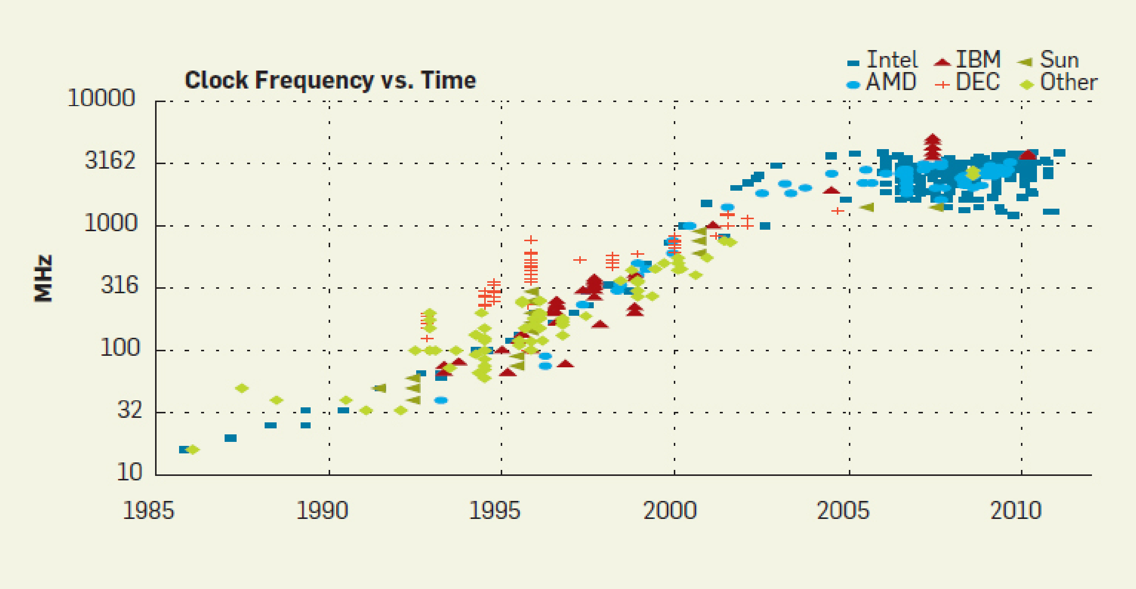


Figure 1.2.: Development of CPU clock frequency since the 1980s [7].

of the thousands of small cores of a GPU, and the up to 20 large and powerful cores of a CPU, the Xeon Phi features a compromise: About 60 cores at a low frequency, but featuring large multithreading and vector processing capabilities.

GPUs have been studied widely in the implementation of massively parallel algorithms. For some algorithms such as image processing, GPUs are a natural fit because of the huge number of weak computation units and streaming memory access. Graph algorithms on the other hand have been challenging to implement efficiently on GPU architectures. They rely on largely irregular data accesses and might be an area where the Xeon Phi's different architecture can show its advantages.

1.2. Contribution

We have ported two existing non-trivial graph algorithms to the Xeon Phi and optimized them.

This thesis presents an examination of the Intel Xeon Phi coprocessor architecture and the ways this architecture can be used in writing new software and porting existing implementations.

The general ideas for both studied algorithms, for the generation of massive complex networks as well as graph drawing using graph clustering, are explained. We port the extensive implementations of both algorithms to the Xeon Phi and describe the problems that were encountered as well as their solutions. Optimizations for the algorithms are offered and evaluated, as well as the performance compared to a two-socket Intel Xeon system.

We found limitations with the offloading that is required for large data sets and caused huge performance decreases that could only partially be alleviated. We present two implementations of algorithms that both scale well on the Xeon Phi. The graph drawing algorithm is able to surpass the host system's performance, especially for sparse graphs.

1.3. Notation and Preliminaries

A *graph* $G = (V, E)$ consists of a set of *vertices* V and a set of *edges* $E \subseteq V \times V$. An *edge* $e = (u, v) \in E$ stands for a connection from the source u to the target v. We denote the number of vertices of a graph with $n = |V|$ and the number of edges with $m = |E|$.

The graphs we work with in this thesis are undirected, so that $(u, v) \in E$ if and only if $(v, u) \in E$. When there is an edge $(u, v) \in E$, then u and v are *neighbors*. Similarly $N(u) = \{v : (u, v) \in E\}$ denotes the *neighborhood* of u.

A *loop* is an edge $(u, u) \in E$ connecting a vertex to itself. The *degree* of a vertex $deg(v)$ is the number of incident edges, counting loops twice. The *distance* between two vertices is the number of edges in a shortest path connecting them. The *diameter* d of a graph is the greatest distance between any pair of vertices.

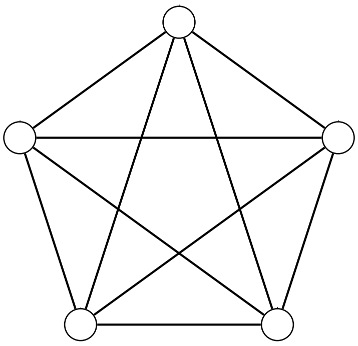


Figure 1.3.: The complete graph K_5

As an example the complete graph $K_5 = (V, E)$ with $|V| = 5$ and $\forall (u, v) \in V \times V : u \neq v \rightarrow (u, v) \in E$ is visualized in Figure 1.3.

Further definitions of terms used in this thesis can be found in the Glossary.

1.4. Outline

This thesis is structured as follows.

Chapter 2 discusses related work in the field of parallelizing and porting some graph algorithms and other algorithms to the Intel Xeon Phi architecture, including evaluations and comparisons to the scope of our work.

Chapter 3 details the general Intel Xeon Phi coprocessor architecture, its core pipeline design, the organization of caches and memory as well as the vector processing unit and how the Xeon Phi can be programmed.

Chapter 4 goes into more detail for the Xeon Phi programming models, looking into parallelization as well as vectorization. For the MIMD parallelization the programming models of OpenMP, OpenMP offloading, Cilk Plus and MPI are presented. For the SIMD vectorization we present two major approaches to achieving it: Manual vectorization by the developer using assembly language or compiler intrinsics, and auto-vectorization by the compiler, often assisted by the programmer.

Chapter 5 introduces the generative algorithm for massive complex networks. The parallel version of the algorithm is presented. General optimizations and Xeon Phi specific ones, as well as the challenges of porting the specific algorithm as well as the surrounding framework, NetworKit, are detailed. In the results the running time of the implementation is compared with that of the host system.

Chapter 6 introduces the multilevel maxent-stress graph drawing algorithm that uses graph clustering. The difficulties in porting the algorithm are explained. Differences to the graph generation algorithm and simple yet effective optimizations are discussed. In the final results we again discuss the resulting performance in comparison to the host system.

Chapter 7 concludes this thesis by summarizing the results and discussing the lessons learned. An outlook into future research concerning graph algorithms on the Xeon Phi is given.

CHAPTER 2

Related Work

Processing sparse graphs and in particular complex networks on accelerators is a non-trivial task due to the input's irregular structure, high synchronization costs, and frequent memory dereferences in sparse data structures [8]. Several promising works exist, however.

Saule and Çatalyürek have investigated the scalability of several graph algorithms on the Intel Xeon Phi [9]. For breadth-first search they have achieved significant speedups, while graph coloring and other irregular computations had lower speedups. In most examples the OpenMP implementation scaled better than those using Intel Threading Building Blocks (TBB) and Cilk Plus. We will take a look at OpenMP, TBB and Cilk Plus in Chapter 4. The authors note that the work required for porting algorithms to the Intel Xeon Phi is relatively low.

Sarıyüce et al. [10] have investigated vectorization for the Xeon Phi taking the problem of computing closeness centrality as an example. This centrality measure determines the importance of vertices based on aggregated distances. The authors compare manual vectorization using 512-bit Xeon Phi intrinsics with automated vectorization. Only a low overhead is found for the automated approach using fairly high level C++ template code with OpenMP pragmas and a modern optimizing compiler. Using this technique, they observe a speedup factor of up to 11 compared to a CPU implementation on real social networks.

Other performance evaluations of the Xeon Phi have shown that the Xeon Phi can compete with GPUs, for example on the problem of sparse matrix multiplication, which is of interest in many scientific applications, such as linear solvers and graph mining. The limiting factor turned out to be the memory latency, not the bandwidth of the Xeon Phi. The sparse kernel performance of the Xeon Phi (a prototype of the SE10P model) has been found to be superior to that of current general purpose processors and GPUs [11]. This result makes the Xeon Phi a particularly interesting accelerator for graph algorithms since many real-world graphs are sparse.

Moreover, a performance comparison of Xeon Phi and GPUs for the Swendsen-Wang algorithm for Monte Carlo simulations has found significant speedups for both accelerator platforms when compared to general purpose CPUs. [12]. The results place the Xeon Phi head to head with current GPU acceleration cards. The adaptation for the Xeon Phi was found to be straightforward.

Two techniques for ray tracing have been adapted to the Intel MIC architecture, successfully using wide-SIMD operations. Traditionally it has been challenging to efficiently map ray tracing algorithms to wide-SIMD hardware [13].

Note that the graphs we consider in the first algorithm are much larger than in the above works and do not fit into the Xeon Phi's memory. This requires us to use offloading and communication to the host system to send partial results.

The book "Intel Xeon Phi Coprocessor High-Performance Programming" investigates the potential of auto-vectorization, as well as manual vectorization of code using multiple examples. We introduce OpenMP, Intel TBB and Cilk Plus as technologies for task separation [14].

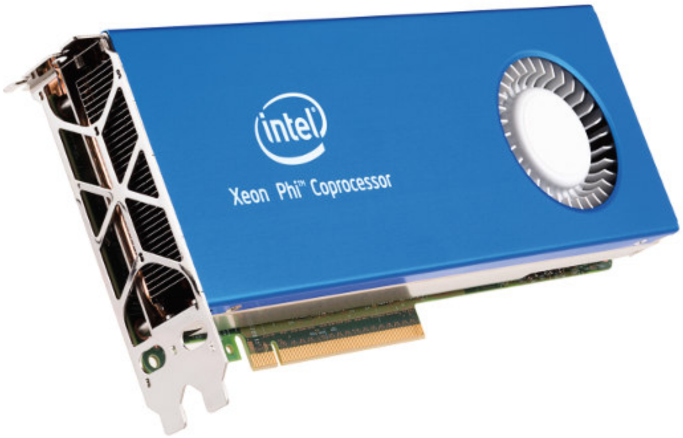
Other general works demonstrate the scalability of Xeon Phi cards to super-computing tasks while maintaining a favorable power-performance trade-off, making the Xeon Phi an interesting alternative to GPU accelerator cards in high performance computing and supercomputers especially [15].

A Best Practice Guide describing approaches to performance improvements is provided, which explains how to use memory alignment, SIMD optimizations as well as OpenMP optimizations [16].

Efficient parallelization using OpenMP is considered by Cramer et al. as an interesting alternative to more laborious rewrites in CUDA or OpenCL for GPUs [17].

CHAPTER 3

Intel Xeon Phi Coprocessor



The Intel Xeon Phi is an accelerator card connected over the PCIe bus and designed to handle highly parallel problems. In this master thesis we work with a 5100 series card (code name Knights Corner) from the first generation of this architecture, which is also known as Intel Many Integrated Core Architecture (Intel MIC). The specific card we use, the Xeon Phi 5110P, contains 60 in-order cores clocked at 1.053 GHz and is based on the original Pentium design from 1994, but augmented with 64-bit support, a 512-bit SIMD vector unit and four hardware threads per core. The internal GDDR5 memory has a size of 8 GB and is connected through 8 memory controllers (GDDR MC) with a theoretical bandwidth of 350 GB/s. All 60 cores, their 512 KiB L2 cache, the memory controllers and the PCIe interface are interconnected using a bidirectional ring bus of Core-Ring Interconnects (CRI) [18]. Cache accesses are kept coherent using a distributed Tag Directory (TD) that tracks the cache lines and their state in all L2 caches. This layout is visualized in Figure 3.1.

