UiO : Department of Informatics
University of Oslo

# Xeon Phi

## A comparison between the newly introduced MIC architecture and a standard CPU through three types of problems.

**Joakim Kristiansen**

Master's Thesis, Spring 2016

# Xeon Phi

Joakim Kristiansen

# preface

This thesis concludes my Master's Degree in Informatics: Programming and Networks with the Department of Informatics at the University of Oslo.

I want to thank my supervisor Arne Maus for thorough follow-up with excellent feedback along the way.

I also like to thank my friends and family for the support and encouragement they have given me throughout this period. Special thanks to my father and Lana Vu for proofreading my thesis.

Last, I would like to thank my plush pikachu, Pika, for countless hours of rubberduck sitting. You have served well.

*University of Oslo, April 2016*
*Joakim Kristiansen*

# abstract

As *Moore's law* continues, processors keep getting more cores packed together on the chip. This thesis is an empirical study of the rather newly introduced *Intel Many Integrated Core (IMIC)* architecture found in the Intel Xeon Phi. With roughly 60 cores connected by a high performance on-die interconnect, the Intel Xeon Phi makes an interesting candidate for *High Performance Computing*. By digging into parallel algorithms solving three well known problems, our goal is to optimize, test and compare run times with a regular Xeon processor. Results and their evaluations will be presented along the way, before a conclusion is drawn in the end. We also present limitations for the Intel Xeon Phi encountered from the tests.

# Contents

**1 Introduction** **1**
1.1 Background . . . 1
1.2 Motivation . . . 2
1.3 Structure of thesis . . . 2

**2 The Intel Xeon Phi coprocessor** **5**
2.1 Architecture . . . 6
2.1.1 Core . . . 7
2.1.2 Pipeline . . . 8
2.1.3 Memory model . . . 12
2.1.4 Ring Interconnect . . . 20
2.1.5 Vector Processing Unit . . . 22
2.2 Compared with Intel® Xeon® . . . 25
2.3 Summary . . . 25

**3 Software Architecture on Xeon Phi** **27**
3.1 Operating System . . . 27
3.1.1 Symmetric Communications InterFace . . . 29
3.1.2 Network Abstraction . . . 29
3.1.3 Tools & Apps . . . 29
3.2 Programming Models . . . 34
3.2.1 Native on host processor . . . 34
3.2.2 Offload . . . 34
3.2.3 Symmetric . . . 35
3.2.4 Native on coprocessor . . . 35
3.3 Summary . . . 36

**4 Matrix Multiplication** **37**
4.1 Problem Definition . . . 37
4.2 Naive Solution . . . 38
4.3 Optimization . . . 40

4.3.1 Transposing Matrix B . . . . . . . . . . . . . . . . . 41
4.3.2 Vectorization . . . . . . . . . . . . . . . . . . . . . . 43
4.3.3 Blocking . . . . . . . . . . . . . . . . . . . . . . . . 46
4.3.4 Data Alignment . . . . . . . . . . . . . . . . . . . . 49
4.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . 52
4.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . 54

**5 Quicksort** **55**
5.1 Problem Definition . . . . . . . . . . . . . . . . . . . . . 55
5.2 Naive Solution . . . . . . . . . . . . . . . . . . . . . . . . 57
5.3 Optimization . . . . . . . . . . . . . . . . . . . . . . . . 61
5.3.1 Better Pivot Element . . . . . . . . . . . . . . . . . . 62
5.3.2 Full Parallel Quicksort . . . . . . . . . . . . . . . . . 64
5.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . 67

**6 Traveling Salesman** **69**
6.1 Problem Definition . . . . . . . . . . . . . . . . . . . . . 69
6.2 Problem Data . . . . . . . . . . . . . . . . . . . . . . . . 71
6.3 Naive Solution . . . . . . . . . . . . . . . . . . . . . . . . 71
6.4 Optimization . . . . . . . . . . . . . . . . . . . . . . . . 76
6.4.1 Better Starting Length . . . . . . . . . . . . . . . . . 77
6.4.2 Better Cutoff . . . . . . . . . . . . . . . . . . . . . . 79
6.5 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . 81
6.6 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . 82

**7 Conclusion & Further Work** **85**
7.1 Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
7.1.1 Matrix Multiplication . . . . . . . . . . . . . . . . . . 86
7.1.2 Quicksort . . . . . . . . . . . . . . . . . . . . . . . . 87
7.1.3 Traveling Salesman . . . . . . . . . . . . . . . . . . . 87
7.2 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . 88
7.3 Further Work . . . . . . . . . . . . . . . . . . . . . . . . 89

**A Problems Source Codes** **91**
A.1 Abel computer cluster . . . . . . . . . . . . . . . . . . . 91
A.2 Matrix Multiplication Links . . . . . . . . . . . . . . . . . 92
A.3 Quicksort Links . . . . . . . . . . . . . . . . . . . . . . . 93
A.4 Traveling Salesman Problem Links . . . . . . . . . . . . . 94

**Bibliography** **95**

# List of Figures


2.1 *Microarchitecture Overview for Intel Xeon Phi* . . . . . . . . . . 6
2.2 *Core Overview for Intel Xeon Phi and Intel Xeon* . . . . . . . . 7
2.3 *Pipeline Overview for Xeon Phi* . . . . . . . . . . . . . . . . . . . 9
2.4 *Pipeline Overview for Xeon* . . . . . . . . . . . . . . . . . . . . 11
2.5 *Intel Xeon Phi and Intel Xeon memory model overview* . . . . 13
2.6 *Ring Interconnect overview for Intel Xeon Phi and Intel Xeon* . 21
2.7 *Masked adding of two vectors* . . . . . . . . . . . . . . . . . . . 24

3.1 *Software Architecture Overview* . . . . . . . . . . . . . . . . . . 28
3.2 *Threading Tools for Xeon Phi* . . . . . . . . . . . . . . . . . . . 30
3.3 *SIMD Tools for Xeon Phi* . . . . . . . . . . . . . . . . . . . . . 32
3.4 *Native on host processor model* . . . . . . . . . . . . . . . . . . 34
3.5 *Offload model* . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.6 *Symmetric model* . . . . . . . . . . . . . . . . . . . . . . . . . . 35
3.7 *Native on coprocessor model* . . . . . . . . . . . . . . . . . . . . 35

4.1 *Matrix Multiplication - Simple Solution Runtimes* . . . . . . . . 41
4.2 *Matrix Multiplication - Transpose Runtimes* . . . . . . . . . . . 43
4.3 *Matrix Multiplication - Vectorization Runtimes* . . . . . . . . . 45
4.4 *Cache Reuse: Normal vs Blocking* . . . . . . . . . . . . . . . . . 48
4.5 *Matrix Multiplication - Blocking Runtimes* . . . . . . . . . . . . 49
4.6 *Matrix Multiplication - Aligned Runtimes* . . . . . . . . . . . . . 52
4.7 *Overview of run times for different sizes of* $n$ . . . . . . . . . . 53

5.1 *Example of Quicksort* . . . . . . . . . . . . . . . . . . . . . . . . 56
5.2 *Quicksort - Simple Runtimes* . . . . . . . . . . . . . . . . . . . . 60
5.3 *Quicksort - Pivot picked from various numbers* . . . . . . . . . . 62
5.4 *Quicksort - Better Pivot Element Runtimes* . . . . . . . . . . . . 63
5.5 *Quicksort - Full Parallel Runtimes* . . . . . . . . . . . . . . . . . 65
5.6 *Overview of run times for different sizes of* $n$ . . . . . . . . . . 66

6.1 *Example of TSP* . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
6.2 *Example of recursion tree* . . . . . . . . . . . . . . . . . . . . . . 72