3.1. Core Pipeline

Being based on the original Pentium CPU, the Xeon Phi's cores' internal architecture is much simpler than that of a modern CPU. Contrary to the common out-of-order cores of modern CPUs the Xeon Phi executes instructions fully in-order,

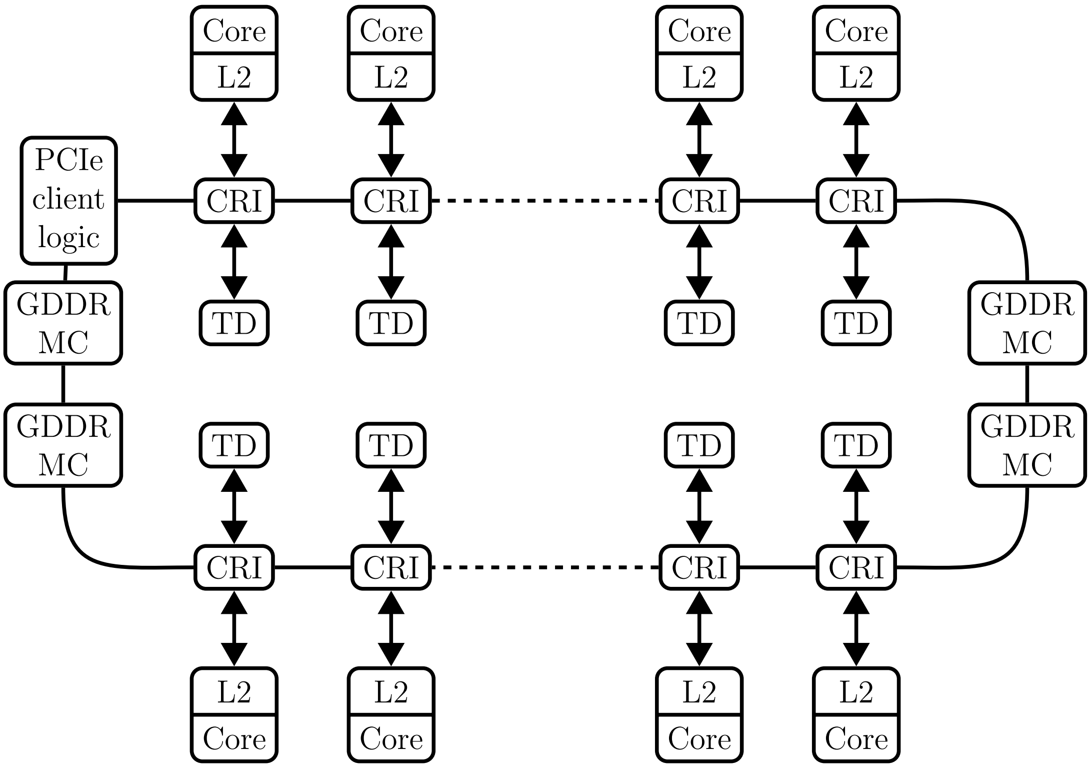


Figure 3.1.: Overview of the layout of a Xeon Phi accelerator card, detailing 8 of the 60 available cores and 4 of the 8 GDDR memory controllers.

which simplifies the core design. The other main differences are Simultaneous Multithreading (SMT) for four threads instead of the usual two threads in Intel's Xeon and i7 CPUs. Additionally the Xeon Phi features a 512-bit wide vector processing unit (VPU), which performs favorably compared to the Advanced Vector Extensions (AVX) of current CPUs. 512-bit wide SIMD (single instruction, multiple data) units are only expected for regular Intel Xeon CPUs by 2016, while the Xeon Phi has been available since 2012. Only the Initial Many Core Instruction set (IMCI) is supported on the Xeon Phi, with no compatibility to the long history of SIMD instruction sets for x86 CPUs, of which the major ones are:

- *MMX* from 1997, capable of processing eight 8-bit integers concurrently (64 bits wide)
- *SSE* (Streaming SIMD Extensions) from 1999, capable of processing four 32-bit floating point numbers concurrently (128 bits wide)
- *SSE2* (2001), *SSE3* (2003), *SSSE3* (2006), *SSE4* (2007), adding new instructions to SSE
- *AVX* (2011) and *AVX2* (2013), expanding floating point and integer commands to 256 bits

A clean cut of compatibility with the x86-64 instructions and extensions leads to simpler CPU cores, but breaks compatibility with existing implementations that have been explicitly optimized using these SIMD extensions.

The internal pipeline design of a single core is shown in Figure 3.2, with an actual photo of the Xeon Phi CPU in Figure 3.3.

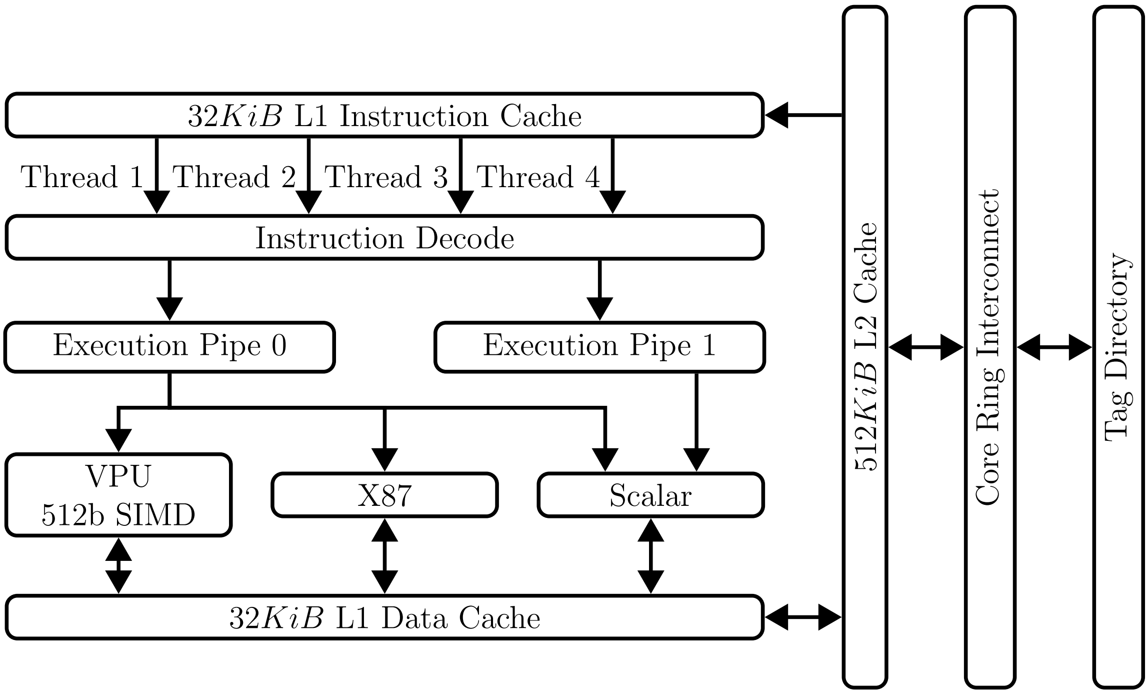


Figure 3.2.: Xeon Phi Core Thread and Instruction Pipeline. The right side shows the core's access to the Core Ring Interconnect. This is the source of the instruction cache as well as connected to the data cache. From the L1 instruction cache four threads are simultaneously prefetched. Two of those threads can run simultaneously on the execution pipelines. While the first pipeline can access the VPU (SIMD), X87 (floating point) as well as scalar execution units, the second pipeline can only access the scalar execution unit.

3.2. Caches and Memory

	L1	L2
Size	$32KiB + 32KiB$	$512KiB$
Associativity	8-way	8-way
Line size	$64B$	$64B$
Banks	8	8
Access time	1 cycle	11 cycles
Policy	pseudo LRU	pseudo LRU

Table 3.1.: Xeon Phi Key Cache Parameters

Each core features a 32 KiB first level data cache as well as another separate 32 KiB instruction cache. No hardware prefetching is available for the L1 data cache and instead the compiler or programmer can use software prefetching instructions.

The L2 cache has a size of 512 KiB. A simple form of a hardware streaming prefetcher performs caching on the L2 cache. The most interesting aspect of the L2 cache is its function as a shared last level cache among all the Xeon Phi cores. The L2 cache is part of the core-ring interconnect, which allows read-only access to the L2 caches of other cores.

Figure 3.3.: Die of the Xeon Phi coprocessor, showing 64 interconnected cores in the center (4 of which are disabled to increase yield)

When a core encounters a cache miss on its own L2 cache, it sends an address request to the tag directories of the other cores. The tag directories track the cache lines in all L2 caches. When a memory address is a encountered in a tag directory, that core's L2 cache sends the requested data. Otherwise the memory address is sent to the memory controller.

Caches are kept coherent among all cores using the MESI protocol. This commonly used cache and memory coherence protocol gets its name from marking each cache line as *modified, exclusive, shared* or *invalid*.

3.3. Vector Processing Unit

The Vector Processing Unit (VPU) of each core contains 32 vector registers of 512 bits width. Up to 32 single-precision or 16 double-precision floating-points operations can be performed per cycle. An example SIMD calculation for a simple vector addition is presented in Figure 3.4.

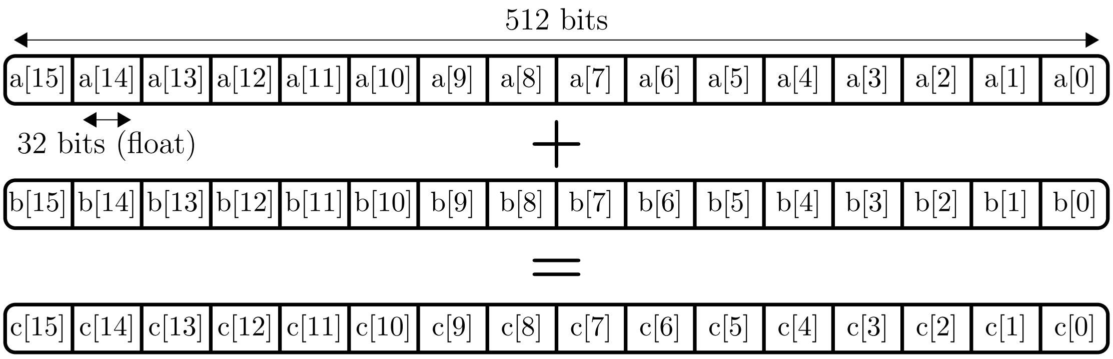

512 bits
a[15] a[14] a[13] a[12] a[11] a[10] a[9] a[8] a[7] a[6] a[5] a[4] a[3] a[2] a[1] a[0]
32 bits (float)
+
b[15] b[14] b[13] b[12] b[11] b[10] b[9] b[8] b[7] b[6] b[5] b[4] b[3] b[2] b[1] b[0]
=
c[15] c[14] c[13] c[12] c[11] c[10] c[9] c[8] c[7] c[6] c[5] c[4] c[3] c[2] c[1] c[0]

Figure 3.4.: Representation of vector addition `vaddps a, b, c`.
All of the 16 pairs of 32-bit values are summed in a single instruction.

3.4. Programming

For the programmer the Xeon Phi is presented as a regular computer running a stripped down Linux operating system. Direct access to the Xeon Phi is possible using SSH over a virtual network. The programming languages C, C++ and Fortran as well as those that compile through either of these languages, such as Cilk, Cilk++ and Nim [19], can be used with Intel's compiler. GCC is adding elementary Xeon Phi support with version 5.1, but is not expected to perform the advanced automatic vectorizations of Intel's compiler. For parallelization, frameworks such as MPI, OpenMP, Cilk Plus and others can be used. This standard tool-chain significantly reduces the overhead of adapting existing code to the Xeon Phi. The *offloading mode* allows regular OpenMP code to run on the host system while offloading calculations, fully or partially, to Xeon Phi coprocessors.

CHAPTER 4

Software Technologies for Vectorization and Parallelization

The Intel Xeon Phi has two main mechanisms for speeding up software, both of which introduce forms of parallelism: Parallelizing the code to use up to 240 threads on 60 cores, and vectorizing the code to use the Vector Processing Unit. In this chapter we explore the possible software technologies to make use of both of these mechanisms.

4.1. Parallelization

Parallelization is a form of MIMD (multiple instruction, multiple data) parallelism, that runs different instructions on different data on each of the processing units, in our case 60 cores with 240 threads on a single Xeon Phi coprocessor. There are multiple approaches available to parallelize code, which we will explain in this section.

Code Block 4.1 Non-parallelized and non-vectorized array addition

```
float a[MAX], b[MAX], c[MAX];
for (i = 0; i < MAX; i++)
  c[i] = a[i] + b[i];
```

As an example we shall parallelize the simple array addition code in Code Block 4.1.

4.1.1. OpenMP

OpenMP is an API for multi-platform shared memory multiprocessing programming. It is available for C, C++ and Fortran and already widely used in existing software, which enables easy porting to the Xeon Phi.

OpenMP consists of a set of compiler directives. When these directives are ignored or the compiler is not aware of them at all, the program simply runs sequentially. When the compiler interprets the directives, they are used to spawn parallel tasks in the defined manner. OpenMP programs start executing with a single thread until a

Code Block 4.2 Parallelized array addition with OpenMP

```
float a[MAX], b[MAX], c[MAX];
#pragma omp parallel for
for (i = 0; i < MAX; i++)
  c[i] = a[i] + b[i];
```

parallel construct is encountered. Only at such a parallel construct a thread pool is spawned to work through parts of the problem in parallel.

Our running example has been parallelized with OpenMP in Code Block 4.2.

4.1.2. OpenMP Offloading

With regular OpenMP pragmas the code can be run directly on the Xeon Phi in the same way as on a regular CPU. An alternative mode of execution is offered by OpenMP offloading, where the main code is executed on the host system. Only specific parts of the code are offloaded to the Xeon Phi.

This has the advantage that the code can run on a host system which can be equipped with significantly more main memory than the Xeon Phi's maximum of currently 16 GB. Subsets of the problem can be offloaded to the Xeon Phi while the host system can either wait or calculate at the same time, using regular OpenMP. Data that has to be handled on the Xeon Phi, needs to be transferred over the PCIe bus and the final results received over it as well.

Code Block 4.3 Parallelized array addition with OpenMP offloading

```
float a[MAX], b[MAX], c[MAX];
#pragma offload target(mic) in(a, b) out(c)
{
  #pragma omp parallel for
  for (i = 0; i < MAX; i++)
    c[i] = a[i] + b[i];
}
```

In the running example in Code Block 4.3 the arrays *a* and *b* can be filled on the host system. With the beginning of the *pragma offload* section they are transferred to the Xeon Phi. We use the *in* and *out* offloading parameters to transfer the exact data that we need. If no such values would be given, all variables occurring in the inner block would be transferred in both directions, first to the Xeon Phi before executing the block, then back to the host system after the execution finished.

When the inner block is executed on the Xeon Phi, the host system waits for the calculation to finish. Inside the inner block all of the Xeon Phi's cores can be used, except for one that is reserved for transferring data over the PCIe interface. Finally the array *c* is transferred back to the host system.

Instead of having the host system wait while the Xeon Phi performs calculations, it is also possible to use partial offloading. While the Xeon Phi is working on a part of the problem, the host system can work on another chunk of the problem itself. (See Code Block 4.4)

Code Block 4.4 Parallelized array addition with asynchronous OpenMP offloading

```
float a[MAX], b[MAX], c[MAX];
// We specify to use the first Xeon Phi so that we can assure
// that the same one is used in the following offload_wait.
#pragma offload target(mic:0) in(a, b) out(c) signal(c)
{
  #pragma omp parallel for
  for (i = 0; i < MAX; i++)
    c[i] = a[i] + b[i];
}
// Host can execute code here while the Xeon Phi is processing
// the previous block at the same time
#pragma offload_wait target(mic:0) wait(c)
// Previous line blocks until the Xeon Phi finishes and signals
```

4.1.3. Cilk Plus

Cilk Plus is a programming language for parallel computing by Intel, based on C and C++. The offloading offered by Cilk Plus is more automatic than with OpenMP, but in return there is less control of what data is transferred and what code offloaded.

Code Block 4.5 Parallel Cilk Plus array addition

```
float a[MAX], b[MAX], c[MAX];
cilk_for (i = 0; i < MAX; i++)
  c[i] = a[i] + b[i];
```

Code Block 4.5 shows the parallel for loop construct `cilk_for` in Cilk Plus, which splits up the work of the loop into chunks that are worked through in parallel.

While the original Cilk++ implementation was proprietary, Intel has decided to open up Cilk Plus by contributing an open source implementation to the GNU Compiler Collection (GCC) and Clang. This might lead to wider adoption.

4.1.4. Threading Building Blocks

Intel Threading Building Blocks (TBB) is a high level C++ template library. It offers another approach to writing parallel programs, including the ability to write tasks according to high-level parallel programming paradigms. Porting existing programs to use TBB and its approach of algorithmic skeletons is more difficult than annotating an existing program with OpenMP pragmas. Our simple calculation has been ported to use TBB's `parallel_for` construct in Code Block 4.6.

Code Block 4.6 Parallel TBB array addition

```
float a[MAX], b[MAX], c[MAX];
parallel_for(size_t(0), MAX, size_t(1), [=](size_t i) {
  c[i] = a[i] + b[i];
});
```

4.1.5. MPI

The Message Passing Interface (MPI) is a standard for message-passing in distributed memory environments. Implementations are available for a wide range of parallel and distributed computers. MPI can be used complementarily to shared memory programming models, such as OpenMP, with MPI splitting a task among distributed machines and OpenMP multi-threading the code on each of the machines.

MPI allows connecting multiple Xeon Phi systems as well as their host systems or other computers with each other. For such highly parallelized systems, MPI is the preferred choice. Transferring big amounts of data using MPI is currently slower than OpenMP offloading for the Xeon Phi. In this thesis we used a single Xeon Phi card, at times in concert with its host system, for which OpenMP was a better fit, so MPI will not be regarded further.

For our running example MPI is also not a good fit, as it would require copying chunks of the array to other systems or processes by message passing. This would be significantly more expensive than simply performing the addition in a shared memory system without any copies.

4.2. Vectorization

Contrary to the MIMD parallelism of the parallelization, vectorization achieves SIMD (single instruction, multiple data) parallelism. This means that a single thread executes a single instruction on a group of data values, up to 16 for the Xeon Phi. The hot spots, tight inner loops, have to use the Vector Processing Unit (VPU) if a significant speedup in single threaded performance is desirable.

4.2.1. Manual vectorization

Vectorization can be achieved most reliably by vectorizing the code in question manually. This means that either assembly instructions directly or vector intrinsics have to be used. Intel provides vector intrinsics for C, C++ and Fortran [20]. They are a way of accessing the Xeon Phi's capabilities in a very direct manner, as most intrinsics represent a one-to-one translation to a single assembly instruction.

This is more manual work than the other methods, and has more room for error, but also allows more control and therefore potentially higher performance. The code is intrinsically non-portable with manual vectorization, and a non-vectorized version has to be maintained as well if portability is a goal.

An example for manual vectorization is presented in Code Block 4.7. It is essential to properly align the arrays and assure that their size `MAX` is a multiple of 16. Otherwise special cases are not handles, which would further complicate this example.

Code Block 4.7 Manually vectorized array addition

```
// Include header containing all Intel Compiler intrinsics
#include <immintrin.h>
float a[MAX] __attribute__((aligned(64)));
float b[MAX] __attribute__((aligned(64)));
float c[MAX] __attribute__((aligned(64)));
// Allocate 512 bit VPU registers
__m512 _a, _b, _c;
for (i = 0; i < MAX; i += 16) {
  // Load 512 bits from memory location &a[i] into register _a
  _a = _mm512_load_ps(&a[i]);
  _b = _mm512_load_ps(&b[i]);
  // Add 16 32-bit floating-point numbers in _a and _b
  and store the results in _c
  _c = _mm512_add_ps(_a, _b);
  // Store the results from _c in the memory location &c[i]
  _mm512_store_ps(&c[i], _c);
}
```

4.2.2. Auto-Vectorization

In contrast to manual vectorization we can use the portable code, without vector intrinsics, and let the compiler automatically vectorize it. The chance of programming errors is lower with this method and the code stays fully portable. The main disadvantage is a dependency on the compiler's ability to vectorize the code well. To help with this it is possible to give the compiler hints and instructions about what to vectorize in what manner. The diagnostics of the compiler can help in figuring out how well the vectorization worked and gives hints for improvements. Relying on auto-vectorization introduces a dependency on the specific compiler version used, whereas any other compiler version may have different performance characteristics with respect to the vectorized code.

The Intel compiler understands a wide range of attributes and pragmas that the programmer can use to improve the vectorization. Using compiler switches a vectorization report can be created, which details the level of vectorization of the code as well as suggests changes to improve it.

Code Block 4.8 Auto-vectorized array addition

```
float a[MAX] __attribute__((aligned(64)));
float b[MAX] __attribute__((aligned(64)));
float c[MAX] __attribute__((aligned(64)));
#pragma simd
for (i = 0; i < MAX; i++)
  c[i] = a[i] + b[i];
```

For example when the data is not properly aligned or the compiler can not assume proper alignment in Code Block 4.8, the compiler informs about this. When the number of iterations does not translate directly to the 512-bit VPU, the compiler will inform about this and handle the remaining data without the VPU. A successful vector report for our example is presented in Figure 4.1.

```
LOOP BEGIN at auto.c(5,8)
   vectorization support: reference c has aligned access
     [ auto.c(9,5) ]
   vectorization support: reference a has aligned access
     [ auto.c(9,5) ]
   vectorization support: reference b has aligned access
     [ auto.c(9,5) ]
   vectorization support: unroll factor set to 2
   SIMD LOOP WAS VECTORIZED
   --- begin vector loop cost summary ---
   scalar loop cost: 6
   vector loop cost: 0.620
   estimated potential speedup: 19.200
   lightweight vector operations: 5
   --- end vector loop cost summary ---
LOOP END
```

Figure 4.1.: Vectorization report created by the Intel Compiler for the auto-vectorized Code Block 4.8

4.2.3. Cilk Plus

In addition to invoking tasks and threads, Cilk Plus also allows specifying how to vectorize code in a high level way. On the other hand the code ends up platform-specific, as a specific compiler for the Cilk Plus language is required.

Code Block 4.9 Vectorized Cilk Plus array addition

```
float a[MAX] __attribute__((aligned(64)));
float b[MAX] __attribute__((aligned(64)));
float c[MAX] __attribute__((aligned(64)));
c[i:MAX] = a[i:MAX] + b[i:MAX];
```

Our running example is given in Code Block 4.9. We will not use this method so it will not be investigated further.

4.3. Platform

All experiments in this thesis were ran on a host system containing two Intel Xeon E5-2680 (Sandy Bridge) processors, featuring 8 cores and 16 threads at a core frequency of 2.7 GHz each. The system contains 256 GB of DDR3 RAM as the shared main memory with a frequency of 1.333 GHz. The server is running openSUSE 13.1

with the Linux kernel 3.11.10-21. This system contains an Intel Xeon Phi 5110p with 60 cores and 240 threads at a frequency of 1.053 GHz as well as 8 GB of GDDR5 main memory. The Intel ICC/ICPC 15.0.1 was used as the C/C++ compiler for the host system as well as the Xeon Phi coprocessor.

CHAPTER 5

Generation of Massive Complex Networks

5.1. Algorithm

Network Properties

Complex networks, which are used to model real data such as the hyperlinks of the world wide web or relationships between people [2], have non-trivial topological features: They are mostly *scale-free*, which means that their degree distribution follows a power law. The number of nodes with degree k is proportional to $k^{-\gamma}$ for a fixed $\gamma > 0$. This implies that there are a few high-degree nodes, so called *hubs*, and many low-degree nodes. Random graphs on the other hand have an exponential degree distribution. An example of a power-law as well as an exponential degree distribution is shown in Figure 5.1.

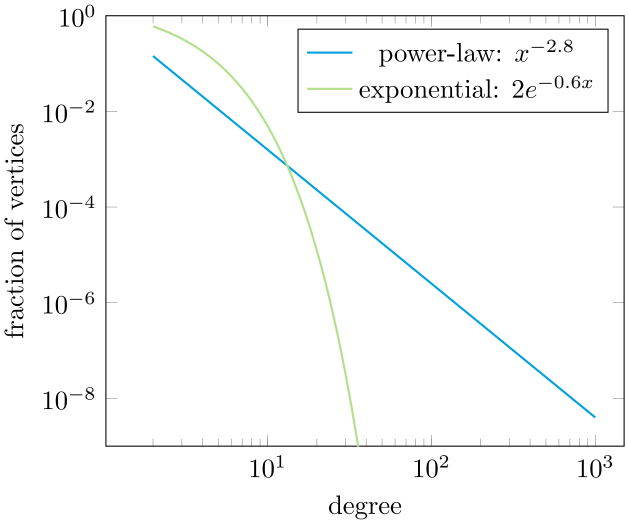


Figure 5.1.: Illustration of a power-law degree distribution in comparison to an exponential distribution. In a log-log plot the power-law is a straight line.

Another common class of networks are small-world networks, so called after the small-world phenomenon that postulates that all people are connected to each other

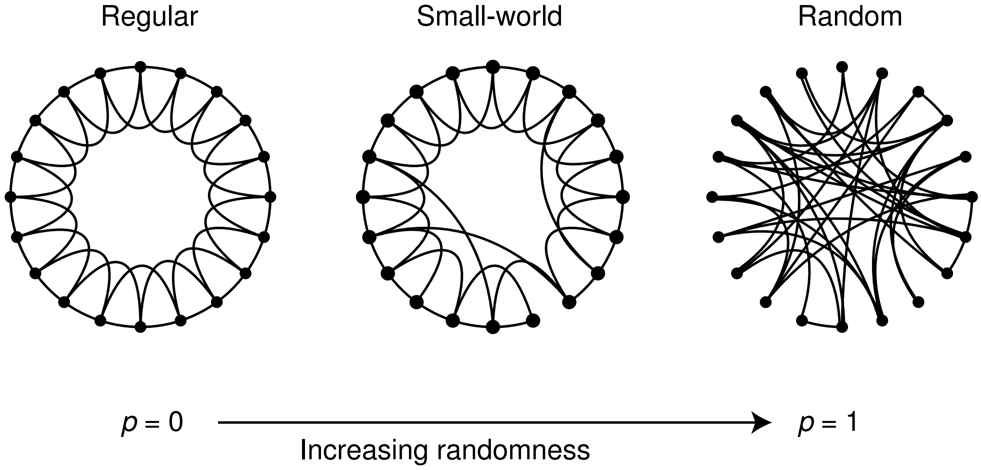


Figure 5.2.: Illustration of a small-world network compared to a regular and random network [22]

by only six degrees (or another small number) of separation. This corresponds to a graph with diameter 6 [21].

For small-world networks a high clustering coefficient is typical as well. The *clustering coefficient* quantifies how many triangles a graph contains compared to the number of triads, which are paths of length 2. This is a measure of the probability of two nodes with a common neighbor to be connected themselves. A simple small-world network is shown in Figure 5.2 [22].

We use the generative model of Krioukov et al., which is based on hyperbolic geometry [23]. The graphs generated with this model have an adjustable power-law degree distribution, a proven high clustering coefficient and community structure of a small-world network [24].