6.3 *Example of task division* . . . . . . . . . . . . . . . . . . . . . . . . 73
6.4 *TSP - Simple Runtimes* . . . . . . . . . . . . . . . . . . . . . . . . . 76
6.5 *TSP - Better Starting Length Runtimes* . . . . . . . . . . . . . . . 79
6.6 *TSP - Better Cutoff Runtimes* . . . . . . . . . . . . . . . . . . . . . 80
6.7 *Overview of run times for different number of cities* $n$ . . . . . . 81

7.1 *Matrix Multiplication Progress, n = 10000* . . . . . . . . . . . . . 86
7.2 *Quicksort Progress, n = 1 billion* . . . . . . . . . . . . . . . . . . . 87
7.3 *Traveling Salesman Progress, n = 18, att532.tsp* . . . . . . . . . . 88

# List of Tables


2.1 *Intel Xeon Phi - Cache characteristics* . . . . . . . . . . . . . . . . 14
2.2 *Intel Xeon - Cache characteristics* . . . . . . . . . . . . . . . . . . . . 15
2.3 *Intel Xeon Phi - TLB characteristics* . . . . . . . . . . . . . . . . . . 18
2.4 *Intel Xeon - TLB characteristics* . . . . . . . . . . . . . . . . . . . . 18
2.5 *Intel Xeon Phi compared to Intel Xeon summary* . . . . . . . . . . 26

# Chapter 1

# Introduction

## 1.1 Background

The **Central Processing Unit (CPU)** is one of the main components within a computer system and may be viewed as the *brain* of the computer. It is the CPU-s job to carry out instructions by performing the necessary arithmetic, control and memory operations as defined by the instruction.

During the last decades the amount of transistors in a dense integrated circuit have close to doubled every two years. This is known as *Moore's law* [1] and was foreseen by Gordon Moore in 1965. However, in 2004 chip development hit a turning point. Powering all transistors is only possible by *Dennard Scaling*, which is still valid. It states that power density stays constant as transistors get smaller. Unfortunately voltage scaling could not keep up and microprocessor designers hit a utilization wall which limited the fraction of a chip that could run at full speed. The remaining unpowered silicon is referred to as *Dark Silicon*. With limited power usage clock frequency also hit the ceiling and the era for multi-core processors arose. Multi-core processors introduced more cores on a chip at the cost of lowered clock frequency. Developers could no longer depend on just hardware for maximum performance, but had to change their own mindset when writing program code. With multi-core processors came an era of very massive parallelism. Vectorization, parallelization and heterogeneity are some of the key elements in this era. [2]

The Intel Xeon Phi product is one attempt that tries to cope with the utilization wall. With its new architecture many low powered cores are connected in an integrated circuit. Due to the high number of cores we are now talking about a many-core architecture rather than a multi-core

architecture.

## 1.2 Motivation

The motivation for this thesis is to look into the fairly new many-core microarchitecture introduced with the Intel® Xeon Phi™and compare it with a regular Intel® Xeon® processor. We would like to check how it copes to various problems. A comparison between the two will be done by looking into three different types of problems. Starting of with a parallel solution for each problem we will measure and reason about to what extent different optimization techniques impacts their run times, before we draw a conclusion based on our empirical studies. The goal of this thesis is not to find better algorithms solving the problems introduced, but rather observe how small optimization impacts the efficiency on the Xeon Phi.

## 1.3 Structure of thesis

The thesis has the following structure.

- **Chapter 2: Intel® Xeon Phi™** describes the essential components of the Intel® Xeon Phi™ coprocessor used throughout this thesis, as well as comparing it with a regular Intel® Xeon® processor.
- **Chapter 3: Software Architecture on Xeon Phi** gives an overview of the programming environment found on the coprocessor as well as the different programming models supported.
- **Chapter 4: Matrix Multiplication** delves into the problem of multiplying two matrices in `C` code, optimizing and comparing run times from the Phi and host Xeon along the way.
- **Chapter 5: Quicksort** delves into the problem of sorting an array of integers using quicksort in `C` code, optimizing and comparing run times from the Phi and host Xeon along the way.
- **Chapter 6: Traveling Salesman** delves into the problem of finding an optimal path from a set of locations in `C` code, optimizing and comparing run times from the Phi and host Xeon along the way.

- **Chapter 7: Conclusion & Further Work** will summarize, discuss and draw a conclusion from found results in the three introduced problems. A section about further work is also included.

- **Appendix: Problems Source Codes** will list links to complete source code for all programs referred to in the thesis.

# Chapter 2

# The Intel Xeon Phi coprocessor

Intel® Xeon Phi™ is the brand name for the family of products using the relatively newly introduced **Intel® Many Integrated Core (Intel® MIC)** architecture. It is worth noticing that we are now talking about a *many core* architecture rather than *multi-core* architecture. That is, we are at a bridge between a multi-core -and cluster system. As of writing, there have been launched two generations of the MIC architecture; **Knights Corner (KNC)** and **Knights Landing (KNL)**. A third one, **Knights Hill (KNH)**, has also been announced [3], but is currently under development. Each generation is an enhanced version of its precedent, which implies that later generations surpass previous ones in performance. However, as KNL still not is available on the market, the Xeon Phi 5100 series will be used as our coprocessor throughout this thesis. It is one of the coprocessor families using the KNC product. As mean of comparison a regular Xeon processor will be used. More specifically the Intel® Xeon® Processor E5-2670.[1] This is the host node for the coprocessor and it features the Sandy Bridge EP microarchictecture released by Intel in 2012.

In this chapter we will delve into the architecture of the MIC. A high level understanding of the cores and how they are interconnected will be introduced. The cache organization model, memory model and the vector processing unit will then be further examined. Along each subsection a comparison with the Xeon is reviewed. Finally, we will sum up the comparisons in its own section.

[1] We will refer to this specific model as only Xeon for the rest of this chapter

## 2.1 Architecture

As stated above does the Intel Xeon Phi, hereby referred to as just the Phi, use the Intel® Many Integrated Core (Intel® MIC) architecture. Each MIC must be attached to an Intel® Xeon® processor-based host platform via a PCI Express bus interface. With the KNL product the MIC architecture will be available as a standalone CPU too. However, in contrast to a GPU program code can be executed fully native on the coprocessor.

different from a GPU can programcode be executed fully native on the coprocessor. The different execution paradigms, coding environment and how to construct optimized code for the coprocessor will be addressed in the next chapters.

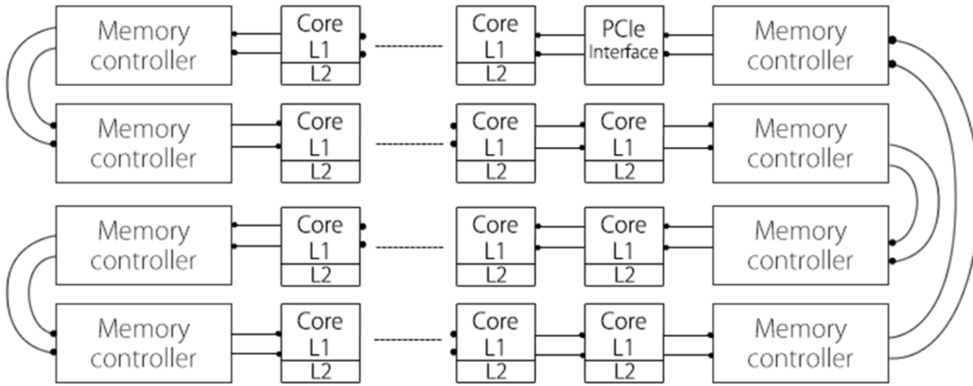


**Figure 2.1:** *Microarchitecture Overview for Intel Xeon Phi*

Figure 2.1 shows a high-level overview of the microarchitecture found on the coprocessor. In the coprocessor there are 61 cores resident in a ball grid array layout on the silicon-chip, connected by a high performance on-die bidirectional interconnect. Each core is instrumented with 4 hardware threads, its own L1 and L2 cache, a 512 bit wide vector processing unit and a set of different interfaces including the core ring interface. In addition there are 8 memory controllers supporting up to 16 memory channels, a system management controller that handles monitoring and tracking of card-level information and it boots a Linux based OS image with as few changes possible from the standard Linux kernel source code. Due to running an OS inside, which may take up a core to service hardware and software requests like interrupts, the coprocessor often end up with 60 cores available for pure computation tasks.