Hyperbolic Geometry

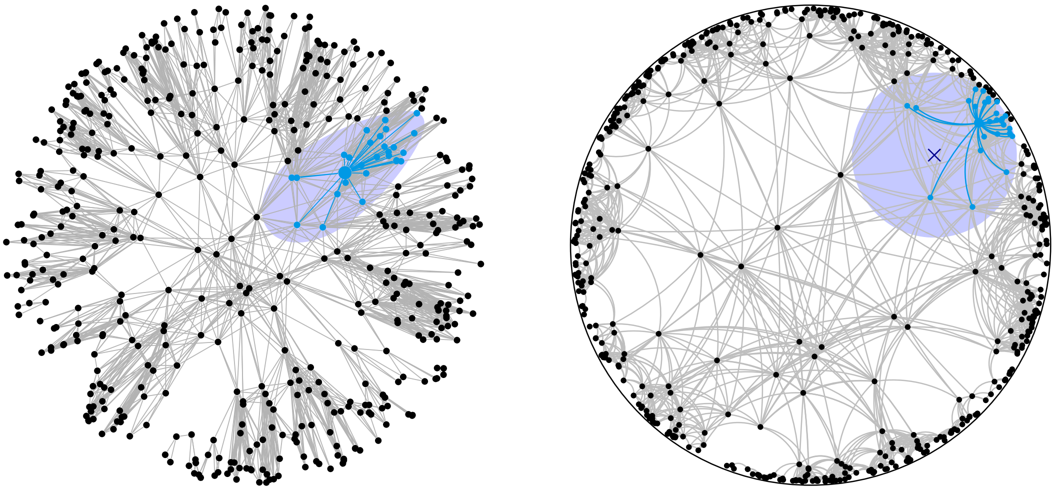
Hyperbolic Geometry is a non-Euclidean geometry. While all other Euclidean postulates are intact, the parallel postulate is negated:

Take any line R and and a point P that is not on R. The parallel postulate of Euclidean geometry states that there is exactly one line through P that is parallel to R, i.e. does not intersect it. In hyperbolic geometry there are multiple lines through P that do not intersect R.

Especially relevant to us is the property of *exponential expansion of space* of hyperbolic geometry. This means that the area of a circle grows exponentially with the distance from the center. A natural embedding of graphs with a tree-like structure is a consequence, as the number of vertices grows in a related manner, exponentially with the depth of a tree.

A hyperbolic graph containing 500 nodes is shown in Figure 5.3a. The neighborhood of a vertex u is defined as all vertices v whose hyperbolic distance $dist_{\mathcal{H}}(u, v)$ is below a fixed threshold. Thus the neighborhood of a vertex is a hyperbolic circle around it.

We use the Poincaré disk model, one of the commonly used representations of hyperbolic space within Euclidean geometry. In this disk model the points of the



(a) Native representation of a graph, visualized in hyperbolic geometry. The light blue circle denotes the neighborhood of the bold blue node.

(b) Poincaré disk model, which transforms the neighborhood into a Euclidean circle with moved center.

Figure 5.3.: Representation of a graph in hyperbolic geometry [25].

hyperbolic geometry are represented inside the unit disk. This model preserves angles and maps hyperbolic circles to Euclidean circles. The hyperbolic distance $dist_{\mathcal{H}}(u, v)$ can be calculated in the Poincaré model with [26]

$$dist_{\mathcal{H}}(u, v) = \text{acosh}\left(1 + 2\frac{||p - q||^2}{(1 - r_p^2)(1 - r_q^2)}\right), \tag{5.1}$$

where $p = (\phi_p, r_p)$ and $q = (\phi_q, r_q)$.

Figure 5.3b shows the same graph as Figure 5.3a, but translated into the Poincaré disk model. The neighborhood is a also a Euclidean circle in this model, simplifying the calculation whether a vertex is a neighbor of another vertex.

Generative Model

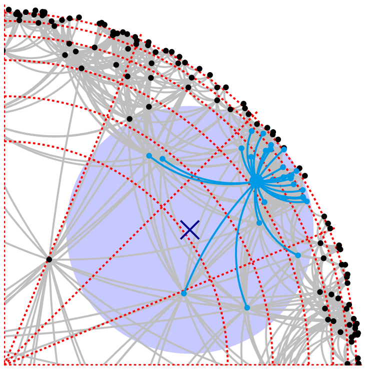


Figure 5.4.: Polar quadtree with the cells denoted in red [25].

Based on the model of Krioukov et al. with a high clustering coefficient and a power-law degree with an adjustable exponent, a faster generation algorithm was built by von Looz et al. [25]. It uses a polar quadtree on the Poincaré disk as its most important data structure. Without the quadtree, the determination of the neighborhood of every vertex requires a distance calculation to every other vertex. The quadtree instead limits this to the vertices contained in the leaf cells covering the neighborhood circle. A section of the polar quadtree is shown in Figure 5.4.

Algorithm 1: Sequential generation algorithm [25]

```
   Input: number of vertices $n$, dispersion $\alpha$, radius $R$
   // The dispersion parameter $\alpha$ determines the probability density of
      the placement of vertices
   Output: $G = (V, E)$
   // Returned graph with $|V| = n$ and expected $|E| = \frac{8}{\pi} n e^{-R/2} \frac{n}{2}$
 1 $V$ = set of $n$ vertices;
 2 $T$ = empty polar quadtree;
   // Create vertices on the unit disk and insert them into polar
      quadtree
 3 for vertex $v \in V$ do
 4     draw $\phi[v]$ from $\mathcal{U}[0, 2\pi)$;
 5     draw $r_{\mathcal{H}}[v]$ with density $f(r) \propto \alpha \sinh(\alpha r)$;
 6     $r_{\mathcal{E}}[v]$ = hyperbolicToEuclidean($r_{\mathcal{H}}[v]$);
 7     insert $v$ into $T$ at $(\phi[v], r_{\mathcal{E}}[v])$;
 8 end
   // Determine neighborhood of vertices, find the vertices inside this
      circle in the quadtree and add them as edges
 9 for vertex $v \in V$ do
10     $C_{\mathcal{H}}$ = circle around $(\phi[v], r_{\mathcal{H}}[v])$ with radius $R$;
11     $C_{\mathcal{E}}$ = transformCircleToEuclidean($C_{\mathcal{H}}$);
12     for vertex $w \in T$.getVerticesInCircle($C_{\mathcal{E}}$) do
13         add $(v, w)$ to $E$;
14     end
15 end
16 Return $G$
```

The sequential generation algorithm is presented in Algorithm 1. First the positions of the vertices are generated randomly in lines 4 (angle) and 5 (distance from center). In line 6 the vertices are mapped to the Poincaré disk. The vertices are stored in the polar quadtree data structure in line 7. Inside the *for* loop in line 9 the neighborhood of each vertex is calculated as a circle and the respective nodes of the quadtree are accessed in a range query to find the vertices in it.

The expected running time of this algorithm is $O((n^{3/2} + m) \log n)$, making it subquadratic thanks to the polar quadtree and a fast range query [25].

My contribution is the adaptation of the algorithm to the Xeon Phi, including general optimizations that improve scalability.

5.2. Implementation

An existing implementation for the hyperbolic graph generation algorithm was used. It is part of NetworKit, a high-performance network analysis toolkit written in C++

using C++ style code, which also exposes the algorithms to Python [27]. This means that contrary to most small algorithms studied in other Xeon Phi adaptations, the code is not written in a low level C programming style but instead uses complicated high level C++ data structures. As part of this thesis NetworKit was adapted to work with the Intel C++ Compiler as well as run on the Xeon Phi.

The Intel C++ compiler (ICPC) 15.0 is mostly compatible with the C++11 standard and the GNU C++ Compiler (g++), but a few differences and bugs still required changes to the source code of NetworKit as a whole:

Calculating the value for π at compile time with a `constexpr` was not possible, so it had to be worked around, as is shown in Code Block 5.1.

Code Block 5.1 Special case `constexpr` without any calculations for the Intel compiler

```
#ifdef __INTEL_COMPILER
constexpr double PI = 3.1415926535897932384626433832795028;
#else
constexpr double PI = 2.0*std::acos(0);
#endif
```

Parts of the code were implemented in high performance assembly, which had to be disabled for the Xeon Phi as it was detected as an x86-64 system so far (see Code Block 5.2).

Code Block 5.2 Disabling assembly instructions for the Xeon Phi architecture

```
#if defined __MIC__
#define TTMATH_NOASM
#endif
```

The Intel compiler is stricter about implicit type conversions, which required minor changes that can be considered good style anyway. One of these changes is depicted in Code Block 5.3.

Code Block 5.3 Explicit type conversion to keep the ICPC happy

```
// previously: std::set<unsigned int> seeds = {seed};
std::set<unsigned int> seeds = {(unsigned int) seed};
```

The C++ monotonic clock `std::chrono::steady_clock` still uses its old name in ICPC when compiling for the Xeon Phi, requiring another `ifdef` (see Code Block 5.4).

The `nullptr` keyword is not supported yet either, so `(void*)0` has to be used instead.

Code Block 5.4 `steady_clock` is still called `monotonic_clock` when compiling for the Xeon Phi

```
#ifdef __MIC__
#define my_steady_clock std::chrono::monotonic_clock
#else
#define my_steady_clock std::chrono::steady_clock
#endif
```

Neither available is a vector's `emplace` method, which can be emulated with `insert` instead, as it is done in Code Block 5.5.

Code Block 5.5 Emulating `Vector::emplace`

```
#if defined(__MIC__)||(defined(__GNUC__)&&!__GNUC_PREREQ(4,8))
result.insert(std::make_pair(i, temp));
#else
result.emplace(i, temp);
#endif
```

The main problem turned out to be a bug in function traits that are used on lambdas (anonymous functions). This is a common concept at the very core functionality of NetworKit. A single such replacement, of which many related ones were necessary, is seen in Code Block 5.6.

Additionally support for default template parameters is missing, which required manually adding the default values to the call sites.

It became clear that many optimizations do not just have an effect on the Xeon Phi, but also speed up execution on a regular CPU. We will look at general optimizations and Xeon Phi specific ones in the following section about our results.

Code Block 5.6 Working around differing templating implementations

```
template<class F, bool InEdges = false,
  typename std::enable_if<std::is_same<
    edgeweight,
    typename Aux::FunctionTraits<F>::template arg<2>::type
  >::value>::type* = nullptr>
auto edgeLambda(F&f, node u, node v, edgeweight ew, edgeid id)
    const -> decltype(f(u, v, ew)) {
  return f(u, v, ew);
}
```

(a) Old implementation, failing to compile with ICPC

```
template<class F, bool InEdges = false,
  typename std::enable_if<
    (Aux::FunctionTraits<F>::arity >= 2) &&
    std::is_same<
      edgeweight,
      typename Aux::FunctionTraits<F>::template arg<2>::type
    >::value
  >::type* = (void*)0>
auto edgeLambda(F&f, node u, node v, edgeweight ew, edgeid id)
    const -> decltype(f(u, v, ew)) {
  return f(u, v, ew);
}
```

(b) New implementation with explicit function arity check, working in g++ and ICPC

5.3. Results

5.3.1. General optimizations

As the queries on the polar quadtree are all independent of one another, they can be parallelized trivially. The level of parallelism gained by this is sufficient, as all interesting graphs consist of a higher number of vertices than we have computing cores available.

The main problem with speeding up the algorithm were the frequent memory reallocations, internally causing locking *malloc* calls. While the algorithm scaled to 4 threads on the Xeon Phi, it couldn't get any benefit from adding any more threads. When using the Intel TBB's *malloc* implementation, the algorithm scaled up to 120 threads. This effect was much bigger on the Xeon Phi than on the Xeon host system. Therefore it is especially important to minimize memory allocations when optimizing code for the Xeon Phi.

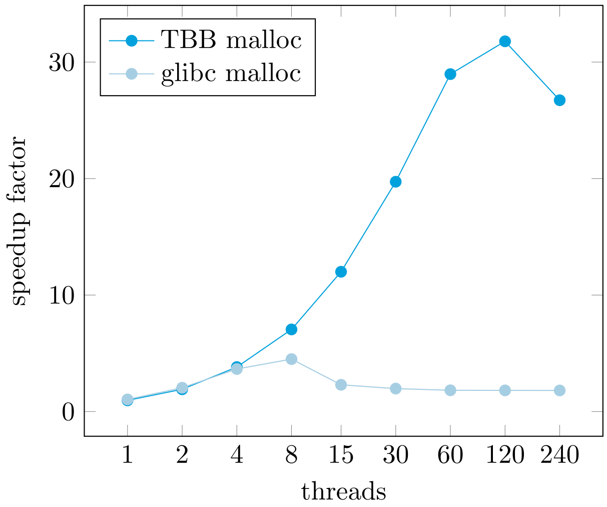


Figure 5.5.: Speedup of the initial implementation for 200000 vertices on the Xeon Phi, with glibc's *malloc* compared to TBB's lock-free *malloc*. Later further optimizations were made that sped up the execution even with malloc.

In initial development, using the lock-free malloc implementation of Intel Threading Building Blocks had a massive impact (see Figure 5.5), in current versions the speedup is a mere factor 2 (see Figure 5.6). We conclude that a lock-free malloc implementation is an important optimization for parallel code that often allocates memory.

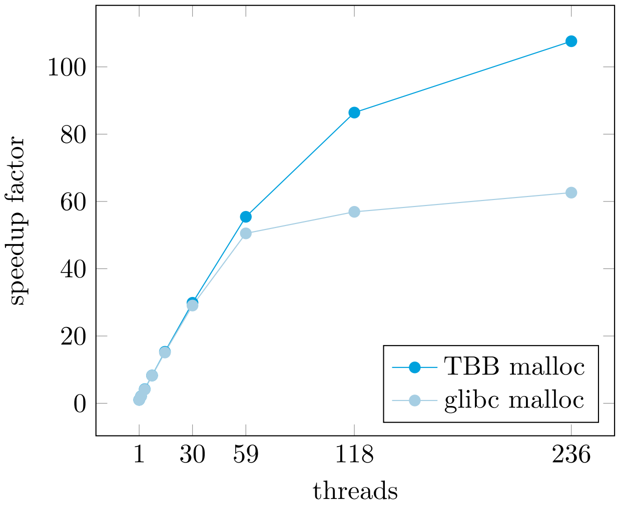


Figure 5.6.: Speedup of full Phi offloading in the final, optimized code, for $2.1 \cdot 10^6$ vertices and $6.8 \cdot 10^9$ edges, with glibc's *malloc* compared to TBB's lock-free *malloc*

The reduction of memory allocations had another big effect, speeding up the code by a factor of two. This was achieved by pre-allocating a data structure of the expected size instead of growing the data structure when adding new vertices to the neighborhood. Another optimization was to reuse data structures instead of allocating new ones in each loop iteration.

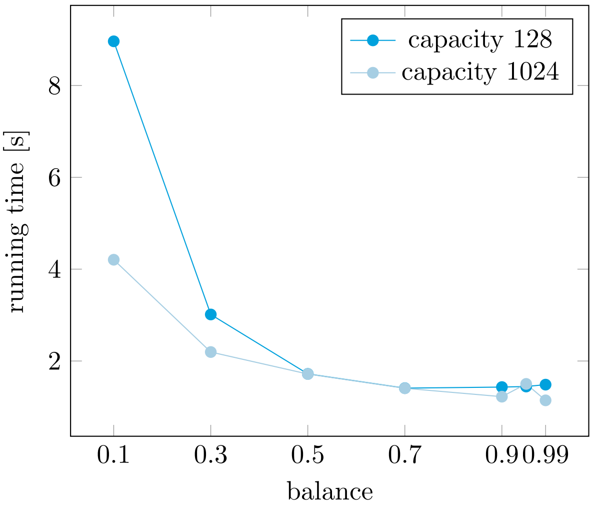


Figure 5.7.: Running time depending on the algorithm's *balance* and *capacity* parameters for partial offloading of a graph with $2.6 * 10^5$ vertices and $3.0 * 10^8$ edges.

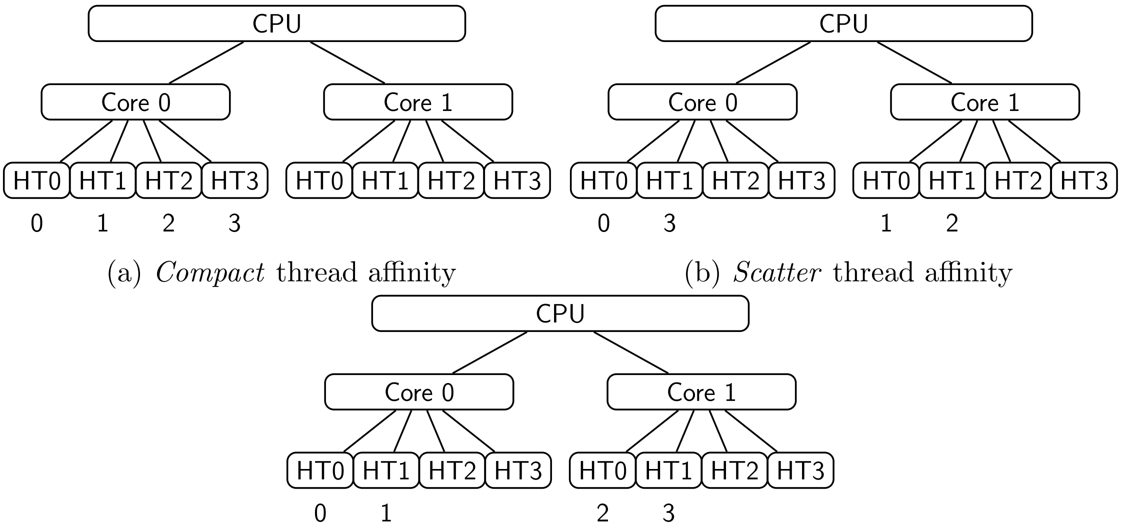
The quadtree has two major variables that can be used to optimize performance for a specific system, the tree *balance* and the *capacity* of leaf cells. The tree *balance* determines how big the children cells get when a leaf is split. A balance of 0.1 implies assigning a greater amount of space to inner children, 0.5 stands for a perfect balance, and 0.9 assigns a greater amount of space to outer children.

In Figure 5.7 the running time depending on the tree balance for capacity 128 and 1024 is shown. We observe that a deliberate imbalance, assigning a greater amount of space to the outer children in each split, is yields better performance. The reason for this is that we keep the innermost leaf cells small, as they will be included in many neighborhood circles. We thereby reduce the amount of unnecessary distance calculations to determine the neighborhood of a vertex. This imbalance optimization was discovered by Moritz von Looz and was confirmed on the Xeon Phi.

Using a `schedule(type, chunk)` OpenMP clause the type of work sharing in an OpenMP loop can be adjusted:

- The `chunk` parameter determines the number of contiguous iterations a thread is assigned at a time.
- The *static* scheduling `type` pre-allocates the iterations for each thread at compile time. This turns out to be too inflexible for our algorithm, as the amount of work per iteration very likely varies due to the widely varying degree of our scale-free graphs.
- The *dynamic* scheduling `type` allocates a new chunk of iterations to a thread as soon as the thread is finished with its current chunk.
- The *guided* scheduling `type` works in the same way as dynamic scheduling, but exponentially decreases the chunk size after each allocation until it reaches the parameter `chunk`. We observed similar performance using dynamic and guided scheduling.

It is possible to control how OpenMP threads are placed on the physical cores using the `KMP_AFFINITY` environment variable (see Figure 5.8):


CPU
Core 0
Core 1
HT0 HT1 HT2 HT3 HT0 HT1 HT2 HT3
0 1 2 3
CPU
Core 0
Core 1
HT0 HT1 HT2 HT3 HT0 HT1 HT2 HT3
0 3 1 2
CPU
Core 0
Core 1
HT0 HT1 HT2 HT3 HT0 HT1 HT2 HT3
0 1 2 3

(a) *Compact* thread affinity

(b) *Scatter* thread affinity

(c) *Balanced* thread affinity

Figure 5.8.: Illustration of `KMP_AFFINITY` thread affinity with 4 threads being placed on an imaginary 2-core system with 4 threads per core.

- *Compact* thread affinity places the threads as close together as possible. For the Xeon Phi this means that when run with 120 threads, they will only use 30 of the cores. This significantly reduces the performance by a factor of 1.4, but not to a factor of 2, which would be expected if the computing power of the cores was the limiting factor. Instead it shows that memory accesses are also a limiting factor.
- *Scatter* thread affinity places the threads as far away from each other as possible.
- *Balanced* thread affinity is only available on the Xeon Phi architecture. It spreads threads to physical cores in the same way as *scatter* affinity, but assures that close thread numbers stay on the same core. This greatly improves cache sharing between related threads that are more likely to have close thread numbers. As this is not the case for our implementation, no significant performance benefit can be measured.

5.3.2. Adaptation to Xeon Phi

Compared to the previous optimizations, the adaptation to the Xeon Phi requires broader changes in the generation algorithm. The individual cores of the Xeon Phi are slower than current general purpose processors and we need to fully exploit the parallelism to use them effectively. Additionally, the 8GB of available memory are too small to store a graph of interesting size. We address this with a semi-external memory algorithm and store only the quadtree containing the vertex coordinates on the card, while edges are streamed out to the host system after they are generated. This approach is compatible with multiple Xeon Phi accelerator cards if they are available.

The Xeon Phi card can be used in two ways: With *full offloading*, we generate all edges on the Xeon Phi and only insert them into the graph on the host system.

This allows the host system to work on other tasks, for example using the generated graph on the fly. With *partial offloading*, as described in Chapter 4, Xeon Phi and the host system both generate edges for parts of the graph in parallel, which is what we use for our semi-external memory algorithm.

The parallel quadtree and point generation cannot be used on the Xeon Phi directly if partial offloading or offloading to multiple cards is used. Because of randomness in the generation and non-deterministic running order of threads, the cards would end up with different vertex coordinates internally, causing the generated graph to be inconsistent. Instead we create the quadtree on the host system and transfer it to the Xeon Phi (or all of them). The code to generate the graph's edges for a subset of vertices is offloaded to the accelerator card. To keep the data transfer overhead low, we use a double buffer on both sides: Once to transfer generated edges from the Xeon Phi to the host system and subsequently to insert the received edges into the graph structure. The general structure of this approach can be seen in Algorithm 2.

A difficulty in the transfer is the fact that the quadtree is a complex C++ object containing other objects and C++ data structures, while OpenMP offloading only supports the transfer of simple linear data structures. This has been solved by using *cereal*, a C++11 header-only serialization library [28]. All relevant classes have been annotated with serialization instructions. A binary stream is created from the quadtree data structure, transferred to the Xeon Phi, and finally deserialized on it. While this takes a significant amount of time, it is still faster than the quadtree generation on the Xeon Phi itself, mainly because of the difficulty of efficiently allocating memory on this platform.

Cereal 1.0.0 did not compile using Intel's C++ compiler, but could be fixed by using workarounds implemented for older GCC versions and Microsoft's C++ compiler. These issues have been fixed in a more recent version of the Cereal library by now.

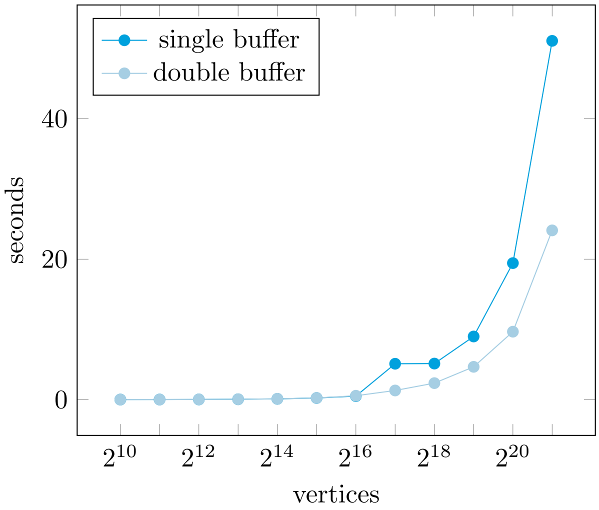


Figure 5.9.: Running times for full offloading with and without double buffering for dense graphs. Starting with 2^{17} vertices at least two offloads are necessary, allowing the double buffering to become effective.

The double buffering only affects running times when at least two offloads are necessary, which starts at $\approx 10^8$ edges. This speedup factor of two can be seen in

Algorithm 2: Double buffering solution to prevent latencies on the Xeon Phi and host alike

```
Input: number of vertices $n$, number of edges $m$
Output: resulting graph graph
points = generatePoints(n, m);
quadtree = createQuadtree(points);
graph = newGraph();
phiIteration = 0;
#pragma transfer quadtree to Phi async
#pragma omp parallel for num_threads(2)
for $i = 0; i < n; i += part$ do
    if omp_get_thread_num() == 0 then
        graph += generatePart(i, i+part);
    end
    else
        if phiIteration == 0 then
            #pragma wait for quadtree to be on Phi
        end
        if isEven(phiIteration) then
            #pragma offload
                buffer1 = generatePart(i, i+part);
            if $phiIteration > 0$ then
                wait until buffer1 has been handled;
            end
            #pragma transfer buffer1 from Phi async
            if $phiIteration > 0$ then
                #pragma wait until buffer2 is transferred
                async graph += buffer2;
            end
        end
        else
            #pragma offload
                buffer2 = generatePart(i, i+part);
            if $phiIteration > 1$ then
                wait until buffer2 has been handled;
            end
            #pragma transfer buffer2 from Phi async
            #pragma wait until buffer1 is transferred
            async graph += buffer1;
        end
        phiIteration += 1;
    end
end
if not isEven(phiIteration) then
    #pragma wait until buffer1 is transferred
    graph += buffer1;
end
else if $phiIteration > 0$ then
    #pragma wait until buffer2 is transferred
    graph += buffer2;
end
Return graph;
```

Figure 5.9, where dense graphs are generated and 10^8 edges correspond to 2^{17} vertices.

To take advantage of the 512-bit wide SIMD unit, we need to restructure the range query loop. Algorithm 3 shows the baseline loop used on other platforms. It cannot be vectorized in its current state because appending elements to Result introduces a data dependency.