The host processor on the other hand has 8 cores connected by an on-die bidirectional interconnect. Each core comes with its own L1 and L2 cache, a 256 bit wide vector processing unit and one memory controller supporting up to four memory channels. Shared among the cores are a *last level cache (LLC)*, the L3 cache. In our tests two of these Xeon cards will be connected together, basically providing computational power of 16 cores.

### 2.1.1 Core

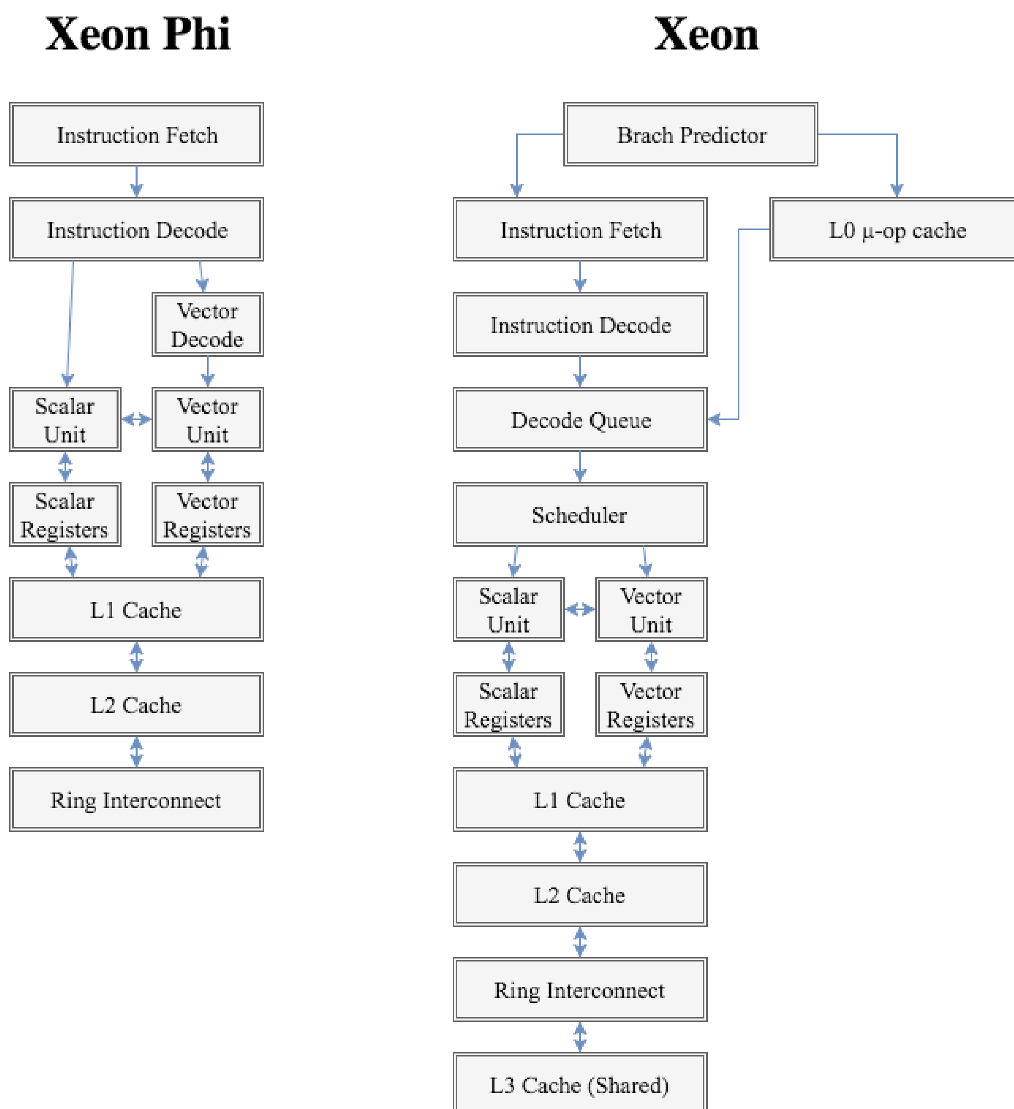


**Figure 2.2:** *Core Overview for Intel Xeon Phi and Intel Xeon*

The coprocessor's core is a modified version of the original Intel® Pentium® P54c processor design and alone it is pretty slow compared to a modern processor. This specific core framework was chosen because it can easily be extended and due to the small footprint it leaves on the silicon board compared to current generations of processors. However, with a clocking rate of only 1.053GHz it is outrun by the regular Xeon, with its rate of 2.6GHz, by a factor of 2.5. The Xeon processor is also able to enter a state named *turbo boost*, which occur when the workload on the processor calls for faster performance. This will increase the frequency operated on in increments of 100MHz, with a peak at 3.3Ghz. However, what makes the coprocessor's architecture so special is that there are a whole bunch of these processors.

Each coprocessor's core follow an in-order execution order model with a 64-bit execution environment. It is capable of executing two instructions per cycle. There are four executions contexts supported per core. The complete architectural state, which contains all the general purpose registers, the floating points registers (ST0-7), segment registers, CR, DR, EFLAGS and EIP, is replicated four times. Certain micro-architectural states like the prefetch buffers, the instruction pointers, the segment descriptors and exception handling are also replicated four times. An overview of the two cores main components can be seen in figure 2.2 on the previous page.

### 2.1.2 Pipeline

Pipeline is an important mechanism used to increase throughput of instructions. As known, handling an instruction is divided into separate parts. Within an instruction cycle there always are an instruction fetch, an instruction decode and an instruction execute. These are handled by different pieces of hardware and as such, waiting for one instruction to complete its cycle yields bad performance. In a pipeline the various pieces of hardware needed to process an instruction are all kept busy at the same time. They perform the needed functions, but on different instructions at a time. For instance can instruction B be fetched into memory while instruction A is being decoded. With modern pipelines each stage usually are divided into several more stages, with the intention of providing better performance. Instruction fetching and instruction decoding are known as the front-end of the pipeline, with the purpose of generating micro instructions. Instruction execution is the back-end of the pipeline.

There exists two execution models to be used with a pipeline; *In-order* -and *Out-of-order (or dynamically)* execution model. The former simply executes

the instructions in sequential order while the latter uses a more complex approach. Out-of-order execution tries to eliminate a certain type of latency causes when the data needed to perform an operation are not available. This introduce advance datastructures and algorithms for queuing and re-ordering of instructions before being executed. Because of this, pipelines used within an out-of-order execution model tend to be bigger than those used in an in-order execution model. Our coprocessor practice a simple in-order model, whereas the host processor uses the other approach.