Algorithm 3: Baseline Range Query within Leaf

```
Input: query circle radius $r$, query circle center $c$, list of vertex positions $pos$, list
       of vertex indices $id$
Output: list of vertices within query circle
Result = empty list;
cap = pos.size();
for $i = 0; i < cap; i{+}{+}$ do
    $p$ = pos[i];
    if $(p_x - c_x)^2 + (p_y - c_y)^2 < r^2$ then
        Result.append(id[$i$]);
    end
end
Return Result;
```

Algorithm 4 in turn adds a level of indirection between the distance calculation and neighborhood list generation. The distance calculations in lines 7 to 11 can then be vectorized. The effectiveness of this change depends on the average degree: In thin graphs the vast majority of considered neighbors are rejected and the second, unvectorized loop (lines 13 to 16) has only few non-empty iterations. In experimental results, the vectorized version of Algorithm 4 yields a speedup of about 1.5 over Algorithm 3. This is less than the theoretical optimum of 8, since only parts of the algorithm can be vectorized and we use 64-bit integers in all relevant data structures, which limits the effect of vectorization compared to 32-bit integers. However, communication and memory accesses are bigger contributors to the running time and the effect of vectorization vanishes when considering the complete algorithm. We implemented bit masks and De Bruijn sequences [29], but observed them to be slower because of the additional computations and more complicated memory access patterns.

Using static instead of dynamic or guided scheduling has the effect of slightly worsening the load balance with partial offloading, compared to the non-offloading implementation on the host system. Since the graph generation task is split into many smaller tasks for offloading to the Xeon Phi, random fluctuations in point density and thus thread workload have a greater effect than on the host system where they balance out over the span of a large task. Since we have to wait for all threads to be finished after every offloaded sub-computation, random fluctuations persist and individual threads thus can run up to 1.35 times longer than others. Thus, using guided or dynamic scheduling yields a speedup of 1.16 compared to static scheduling.

With all optimizations combined, a parallel speedup of over 100 is achieved compared to the non-parallel solution on the Xeon Phi, see again Figure 5.5. This compares

Algorithm 4: Vectorizable Range Query within Leaf

```
Input: query circle radius $r$, query circle center $c$, list of vertex positions $pos$, list
       of vertex indices $id$
Output: list of vertices within query circle
Result = empty list;
cap = pos.size();
Bits = boolean array of size cap, initialized to False;
#pragma ivdep
#pragma vector always
#pragma simd
for $i$ = 0; $i$ <cap; $i$++ do
    /* This loop is vectorized                                   */
    $p$ = pos[i];
    if $(p_x - c_x)^2 + (p_y - c_y)^2 < r^2$ then
        Bits[$i$] = True;
    end
end
for $i$ = 0; $i$ < cap; $i$++ do
    /* Not vectorized                                            */
    if Bits[i] then
        Result.append(id[$i$]);
    end
end
Return Result;
```

well to other works for scaling graph algorithms on the Xeon Phi [9], especially for complex networks.

The Xeon Phi can be used to offload the graph generation entirely, freeing the host system to use the generated graph in parallel. Because of the large size of our graphs, they do not fit into the card's memory and intermediate results need to be sent to the host system. Due to this communication bottleneck, we see lower speedups than in existing literature, where communication is usually not considered as part of the problem.

Our implementation enables the use of partial offloading, where the Xeon Phi and the host system generate the graph simultaneously. However, this has not resulted in significant speedups over a generation done completely on the host system, as is shown in Figure 5.10. It was however observed that 4 to 8 threads are necessary on the host system to efficiently handle the data transfers from the Xeon Phi, while still leaving the host system with enough capacity to solve significant parts of the problem by itself in parallel, as is depicted in Figure 5.11.

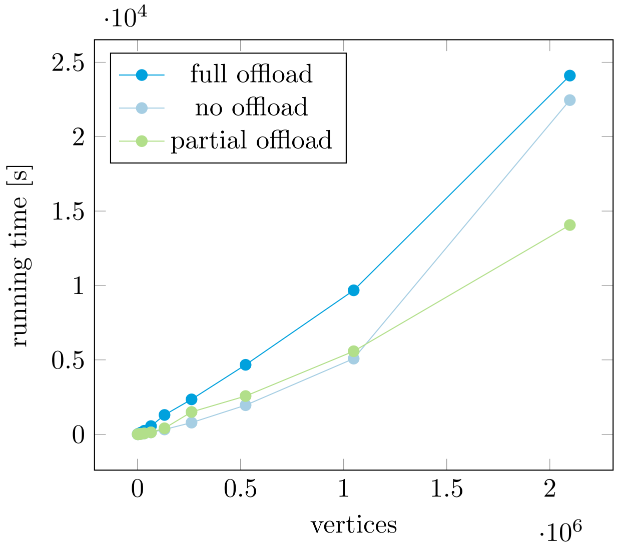


Figure 5.10.: Performance comparison for *no offloading* (run the entire code on the host system), *full offloading* (host system sends all parts of the problem to the Xeon Phi) and *partial offloading* (host system and Xeon Phi both solve parts of the problem) of the generation of a graph with variable number of vertices.

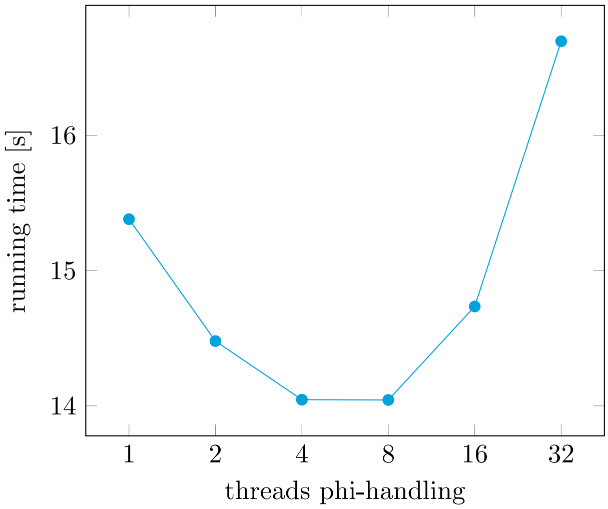


Figure 5.11.: Performance for partial offloading of the generation of a graph with $2.1 \cdot 10^6$ vertices and $6.8 \cdot 10^9$ edges, depending on the number of threads that are set aside on the host system for communicating with the Xeon Phi

CHAPTER 6

Graph Drawing using Graph Clustering

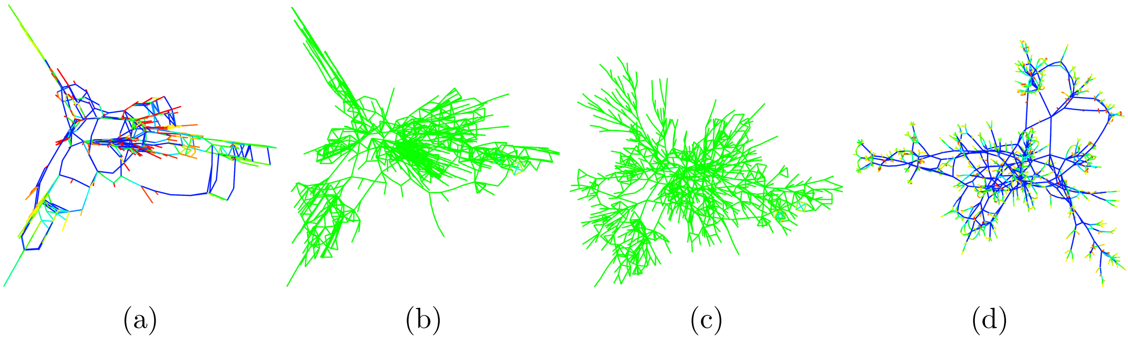

(a) (b) (c) (d)

Figure 6.1.: Drawing of the 1138_bus graph by (a) PivotMDS [30], (b) PivotMDS(1), (c) Maxent and (d) sfdp [31]. A red-to-green-to-blue scale is used to encode the edge lengths from short to long, where edges shorter than half the median edge length are red and those longer than 1.5 times the median are blue.

6.1. Algorithm

In this chapter we work with a preliminary implementation of the multilevel maxent-stress graph drawing algorithm developed by Meyerhenke et al. [32].

Full Stress Model

The graphs we consider have target lengths assigned to their edges. In such cases the full stress model is promising. It models physical springs connecting all pairs of vertices of the graph. The physical forces of the springs then push and pull the vertices relative to each other and thereby create a layout of the graph, minimizing the amount of stress energy in the system. This full stress model requires the calculation of ideal distances between all pairs of vertices. Johnson's algorithm to calculate the shortest paths between all pairs of vertices in the graph requires $O(|V|^2 \log |V| + |V||E|)$ time and $O(|V|^2)$ memory. This is too computationally expensive for large graphs.

Maxent-Stress Model

An optimization to this model is proposed by Gansner et al. [31]. Instead of the full stress model a sparse stress model is used, achieving only the pre-determined target edge lengths. The remaining degree of freedom is then resolved by maximizing the entropy of the layout, which gives the model its name, as it combines *maximizing entropy* with a *stress model*. The maxent-stress model was shown to perform well in the measure of full stress that remains in the graph after drawing it. Formally the maxent-stress $M(x)$ can be defined as

$$M(x) = \sum_{\{u,v\} \in E} w_{uv} \left(||x_u - x_v|| - d_{uv} \right)^2 - \alpha \sum_{\{u,v\} \notin E} \ln ||x_u - x_v||, \tag{6.1}$$

where d_{uv} is the target distance between nodes u and v, $w_{uv} = \frac{1}{d_{uv}^2}$ is a weight factor and the scaling factor α modulates the strength of the entropy term. This maxent-stress $M(x)$ is minimized by solving a series of linear systems.

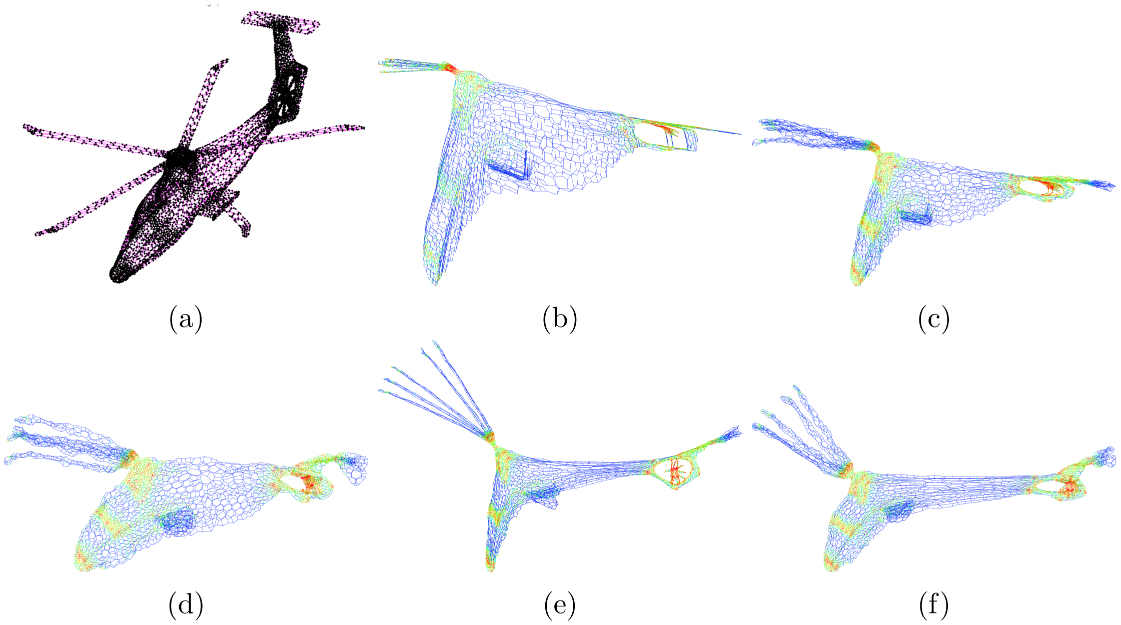


Figure 6.2.: Original commanche graph (a) and drawings by (b) PivotMDS, (c) PivotMDS(1), (d) PivotMDS(2), (e) Maxent and (f) Maxent(2) [31].

See Figures 6.1 and 6.2 for two examples of graphs drawn using the maxent-stress model, in comparison to other graph drawing algorithms.

Multilevel Maxent-Stress Algorithm

The algorithm presented by Meyerhenke et al. [32] also aims to minimize the maxent-stress, but does so by clustering the graph in multiple levels of hierarchy. Clusters are contracted into new supervertices in each round, setting the weight of the new vertex to the sum of the original vertex weights. At the coarsest hierarchy level an initial layout is done, which is then improved on each finer level by iterating a scheme to solve the maxent-stress model:

$$x_u \leftarrow \frac{1}{\rho_u} \sum_{\{u,v\} \in E} w_{uv} \left(x_v + d_{uv} \frac{x_u - x_v}{||x_u - x_v||} \right) + \frac{\alpha}{\rho_u} \sum_{\{u,v\} \notin E} \frac{x_u - x_v}{||x_u - x_v||^2}, \tag{6.2}$$

where $\rho_u = \sum_{\{u,v\} \in E} w_{uv}$.

This approach additionally exploits the hierarchy, as densely connected vertices end up in the same cluster and get drawn closer to each other [32].

While solving the linear system can not be parallelized, the multilevel maxent-stress algorithm uses shared-memory parallelism in the form of OpenMP. New coordinates of the vertices in the same iteration are computed independently in multiple threads using equation 6.2. The distance approximations in the entropy term are parallelized as well.