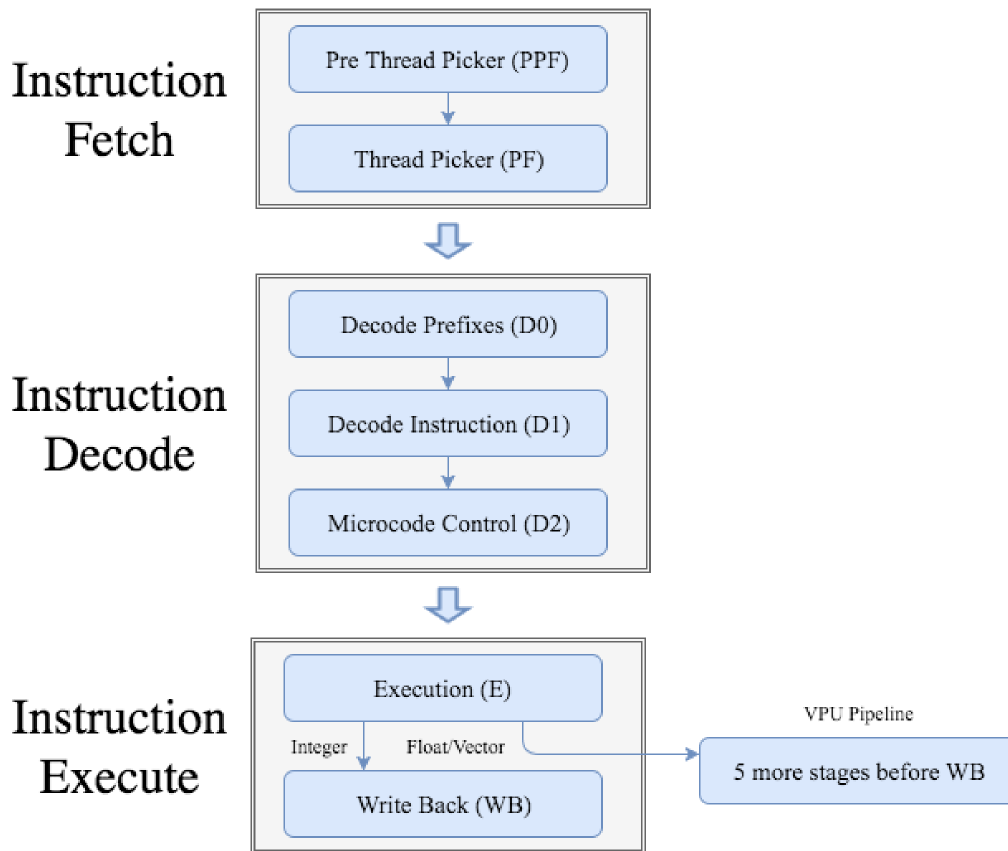


**Figure 2.3:** *Pipeline Overview for Xeon Phi*

The coprocessor's in-order pipeline consist of seven stages for integer instructions plus an additional six stages for the vector pipeline. The vector pipeline will be described in the vector processing unit section *(2.1.5)*. This execution model is employed to reduce size required on chip as well as being far less complex. Since four fully hardware thread contexts are supported, each core can concurrently execute instructions from four threads. Different thread's instruction may be scheduled for each cycle. This helps reducing

the latency effects caused by the vector processing unit and memory accesses. The first two stages, **Pre Thread Picker (PFF)** and **Thread Picker (PF)**, represent the instruction fetch stage. In the PFF stage instructions for a thread context will be prefetched into its prefetch buffer. To support the rapid switch of context, each thread has its own ready to run buffer consisting of two instruction bundles. Since each core is capable of performing two instructions per clock cycle there are two instructions packed in each bundle. Priority to refill a prefetch buffer is given to the thread running at current cycle. It is the picker functions job, the second component in the pipeline, to choose which thread to execute in next cycle. It operates in a simple round-robin manner. Should the executing context have a control transfer[2] to a target not contained in its buffer a miss to the instruction cache will be triggered. This cause the contexts buffer to be flushed and reloaded with appropriate target instructions. Further, if the transfer point is not to be found in the instruction cache a core stall will be initiated, which may result in performance penalties.

The instruction decoder, more specific stage D0 and D1, is simplified to be a two-cycle unit. A result of this change is that the core cannot issue instructions from the same hardware context in two consecutive cycles. That is, if in cycle $\mathbf{N}$ instruction from the first context is issued, then in cycle $\mathbf{N + 1}$ the core can issue instructions from any other context except the first one. Failing to do so will result in a maximum of 50% utilization of the cores potential. It is therefore crucial to use at least two hardware contexts. With multiple contexts utilized a different thread's instruction may be scheduled each cycle.

Next are the instructions sent down to the execution unit. Two instructions can be executed per clock cycle, one on the U-pipe and one on the V-pipe with some restrictions to the instruction type for the V-Pipe. [4] Integer instructions will now be executed in one of the two **Arithmetic Logic Unit (ALU)**-s and finish after write back of results. Vector instructions will continue working until they are done in the vector pipeline six cycles later.

In figure 2.3 on the preceding page one can see some of the main components in the Xeon processor's pipeline. Much effort has been made in creating a good branch predictor to decrease the number of pipeline flushes that occur on a miss-prediction. A branch predictor is also present in each Xeon

[2] A transfer to a target not necessary following the linear flow. For instance a JMP or CALL instruction. See `https://pdos.csail.mit.edu/6.828/2014/readings/i386/s03_05.htm` for further information.

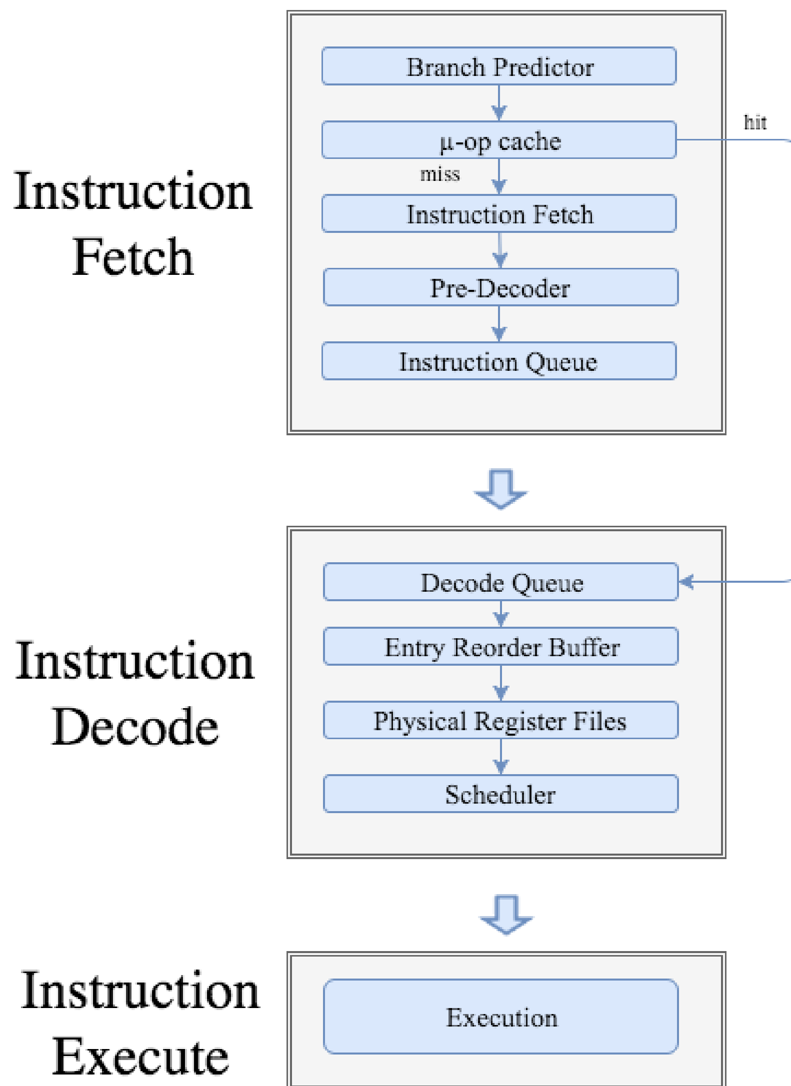


**Figure 2.4:** *Pipeline Overview for Xeon*

Phi core, but not as efficient. Also present is a micro-operation cache, which on hit let the instruction bypass the whole instruction fetch part of the pipeline. Depending on whether a hit occurred or not the pipeline will be 14 -to 19 stage long. The front-end part of the pipeline is in-order, whereas the back-end follow an out-of-order execution. Simplified the back-end use its own bookkeeping system to keep track of the different micro-operations, their execution status and their required elements of data. The micro-operations are executed in any order according to when all data and needed resources are available for the certain operation. For

scalar arithmetic operations three can be done at a time. The bookkeeping and scheduling of micro-operations is fairly complex, and requires multiple dedicated queuing structures such as the instruction queue, the decode queue and the physical register files. Different from the coprocessor is the support for hyper-threading. Hyper-threading is a mechanism only found on out-of-order execution engines. With each of the eight physical cores residing on Xeon's chip, the operating system addresses two logical cores and shares the workload between them when possible. The two logical cores shares resources such as the execution engines, caches and the bus to transfer data, but have their own architectural state. This resembles very much that of the coprocessor. The main difference though, is that a hyper-threaded core can reach peak performance with only one of the logical cores utilized. That is, maximum performance is possible with number of threads equal to number of cores, whereas the coprocessor needs at least two threads per core. This is due to how the out-of-order execution engine takes care of latency hiding in a way that the in-order execution model can not.

### 2.1.3 Memory model

From a core's perspective is the memory abstracted as one collected unit, but in reality it is split. Dividing the memory into separate levels has been around for a long time. Memory accessing is perhaps one of the most expensive operations and when executed will stall the running context for many clock cycles. Due to this, multiple levels of memory are normally found in a processor memory system. With increasing size comes latency penalty. They are distinguishes by their response time. Roughly there are three classes:

- Internal: Processor Registers and Caches
- Main: Systems Random Access Memory
- Secondary Storage: Disk

The last class, Disk, can store magnitude order of *GB* and *TB* of data at the cost of being extremely slow compared to the above ones. It will not be discussed any further. In fact, even if all of a programs instructions and data reside within main memory the need for faster access time still exists. This is where caches play an important role. From L0 to L3 do the caches increase in size and access time. It follows that as much hit as possible in the

lower levels of cache yields faster access time in overall. When the processor fetches instructions and data it will always try fetching from the lowest level of cache. If the requested data is not found in the lowest level of cache, the memory unit will continue looking down the hierarchy. We will only look into the internal -and main memory system part of the coprocessor's and host processor's memory models.

The memory model for the Xeon Phi closely resembles that of the Xeon, but in smaller magnitude. From figure 2.5 we see that it consist of a three level hierarchy comparing to the five level hierarchy found in the Xeon processor. There are two levels of cache, the L1 and L2 cache, as well as the main memory. For the Xeon there are the cache levels L0 to L3 along with the main memory. The L0 cache is a new technique introduced with the Sandy Bridge microarchitecture, which is used to store micro-operations for efficient instruction fetching in the pipeline. Both Xeon versions support Virtual Memory and exploit all mechanisms prefetching, direct memory access and streaming stores.

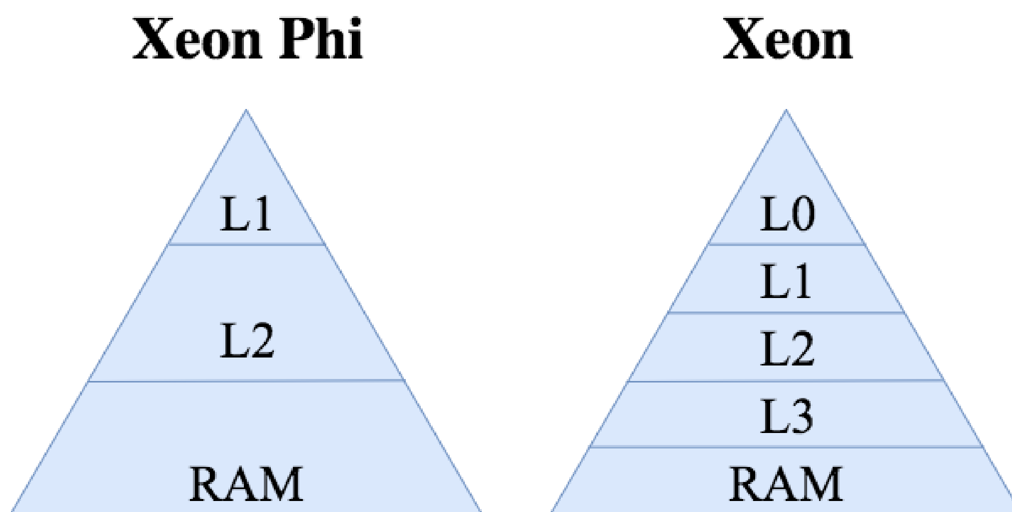


**Figure 2.5:** *Intel Xeon Phi and Intel Xeon memory model overview*

### Cache organization on the Xeon Phi

Included in each coprocessor core are both a private L1 -and L2 cache. Using a distributed **Tag Directory (TD)** mechanism, the cache accesses are kept coherent across the entire coprocessor such that any cached data referenced

remains consistent across all cores without software intervention. The TD is not kept centralized but is broken up into 64 distributed tag directories across the chip, each responsible for maintaining the global coherence state in its assigned cache lines. The replacement algorithm used for both caches is based on a pseudo least recently used (LRU) implementation. Both caches have a 64-byte cache line.

The L1 cache is divided into a 32KB Instruction-cache and a 32KB data-cache. It has 1 cycle access time if it is an integer load, but vector instructions experience different latencies and requires at least 3 clock cycles. The L2 cache is boosted up to 512KB with high speed access to the other cores L2 caches via the interconnect (see *subsection 2.1.4*). It is unified, meaning that both instructions and data are cached. Should target data or code reside in some other core's L2 cache it will not be directly addresses there, but rather fetched into the executing cores own L2 cache. This means that two or more cores sharing data will replicate the data among the cores' various L2 caches. In the highly unlikely case of no shared data or code between any of the cores the total amount of L2 cache is equal to 30.5-MB. Whereas, if every core shared exactly the same code and data in perfect synchronization, then the effective total L2 cache size of the chip is only 512KB. Accessing the L2 cache has 11 cycle best case with an expected idle access time of approximately 80 cycles if the needed data do not reside in the core's own L2 cache. The L2 cache is inclusive, with the whole L1 data cache included. In the old P54c design any miss to the cache hierarchy would be a core-stalling event such that the program would not continue executing until the missing data were fetched and ready for processing. In an Intel Xeon Phi coprocessor core a miss in the L1 or L2 cache does not stall the entire core. Misses to the cache will not stall the requesting hardware context of a core unless it is a load miss, that is when necessary data must be fetched from main memory due to not being found in the cache. Upon encountering a load miss, the hardware context with the instruction triggering the miss will be suspended until the data are brought into the cache for processing. This allows the other hardware contexts in the core to continue execution.

| **Cache** | **Size** | **Ownage** | **Access Time** |
|---|---|---|---|
| L1 Instruction | 32 KB | Private | 1-3 |
| L1 Data | 32 KB | Private | 1-3 |
| L2 Unified | 512 KB | Private | 11-80 |

**Table 2.1:** *Intel Xeon Phi - Cache characteristics*

### Cache organization on the Xeon

Likewise, each core in the Xeon processor have its own L1 and L2 cache. The L1 has equal size for both instruction and data part and an access time of 4 cycles for integer instructions. Vector -and floating point instructions requires 1 or 2 cycles more. The L2 cache is only half the size with its 256KB and not shared among the other cores. This gives less L2 cache per core than on the Xeon Phi. The access time to the L2 cache is approximately 11 cycles. There is also an inclusive L3 cache shared among all the cores. It is 20MB in size and physically partitioned between the cores. Much like the L2 cache in the Xeon Phi, is the L3 cache for the Xeon linked with a bi-directional interconnect ring. Accessing the L3 cache takes around 26-31 cycles, depending on which cache slice needs to be accessed. This cache is, nonetheless, shared such that all cores theoretically can allocate into anywhere within it. In practice this would not be the case, as sophisticated algorithms are used to ensure that a core's needed data are stored close to the core itself. The coprocessor's L2 cache are not shared in such fashion. There is a difference in *allocating* and *querying* for the necessary data. Only allocating into own private L2 cache are allowed. On a L2 cache miss, the other L2 caches have to be queried since one of them might contain a modified version of the needed cache-line.

| Cache | Size | Ownage | Access Time |
|---|---|---|---|
| L1 Instruction | 32 KB | Private | 4-6 |
| L1 Data | 32 KB | Private | 4-6 |
| L2 Unified | 256 KB | Private | 11 |
| L3 | 20 MB | Shared | 26-31 |

**Table 2.2:** *Intel Xeon - Cache characteristics*

### Prefetching

A method to reduce the amount of cache misses and maximize the availability of data to the computation units of the cores, is prefetching. Prefetching is a request to the coprocessor's cache and memory access subsystem to look ahead and begin the comparatively slow process of bringing data we expect in the near future into the much faster to access L1 and/or L2 caches.