Two initial clusterings of the multilevel maxent-stress algorithm can be seen color-coded into the resulting graphs in Figure 6.3.

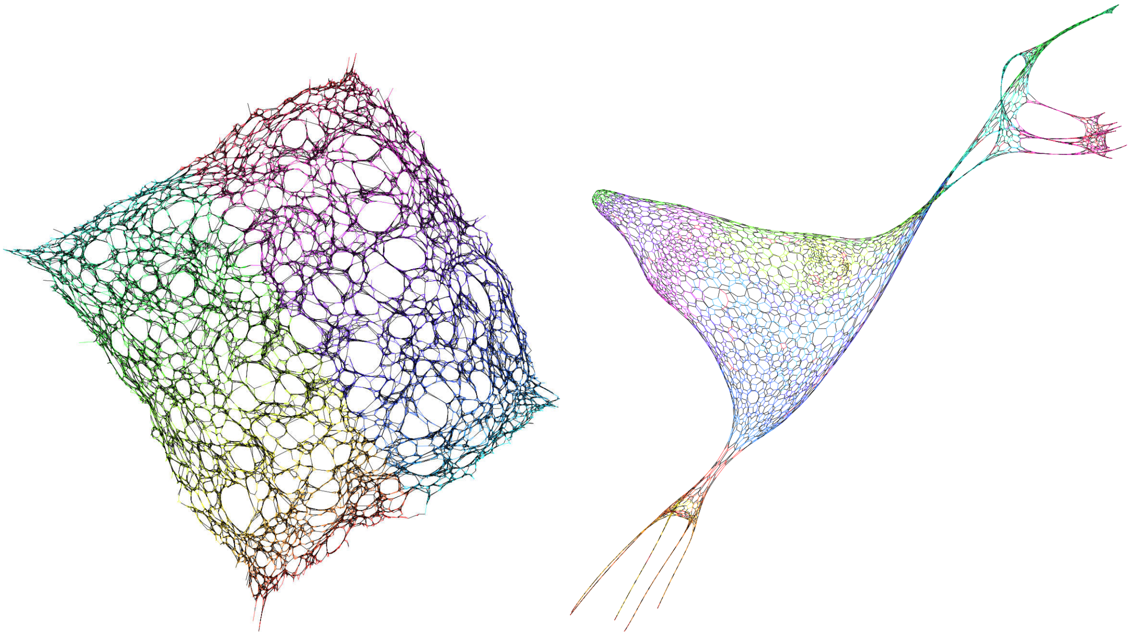


Figure 6.3.: Finished drawings with the clustering graph drawing algorithm, of the rgg_n_2_15_s0 (a) and commanche_dual (b) graphs. Colors encode the different initial clusters used in the algorithm.

6.2. Porting

While the graph generation algorithm was implemented inside NetworKit, the graph drawing algorithm implementation is a stand-alone C++ program.

The considered graphs consist of up to 365 050 edges and therefore fit into the memory of the Xeon Phi without any problems. This greatly simplifies the porting process as the entire calculation can be run on the Xeon Phi, instead of offloading parts of it from the host system, which we observed to be a major performance impediment when large amounts of data have to be transferred.

The *scons* build file `SConstruct` was adapted to compile for the Xeon Phi, as can be seen in Code Block 6.1.

A major obstacle again was the compilation of the source code libraries. Argtable, the command-line option parsing library used, could be compiled with the following

Code Block 6.1 SCons instructions for Xeon Phi compilation with the Intel C/C++ compiler

```
env.Replace(CC = 'icc')
env.Replace(CXX = 'icpc')
env.Replace(AR = 'xiar')
env.Replace(LD = 'xild')
env.Replace(LINK = 'icpc')
env.Append(CCFLAGS = '-mmic')
env.Append(CXXFLAGS = '-mmic')
env.Append(LINKFLAGS = '-mmic')
env.Append(CCFLAGS = '-openmp')
env.Append(CXXFLAGS = '-openmp')
env.Append(LINKFLAGS = '-openmp')
```

command, but required minor fixes in the source code to compile with the Intel compiler:

```
CXX=icpc CC=icc CXXFLAGS=-mmic CFLAGS=-mmic ./configure
--host x86_64-k1om-linux && make
```

To find the functions `isspace` and `toupper`, the `<ctype.h>` header had to be included. In some occurrences of pointer assignments Intel's compiler is more strict than GCC and Clang, so `result = malloc(len);` had to be changed to `result = (char*)malloc(len);` in a few occurrences.

The second dependency of the code was *Cairo*, to render the final graph drawing. The dependencies of *Cairo* itself, *zlib*, *pixman1* and *libffi* could be compiled in a similar manner to *argtable*. Unfortunately a bug in the integration of libffi to glib, the last Cairo dependency, prevented a compilation for the Xeon Phi architecture.

Instead the drawing step was skipped, thereby eliminating the Cairo dependency. The actual drawing can be performed just as well on the host system using the calculated graph layout.

As a consequence of the lack of major dynamic allocations during the execution of the algorithm, an alternative *malloc* implementation, like Intel TBB's *malloc*, has a much smaller effect.

The code contained a calculation $dist_q = dist^q$, which was useless as q was constantly 1 in the current implementation of the algorithm. This simple optimization in the innermost hot loop speeds up the execution by a factor of 1.4.

Reducing the accuracy of calculations by using 32bit floats instead of 64bit doubles is another potential route that can be used for speedups, but was not employed. This allows for the Xeon Phi's 512 bit SIMD unit to work with 32 values at once instead of just 16.

In Code Block 6.2 an extraction of the inner loop of the algorithm's implementation is depicted, calculating the distances for a single vertex. Executed as a separate single-threaded program, the code achieves a factor 6.1 speedup by auto-vectorizing

Code Block 6.2 Extraction of the inner loop of the graph drawing algorithm, which is involved in the local optimization of the clusters

```
typedef unsigned int EdgeID;

struct refinementNode {
  double partitionIndex;
  double x;
  double y;
};

refinementNode node;
unsigned int nodeID;
std::vector<refinementNode> targets(NUMBER_OF_NODES);

double n_S_x = 0;
double n_S_y = 0;

for(EdgeID target = 0, end = NUMBER_OF_NODES; target < end;
    ++target) {
  if( nodeID == target ) continue;

  // calculate the distance of node and targets[target]
  double diffX       = node.x - targets[target].x;
  double diffY       = node.y - targets[target].y;
  double dist_square = diffX*diffX+diffY*diffY;
  double dist        = sqrt(dist_square);

  n_S_x += diffX/dist;
  n_S_y += diffY/dist;
}
```

the code, comparing `-O3 -no-vec` with `-O3`. This comes remarkably close to the optimally possible factor 8 speedup, considering that we use 64bit double numbers and use a 512bit SIMD unit for vector calculations. The success of automatic loop vectorization can also be analyzed using the Intel compiler option `-vec-report=7`, which reveals a potential speedup factor of 7.0 for the auto-vectorization performed, as seen in Figure 6.4.

This effect can not be observed in the actual program, where vectorization only gives a minor speedup, while parallelization gives a speedup factor of over 80. A possible reason for this is that the vectorization is less advantageous when other calculations have to be run at the same time and when the memory connection is fully busy anyway. Additionally, we are running 4 threads for each of the 60 cores already, further reducing the effect of vectorization.

```
LOOP WAS VECTORIZED
masked strided loads: 2
--- begin vector loop cost summary ---
scalar loop cost: 131
vector loop cost: 18.620
lightweight vector operations: 37
--- end vector loop cost summary ---
```

Figure 6.4.: Relevant part of the vectorization report created by the Intel Compiler

6.3. Results

Graph	n	m	Description	Reference
		Small Graphs		
rgg_n_2_15_s0	32 768	160 240	Random Graph	[33]
commanche	7920	11 880	Helicopter	[33]
rajat06	10 922	18 061	Circuit Simulation	[33]
delaunay_n15	32 768	98 274	Delaunay Triangulation	[34]
		Large Graphs		
luxembourg	114 599	119 666	Road Network	[35]
nyc	264 346	365 050	Road Network	[36]

Table 6.1.: Basic properties of benchmark set

Basic properties of the benchmark set are listed in Figure 6.1.

Graph	Running time with Xeon Phi threads									Host
	1	2	4	8	15	30	60	120	240	32
rgg_n_2_15_s0	201.4	102.1	52.6	27.9	16.5	10.0	7.2	5.9	5.3	3.6
commanche	225.2	110.9	56.5	27.4	15.9	8.2	4.7	3.2	2.6	3.5
rajat06	289.5	147.5	73.8	40.2	21.6	11.8	6.2	4.2	3.5	4.5
delaunay_n15	576.7	289.3	146.1	76.7	41.8	23.7	13.5	9.5	7.8	9.0
luxembourg	11195.5	5711.4	2848.6	1420.3	794.0	409.5	215.4	130.9	89.5	166.5
nyc	–	–	–	15935.9	8585.5	4357.2	2318.0	1401.56	960.0	1845.9

Table 6.2.: Running time (in seconds) of the graph drawing implementation with variable number of threads on the Xeon Phi, as well as 32 threads on the host's 2 Intel Xeon CPUs for reference. A limit of 8 hours was imposed and ”–“ denotes runs that did not finish within that time.

As can be seen in Table 6.2 the running time on a single Xeon Phi is lower than that of the dual-socket Intel Xeon host system containing a total of 16 cores. Especially for large graphs the Xeon Phi gains a large advantage, being up to 1.9 times as fast on the New York City road network. In figure 6.5 we observe that the algorithm does not scale as well on denser graphs such as *rgg_n_2_15_s0* and *delaunay_n15* compared to the sparse graphs.

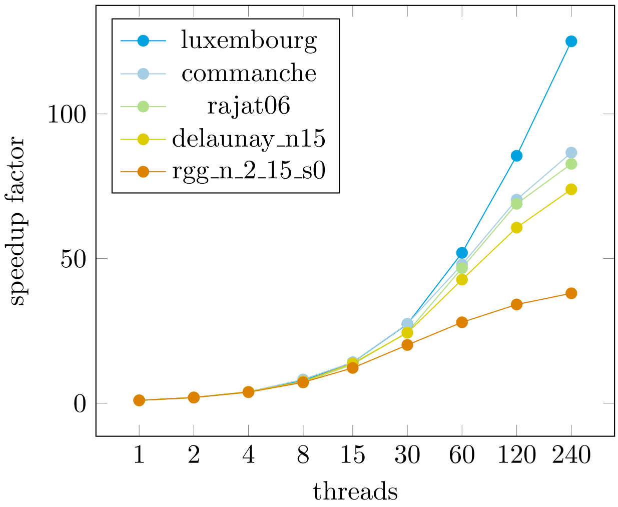


Figure 6.5.: Speedup of the multilevel maxent-stress algorithm on the Xeon Phi

The main advantages of this algorithm is that it is good fit for parallelization and vectorization. Additionally realistic and interesting data sets fit into the main memory of the Xeon Phi, and therefore complicated offloading constructs are unnecessary. Under these conditions an existing implementation of a graph algorithm can be easily ported to the Xeon Phi, leading to a significant speedup.

CHAPTER 7

Conclusion and Outlook

This work studied the process of porting complex algorithms implemented in the form of high level C++ programs, including their libraries, to the Intel Xeon Phi accelerator architecture. Performance benchmarks show that partial offloading of large graph generation data sets does not yield performance improvements, but in graph visualization the Xeon Phi can outperform a two-socket Intel Xeon (Sandy Bridge) system. Various challenges and optimizations were considered and discussed. The similarity of the Xeon Phi architecture to a regular multi-core CPU means that many of the optimizations, such as reduced memory allocations and a lock-free malloc implementation, lead to better performance on CPUs as well.

Not all algorithms are a good fit for porting to the Xeon Phi coprocessor architecture. When memory is accessed more in an irregular manner than as a stream, the benefit of the Xeon Phi's GDDR5 memory disappears and memory latency becomes a limiting factor. Parallelization and vectorization are necessary in concert to scale beyond regular CPUs. Having to offload parts of the problem from the host system to the Xeon Phi is problematic when large amounts of data have to be transferred in either or both directions.

Nevertheless it was possible to achieve higher performance on the graph drawing algorithm with the considerably cheaper Xeon Phi card than with the dual-socket Intel Xeon system, using the same code base. For existing algorithm implementations for CPUs the effort of porting to the Xeon Phi is significantly lower than that of rewriting them for the vastly different GPU architectures. This marks an important trade-off that has to be made, whether to use the more general and portable Xeon Phi, or more parallel and problem specific GPUs. Highly irregular data access patterns, as required for the parallel graph algorithms we studied, pose a problem for the parallel memory access that is vital for high performance computing on GPUs [37]. A direct comparison between similarly optimized implementations of parallel graph algorithms for a GPU and Xeon Phi is a possible future research topic in this area.

The new Xeon Phi coprocessor named Knights Landing is expected to be released at the end of 2015 and promises to solve some of the observed deficiencies and will be interesting for future research. A total of up to 72 more modern Airmont cores promise single core performance increases. The increase of memory from currently up to 16 GB to up to 384 GB will alleviate the need to offload parts of the data. Each of the cores will feature two 512-bit vector units instead of one, with an updated set of SIMD instructions, as well as full x86 compatibility.