Two types of prefetching are provided, software -and hardware prefetch support. Software prefetching is provided in the coprocessor vector processing unit instruction set and can be added as directives to the program

code telling the compiler where, when and what to prefetch. These can also be added as compiler control options when compiling the code. These are advanced mechanism provided and should carefully be used. Typically the compiler will insert L1 and L2 cache prefetch instructions by default based on the access pattern of the code. The other type of prefetching is the L2 cache hardware prefetcher (HWP). Within each core there are 16 data streams managed by the HWP. Once the stream direction is detected then multiple and as needed prefetch requests are made to maintain the data flow. By using both types of prefetching mechanism effectively, the performance can be significantly improved due to reducing the overall probability that all threads of a core will be stalled waiting for a memory operand to be accessible in the cache.

### Main Memory

Along with the L1 -and L2 cache residing in each coprocessor core the chip is instrumented with eight memory controllers distributed symmetrically around the interconnect. They are connecting to GDDR5 memory and are known as the GBOX-es. They are directly interfaced to the ring interconnect at full speed, receiving physical addresses with each request. Each GBOX is responsible for reading data from and writing data to GDDR5 memory and translating memory read and write requests into GDDR5 commands. GDDR5 is modern type of **Random Access Memory (RAM)** based on DDR3, the main memory used in Xeon, and is suited for high-performance computing among others. [5] As with a GPU the Xeon Phi moves a lot of data around due to the high number of cores, which increases the need for more bandwidth. The *G* in GDDR5 stands for *Graphics* and modern RAM designed for graphic cards are able to read and write to memory within the same cycle. However, such devices have a higher latency than normally found on system memory. The actual access time to main memory are nowhere to be found within its software manual, but an empirical study [6] done be researchers from *Delft University of Technology, the Netherlands*, *University of Amsterdam, the Netherlands* and *National University of Defense Technology, China* showed that it takes approximately 302 cycles. Each GBOX comprises three main units: the FBOX (interface to the ring interconnect), the MBOX (request scheduler) and the PBOX (physical layer that interfaces with the GDDR devices). The MBOX can access two 32 bits memory channels that are completely independent of one another, leaving us with 16 memory channels and a total of 8GB storage capacity. It is the MBOX's responsibility to provide connection between the agents in the system and it is connected to both

the FBOX and PBOX. The PBOX communicates with the GDDR memory device and contains the input/output FIFO buffers. There is an interface speed of 5.5 gigatransfers per second (GT/s) and each memory transaction to GDDR5 memory is 4 bytes of data. This results in a total transfer rate of:

$$\texttt{5.5 GT/s} \cdot \texttt{4B} \cdot \texttt{16 channels = 352 GB/s}$$

However, due to complex system factors in the hardware implementation one can expect to achieve an effective peak of 50-60% of the maximum transfer rate. [7]

Comparing with the Xeon with a total transfer rate of only 51.2GB/s much more data can be read to or written from the main memory as long as the data are evenly distributed among the memory channels. The Xeon features four DDR3 memory channels each able to transmit 8 bytes per transaction at a rate of 1.6 GT/s. DDR3 is an older type of memory used primarily as system memory for desktop computers. It can not read and write to memory in the same cycle as the GDDR5 and has about eight time slower data processing rate.

### Virtual Memory

The coprocessor supports **Virtual Memory (VM)** management. As known, VM is a function in the operating system which gives processes the opportunity to use separate address spaces called virtual address spaces. This technique has its benefits. The processes view their memory space as a contiguous one, thinking they have access to the entire memory. It is the operating system's job to map virtual addresses to physical ones. In other words are each process protected in such way that it cannot directly access the physical memory. Nearly all implementations of VM divide the virtual address space into chunks of contiguous memory known as *pages*. Our coprocessor supports VM with 3 page sizes: *4KB, 64KB* and *2MB*. Both 4KB and 2MB are commonly used in a Linux based environment. 64KB on the other hand is not yet supported by the Linux kernel and is rarely used as the choice of page size.

To speedup the translation process from a virtual memory address to a physical one there are added three **Translation Lookaside Buffers (TLB)** to each core. These are special types of cache used only to look up virtual address keys to physical address values. Two TLBs, an instruction

| TLB | Page Size | Entries | Maps |
|---|---|---|---|
| L1 Data TLB | 4 KB | 64 | 256 KB |
| | 2 MB | 8 | 16 MB |
| L1 Instruction TLB | 4 KB | 32 | 128 KB |
| L2 Data TLB | 4 KB, 64 KB, 2 MB | 64 | 128 MB |

**Table 2.3:** *Intel Xeon Phi - TLB characteristics*

TLB (iTLB) and a data TLB (dTLB), cooperates with the L1 cache and a dTLB cooperate with the L2 cache. The characteristics of the TLBs are listed in table 2.3. It is worth mentioning that there are no restrictions for mixing the page size entries (4KB, 2MB) within the L2 TLB. On a TLB miss, a four-level page table walk is performed as usual, stalling the current executing thread for many more cycles than on a hit. A page-walk requires readings of multiple memory location to determine the physical address being requested before it is mapped into the TLB. It it therefore vise to choose a page size most suited for the application running. The use of smaller page size entries will result in less overall memory mapped and available for fast access. Huge page sizes could be a smart choice in certain applications using large data-sets and arrays, depending on their memory access pattern. The large pages can provide significant boost in performance by reducing the number of TLB misses.

| TLB | Page Size | Entries | Maps |
|---|---|---|---|
| L1 Data TLB | 4 KB | 64 | 256 KB |
| | 2 MB | 32 | 64 MB |
| | 4 MB | 32 | 128 MB |
| | 1 GB | 4 | 4 GB |
| L1 Instruction TLB | 4 KB | 12 | 512 KB |
| | 2 MB | 8 | 16 MB |
| | 4 MB | 8 | 32 MB |
| STLB | 4 KB | 512 | 2 MB |

**Table 2.4:** *Intel Xeon - TLB characteristics*

The Xeon is also equipped with several TLBs as shown in table 2.4. There are an instruction and data TLB connected to the L1 caches, as well as a second layer TLB. It allows 1GB huge pages as well as the standard 4KB and 2MB. When using 1GB pages there are support for two entries in the L1 iTLB and four entries in the L1 dTLB. The second level TLB, the STLB, supports only 4KB pages with 512 entries.

### Direct Memory Access

**Direct Memory Access (DMA)** is another feature supported on the mic as well as the host. This is a common hardware function used to relieve the CPU from the burden of copying large blocks of data. To move a block of data, the CPU constructs and fills a buffer, if one does not already exist, and then writes a descriptor into the DMA Channel's Descriptor Ring. A descriptor describes details such as the source and target memory addresses and the length of data in cache lines. There are 8 DMA channels operating simultaneously, each with its own independent hardware ring buffer that can live in either local or system memory. Transfers are supported in either directions (host / coprocessor) and can be initiated by either side. The DMA block operates at the core clock frequency and the following movements of data are allowed:

- From GDDR5 memory to system memory.
- From system memory to GDDR5 memory.
- From GDDR5 memory to GDDR5 memory.

### Streaming Stores

In general, in order to write to a cache line, the Xeon Phi coprocessor needs to read in a cache line before writing to it. One problem with this implementation is when the written data is not reused in the nearest future. Then we unnecessarily take up the bandwidth for reading non-temporal data, and waiting for the LRU policy to evict the read cache-lines gives no benefit. Evicting those cache-lines sooner can make way for other data that are more important. The algorithm 1 shows an example. Here we have three vectors, $A$, $B$ and $C$ of size $N$. We see that vector $A$ is read unnecessary in each iteration of the loop since the value will not be further pursued.

**Algorithm 1** *Example of where Streaming Stores can be useful*

$A[N], B[N], C[N]$
**for** $i \leftarrow 1, ..., N$ **do**
  $A[i] \leftarrow B[i] + C[i]$
**end for**

To avoid this a mechanism known as streaming stores is supported. It is an instruction that do not read in data if the data is a streaming store and can

operate on data on any cache-line multiple size. This is also supported on the Xeon, but specific with the new instruction set extension for the Xeon Phi are streaming stores instructions targeted for the new larger vector processing unit.

### 2.1.4 Ring Interconnect

The interconnect is implemented as a high bandwidth bidirectional ring. All L2 caches, GDDR5 memory controllers and distributed tag directories are directly connected to the ring interface. Both the memory controllers and tag directories are uniformly distributed along the ring to provide a smooth traffic characteristic. An overview of the ring interconnect can be seen on figure 2.6 on the next page. Beside what are shown does the PCIe interface chip reside on the ring too. It is used to connect to the PCIe bus to communicate with the host. However, we notice that the idea of using a ring structure is not newly introduced with the Xeon Phi as the Xeon also has this feature. Here the large L3 cache is shared among the eight cores.

As seen in figure 2.6 on the facing page does each direction consist of five independent rings. The first and largest is the data block ring. It is 64 bytes wide to support the high bandwidth requirement caused by the large number of cores. The next two rings, the address rings are much smaller and are used to send read/write commands and memory addresses. Finally, the two smallest rings are the acknowledgment rings, which sends flow control and coherence messages.

When a core triggers a L2 cache miss, an address request will be sent on the address ring to the tag directories. Should the requested data block be found in another core's L2 cache, a forwarding request is sent to that core's L2 cache over the address ring and the requested data block will be forwarded on the data block ring. Should the requested data block not reside in any of the other caches, a memory address is sent from the tag directory to the memory controller. There is an all-to-all mapping between the tag directories and the memory controllers, and the addresses are evenly distributed across the memory controllers. This eliminates hot-spots and provide an uniform access pattern. Lastly, the memory controller will fetch the requested data block from main memory and it is returned back to the core over the data ring. Adding up, this process generates one data block transfer over the data block ring, two address requests transmitted over the address ring and two acknowledgment messages transmitted over the acknowledgment ring. Since the data block ring is increased in size to support the required data bandwidth, we need to increase the number

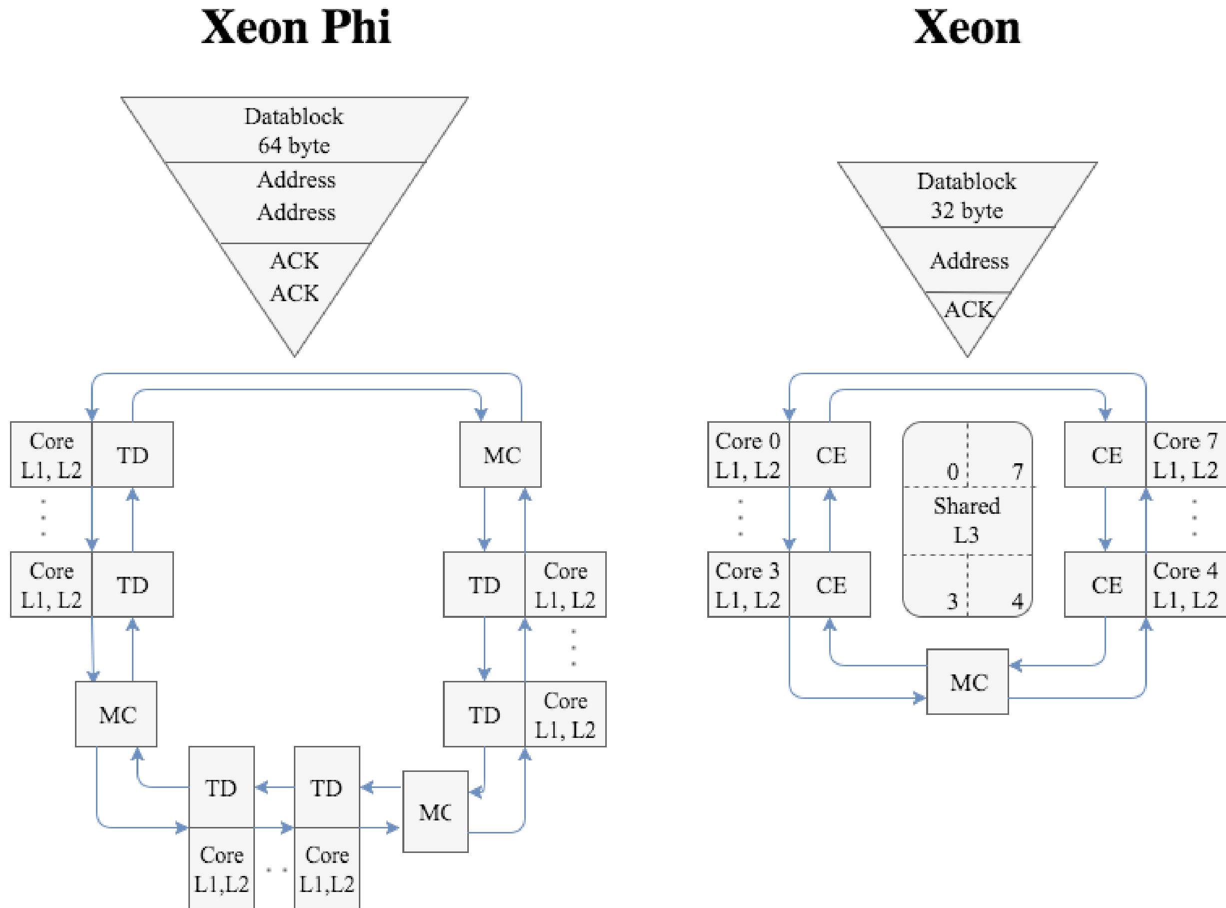


**Figure 2.6:** *Ring Interconnect overview for Intel Xeon Phi and Intel Xeon*

of address and acknowledgment rings by a factor of two matching the increased bandwidth requirement caused by the higher number of requests on these rings. This is why each direction is supplied with two address -and acknowledgement rings.

Only three rings are provided in each direction for the Xeon's interconnect. There are one datablock ring which provide a transfer rate of half a cache-line, 32 bytes, one address ring and one acknowledgement ring. All cores are connected to the ring via a **Coherence Engine (CE)**. All core transactions that access the LLC are directed from the core to the CE via the ring interconnect. Just like the TD, does the CE maintain coherence between the cores. They are also responsible for routing data from the LLC to the requesting core.

### 2.1.5 Vector Processing Unit

Modern processors most likely include a **Vector Processing Unit (VPU)** component. The executing operations are divided into two categories, scalar operations and vector operations. The former are operating on pair of scalar variables. These can for instance be two integers, two floats or two doubles. Scalar operations are performed in the ALU-s. However, many programs have independent data from similar context stored in one dimensional arrays, or vectors as they are called. As with scalar variables, one often needs to combine two vectors together with some sort of operator. In the simplest case this can be to add or subtract one from the other. It is here the VPU comes in handy. Given three vectors $A$,$B$ and $C$ of size 10, we would like to add $A$ and $B$ and store the result in $C$. This would be done as 10 scalar operations if a single ALU was to be used as shown in algorithm 2. Here the instruction *add* must be fetched and decoded every iteration and each iteration three memory address translations are needed.