Bibliography

[1] "Opte Internet Maps," http://www.opte.org/maps/, accessed: 2015-08-18.

[2] M. E. J. Newman, "The Structure and Function of Complex Networks," *SIAM Review*, vol. 45, no. 2, pp. 167–256, 2003.

[3] D. Papo, J. M. Buldú, S. Boccaletti, and E. T. Bullmore, "Complex network theory and the brain," *Philosophical Transactions of the Royal Society of London B: Biological Sciences*, vol. 369, no. 1653, 2014.

[4] J. P. Thiery and J. P. Sleeman, "Complex networks orchestrate epithelial–mesenchymal transitions," *Nature reviews Molecular cell biology*, vol. 7, no. 2, pp. 131–142, 2006.

[5] J. Abello, F. van Ham, and N. Krishnan, "ASK-GraphView: A Large Scale Graph Visualization System," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 5, pp. 669–676, Sept 2006.

[6] S. Kirmani and P. Raghavan, "Scalable Parallel Graph Partitioning," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. ACM, 2013, pp. 51:1–51:10.

[7] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "CPU DB: Recording Microprocessor History," *Commun. ACM*, vol. 55, no. 4, pp. 55–63, Apr. 2012.

[8] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[9] E. Saule and Ümit V. Çatalyürek, "An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture," in *IPDPS Workshops'12*, 2012, pp. 1629–1639.

[10] A. E. Sariyuce, E. Saule, K. Kaya, and U. V. Catalyurek, "Hardware/Software Vectorization for Closeness Centrality on Multi-/Many-Core Architectures," in *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, May 2014, pp. 1386–1395.

[11] E. Saule, K. Kaya, and U. Catalyurek, "Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi," in *Parallel Processing and Applied Mathematics*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, Eds. Springer Berlin Heidelberg, 2014, pp. 559–570.

[12] F. Wende and T. Steinke, "Swendsen-Wang Multi-cluster Algorithm for the 2D/3D Ising Model on Xeon Phi and GPU," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. ACM, 2013, pp. 83:1–83:12.

[13] C. Benthin, I. Wald, S. Woop, M. Ernst, and W. Mark, "Combining Single and Packet-Ray Tracing for Arbitrary Ray Distributions on the Intel MIC Architecture," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 18, no. 9, pp. 1438–1448, Sept 2012.

[14] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*, 1st ed. Morgan Kaufmann Publishers Inc., 2013.

[15] B. Li, H.-C. Chang, S. Leon Song, C.-Y. Su, T. Meyer, J. Mooring, and K. Cameron, "The Power-Performance Tradeoffs of the Intel Xeon Phi on HPC Applications," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW). IEEE*, 2014.

[16] M. Barth, K. Sweden, M. Byckling, C. Finland, N. Ilieva, N. Bulgaria, S. Saarinen, M. Schliephake, V. Weinberg, and L. Germany, "Best Practice Guide Intel Xeon Phi v1." 2013.

[17] T. Cramer, D. Schmidl, M. Klemm, and D. an Mey, "OpenMP Programming on Intel R Xeon Phi TM Coprocessors: An Early Performance Comparison," 2012.

[18] J. Reinders, "An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi$^{\text{TM}}$ coprocessors," *by Intel Corporation*, 2012.

[19] "Nim Programming Language," http://nim-lang.org/, accessed: 2015-08-18.

[20] "Intel Intrinsics Guide," https://software.intel.com/sites/landingpage/IntrinsicsGuide/, accessed: 2015-08-18.

[21] Q. K. Telesford, K. E. Joyce, S. Hayasaka, J. H. Burdette, and P. J. Laurienti, "The Ubiquity of Small-World Networks." *Brain Connectivity*, vol. 1, no. 5, pp. 367–375, 2011.

[22] D. J. Watts and S. H. Strogatz, "Collective dynamics of'small-world'networks." *Nature*, vol. 393, no. 6684, pp. 409–10, 1998.

[23] D. Krioukov, F. Papadopoulos, M. Kitsak, A. Vahdat, and M. Boguñá, "Hyperbolic geometry of complex networks," *Physical Review E*, vol. 82, Sep 2010.

[24] L. Gugelmann, K. Panagiotou, and U. Peter, "Random Hyperbolic Graphs: Degree Sequence and Clustering," in *Automata, Languages, and Programming*, ser. Lecture Notes in Computer Science, A. Czumaj, K. Mehlhorn, A. Pitts, and R. Wattenhofer, Eds. Springer Berlin Heidelberg, 2012, vol. 7392, pp. 573–585.

[25] M. von Looz, C. Staudt, R. Prutkin, and H. Meyerhenke, "Fast generation of dynamic complex networks with underlying hyperbolic geometry," *arXiv preprint*, 2015.

[26] J. Anderson, *Hyperbolic Geometry*, ser. Springer Undergraduate Mathematics Series. Springer London, 2006.

[27] C. Staudt, A. Sazonovs, and H. Meyerhenke, "NetworKit: An Interactive Tool Suite for High-Performance Network Analysis," *CoRR*, 2014.

[28] "cereal - A C++11 library for serialization," http://uscilab.github.io/cereal/, accessed: 2015-08-18.

[29] N. G. de Bruijn, "A Combinatorial Problem," *Koninklijke Nederlandsche Akademie Van Wetenschappen*, vol. 49, no. 6, pp. 758–764, Jun. 1946.

[30] U. Brandes and C. Pich, "Eigensolver Methods for Progressive Multidimensional Scaling of Large Data," in *Graph Drawing*, ser. Lecture Notes in Computer Science, M. Kaufmann and D. Wagner, Eds. Springer Berlin Heidelberg, 2007, vol. 4372, pp. 42–53.

[31] E. Gansner, Y. Hu, and S. North, "A Maxent-Stress Model for Graph Layout," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 19, no. 6, pp. 927–940, June 2013.

[32] H. Meyerhenke, M. Nöllenburg, and C. Schulz, "Drawing Large Graphs by Multilevel Maxent-Stress Optimization," *CoRR*, 2015.

[33] "The University of Florida Sparse Matrix Collection," https://www.cise.ufl.edu/research/sparse/matrices/, accessed: 2015-08-18.

[34] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, Eds., *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, ser. Contemporary Mathematics, vol. 588. American Mathematical Society, 2013.

[35] D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner, "Benchmarking for graph clustering and partitioning," in *Encyclopedia of Social Network Analysis and Mining*, 2014, pp. 73–82.

[36] C. Demetrescu, A. V. Goldberg, and D. Johnson, Eds., *The Shortest Path Problem: 9th DIMACS Implementation Challenge*, vol. 74. American Mathematical Society, 2009.

[37] F. Dehne and K. Yogaratnam, "Exploring the Limits of GPUs With Parallel Graph Algorithms," *CoRR*, 2010.

Appendix

A. Glossary

AVX

Modern SIMD instruction set extension, appeared in 2011. Together with AVX2 expands SIMD registers to 256 bits and offers new instructions for more parallelism.

Cache Line

Fixed size block which is transferred between the memory and cache.

Cache Miss

When a cache is consulted, but does not contain the data from the desired memory location. The data then has to be fetched from a higher level cache or the main memory and is subsequently stored in the cache.

Cilk Plus

Parallel programming language by Intel based on C and C++, supporting high level multithreading and vectorization constructs. Originally developed as Cilk and Cilk++ by the group of Charles E. Leiserson at MIT.

Clock Frequency

The frequency at which a processor is running, indicates the performance within a CPU family. Power consumption and heat dissipation have placed a limit on clock frequency for the last 10 years.

Complex Network

Graph with non-trivial topological features, such as being scale-free and small-world. These features do not occur in simple networks or random graphs, but often in graphs based on real-world data.

Coprocessor

Processor that is used inside of another computer to support the CPU.

FSM *Full Stress Model*

Model for graph drawing that models physical springs connecting all pairs of vertices of the graph. The physical forces of the springs push and pull the vertices relative to each other and thereby create a layout of the graph, minimizing the amount of stress energy in the system.

GCC *GNU Compiler Collection*

Originally GCC stands for the GNU C Compiler, now also extended to compile C++ and other languages. One of the most widely used optimizing compilers.

GDDR5 *Graphics Double Data Rate, type five*
Memory type used by the Xeon Phi coprocessor. Originally designed for graphics cards.

glibc *GNU C Library*
The standard C library most commonly used in combination with the Linux kernel.

Graph Drawing
Drawing graphs is the problem of representing them in a pictorial layout, making their features more easily perceptible for humans. Furthermore graph drawing can be used as a preliminary step in other applications such as graph partitioning.

Graph Generation
Algorithm that generates graphs based on a set of parameters. Can be used when real-world data can not be used because of privacy or scalability reasons. Graph generation algorithms can be used to create realistic complex networks for the specific use case.

GPU *Graphics Processing Unit*
Stream processor used in computer graphics hardware and increasingly for general purpose computation.

Hyperbolic Geometry
Non-Euclidean geometry with all Euclidean postulates intact except for the parallel postulate. Especially relevant to us is the property of exponential expansion of space of hyperbolic geometry. This means that the area of a circle grows exponentially with the distance from the center.

HT *Hyper-Threading*
Proprietary implementation of simultaneous multithreading (SMT) in Intel's CPUs and also the Xeon Phi coprocessor.

ICPC *Intel C++ Compiler*
Proprietary compiler by Intel for the C++ programming language, which can perform advanced automated vectorizations and supports the Intel Xeon Phi architecture.

IMCI *Initial Many Core Instructions*
Name for the new instruction set available in the Xeon Phi Knights Corner coprocessor, including 512-bit wide SIMD instructions.

In-order Processor
Type of processor design used in old and simple processors and the Xeon Phi coprocessor, executing each instruction in the same order as it is specified in the program. In contrast to *out-of-order execution* which reorders instructions to prevent delays.

Instruction Pipeline
Technique used in CPU design to increase the speed of instruction execution. Breaks up an instruction cycle into a sequence of steps, enabling parallel execution of different steps.

KiB *Kibibyte*
1024 bytes, whereas a *KB* (Kilobyte) means 1000 bytes.

L1 Cache
The level 1 cache of a CPU, the fastest and smallest cache.

L2 Cache
The level 2 cache of a CPU, bigger but slower than the level 1 (L1) cache.

malloc
Standard function in the C programming language to allocate dynamic memory on the heap. Size of the allocation can be specified at runtime.

MIC *Many Integrated Core Architecture*
The architecture of the Xeon Phi coprocessor.

MIMD *Multiple Instruction Multiple Data*
Technique to achieve parallelism. 60 cores and 240 threads are available on the Xeon Phi for MIMD parallelism.

MPI *Message Passing Interface*
Standardized interface for message-passing systems, implemented for many shared and distributed memory systems. Available for C, C++, Fortran and other languages.

Neighborhood
The set of vertices (neighbors) directly connected to a vertex in a graph.

NetworKit
High performance open-source network analysis toolkit written in C++ that exposes the algorithms to Python as well.

OpenMP *Open Multi-Processing*
Set of compiler directives (pragmas), library routines and environment variables for shared memory multiprocessing, available in C, C++ and Fortran. When a compiler does not understand OpenMP or the interpretation of OpenMP directives is disabled, the source code is still interpreted as a valid non-parallel program.

OpenMP Offloading
Executing the main program on a host system, while offloading parts of the program to an accelerator card such as the Xeon Phi.

Scale-Free Network
Network with a power-law degree distribution. The number of nodes with degree k is proportional to $k^{-\gamma}$ for a fixed $\gamma > 0$. This implies that there are a few high-degree nodes, so called *hubs*, and many low-degree nodes. Random graphs on the other hand have an exponential degree distribution.

Shared Memory Architecture
Multiprocessing design where processors or processor cores have access to a globally shared memory. In contrast to distributed memory architecture, in which each processor system has only access to its own memory.

SIMD *Single Instruction Multiple Data*
Processing multiple pieces of data with the same operation to exploit data level parallelism. SIMD instructions are available in modern CPUs.

Small-World Network
Common class of networks, so called after the small-world phenomenon that postulates that all people are connected to each other by only six degrees (or another small number) of separation. For small-world networks a high clustering coefficient is typical as well.

SMT *Simultaneous Multithreading*
Technique for improving the efficiency of superscalar CPUs with hardware multithreading, executing multiple independent threads on a single physical processor core.

TBB *Threading Building Blocks*
High level C++ template library developed by Intel, containing data structures, algorithms and high level algorithmic skeletons for parallel programming.

VPU *Vector Processing Unit*
Unit of the Xeon Phi that processes vector (SIMD) operations.

x86-64
64-bit version of the x86 CPU instruction set, based on, and retaining full compatibility to, the original Intel 8086 CPU from 1978. Most common CPU instruction set in many settings, such as supercomputers, servers and PCs.

x87
Originally an extension to the 8086 processor, today the floating point related instruction subset of the x86 architecture.

Xeon
The name of Intel's current x86-64 CPUs for workstations and servers.

Xeon Phi
Coprocessor architecture by Intel with many cores and a 512-bit wide SIMD unit. Was originally in development since 2006 as the unreleased Larrabee architecture. First products have been released under the name *Knights Corner* in 2012. A new release of Xeon Phi coprocessors is planned for the end of 2015.