**Algorithm 2** *Example of scalar adding of two vectors*

$A[10], B[10], C[10]$
**for** $i \leftarrow 1, ..., 10$ **do**
  $A[i] \leftarrow B[i] + C[i]$
**end for**

To a vector processor this task look quite different. The instruction is read and decoded once, only three memory translations are needed and the elements within the vectors are added in parallel. Algorithm 3 illustrate the procedure. Handling multiple data with a single instruction is known as a **Single Instruction Multiple Data (SIMD)** command.

**Algorithm 3** *Example of vector adding of two vectors*

$A[10], B[10], C[10]$
$A[0, ..., 9] \leftarrow B[0, ..., 9] + C[0, ..., 9]$

Introduced with Intel Xeon Phi there is a brand new VPU capable of processing 512 bit at a time. This 512-bit SIMD instruction set is officially known as **Intel® Initial Many Core Instructions (Intel® IMCI)**. Thus, the VPU can perform 16 single-precision or 8 double-precision elements per clock cycle. This correspond to support for integers (both 32-bit and 64-bit), floats (32-bits) and doubles (64-bits). No lower granularities, such as byte or short, are supported. It implements a different **Instruction Set Architecture (ISA)** with 218 new instructions

compared to those implemented in the Xeon family of SIMD instructions, which means it is not backward compatible with previous generations such as AVX (featured on our Xeon) and SSE. To utilize the VPU is just as important as using many threads for performance gain. The VPU contains 128 512-bit vector registers divided equally among the threads, eight new mask registers and a status register (VXCSR) per thread and an **Extended Math Unit (EMU)**. It is fully pipelined and can execute most instructions with 4-cycle latency with single-cycle throughput. One vector can be read/written per cycle from/to the vector register file or data cache.

The vector pipeline consist of six stages: *D2, E, VC1, VC2, V1-V4, WB*. The first stage D2 will decode the vector instruction, then in stage E a check for data dependency is performed by the VPU. Should one be found between two consecutive instructions latency will increase. Stages VC1 and VC2 will load in necessary data and the operation is performed in the four cycle V1-V4 stage. Finally the results are written back to memory in the WB stage.

**Advanced Vector Extensions (AVX)** used in the Xeon only provide a 256-bit SIMD instruction set. They are, however, treated as simple instructions by the decoder and do not need their own pipeline. Both scalar and vector instructions goes into the same scheduling queue in the decoder for the Xeon's pipeline.

### Vector Mask Registers

The eight mask registers are 16 bits wide, used by the compiler in a variety of ways and control the update of vector registers within a calculation. For instance can mask registers hold carry bits from integer arithmetic vector operations or comparison results. The primary benefit of masking is that certain loops containing simple conditional statements may be vectorized, which will maintain high data throughput even in some data dependent circumstances. Vector operations done in combination with masking will only perform the operation of locations where the mask bit is set. A simple example will make it easier to understand.

The code in algorithm 4 on the next page add all elements of vector $B$ and $C$ into $A$ if the iteration counter $i$ is an even number. The mask register will be constructed as shown in figure 2.7 on the following page and only where bits are set will values be added together. The other values contained in $A$ on a zero specific location will remain the same after the addition.

**Algorithm 4** *Example of a simple conditional branch*

$A[N], B[N], C[N]$
**for** $i \leftarrow 1, ..., N$ **do**
  **if** EVEN($i$) **then**
    $A[i] \leftarrow B[i] + C[i]$
  **end if**
**end for**

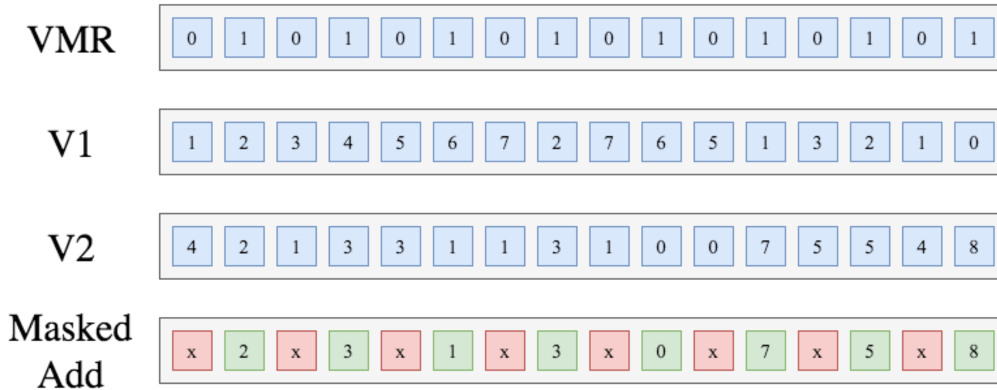


**Figure 2.7:** *Masked adding of two vectors*

### Extended Math Unit

The EMU allows operations such as square root, power, div and log to be executed in a vector fashion. The hardware provide the elementary functions $\frac{1}{x}$, $\frac{1}{\sqrt{x}}$, $2^x$ and $log_2$, which will make it possible to derive for instance the above stated functions. Elementary functions have a latency of 4-8 cycles with 1-2 throughput, whereas the compound functions varies from 8-16 cycles of latency with 2-4 throughput.

### Vector Instructions

As with the Xeon the vector operations are ternary. There are two sources and one destination, which may be different than the sources. The IMCI set also include instructions for **Fused Multiply-Add (FMA)** and scatter/gather, which AVX does not feature. FMA is also a ternary operation, but all three operands are sources with one of them acting as the destination as well:

$$\texttt{source1} \longleftarrow \texttt{source1} \times \texttt{source2 + source3}$$

Scatter and gather instructions can be used when manipulating data with irregular access pattern, where one access only a subset of a vector. A typical gather instruction is

$$\texttt{x[i]} \longleftarrow \texttt{y[index[i]]}$$

where one read values from a dense vector $y$ into a sparse vector $x$ using a list of indices *index*. The scatter instruction is the opposite

$$\texttt{y[index[i]]} \longleftarrow \texttt{x[i]}$$

where data is written to indexes in a dense vector $y$ from the sparse vector $x$. Usually such memory access can not be vectorized, but with gather/scatter instructions sparse locations of memory are fetched into a dense vector register or vice verse, allowing vectorization.

## 2.2 Compared with Intel® Xeon®

A lot of specifications and numbers have been introduced in the previous sections. For consistency and easier lookup, table 2.5 on the next page have been added. Here the most important aspects of what have been introduced earlier are listed.

## 2.3 Summary

The focus of this chapter was to get a good understanding of the new MIC architecture introduced with Xeon Phi. The use of simpler cores allows more of them to fit within a single chip and we are now talking about a many core architecture rather than a multi-core architecture. A high performance on-die bidirectional interconnect ring links the cores together, providing reads from each others L2 caches. Also memory controllers and the pci interface reside on the ring interconnect. Throughout each section, we compared our coprocessor with a regular Xeon, summarizing the specific details.

The next chapter will look at the different programming models supported on the MIC, as well as the programming environment.

| | Xeon Phi 5120 | Xeon E5-2670 |
|---|---|---|
| | **General** | |
| Architecture | Intel Many Integrated Core | Sandy Bridge EP |
| Execution Environment | 64-bit In-order | 64-bit Out-of-order |
| Instruction Set Extension | IMIC | AVX |
| | **Performance** | |
| # of Cores | 61 | 8 |
| # of Threads | 244 | 16 |
| Processor Base Frequency | 1.053 GHz | 2.6 GHz |
| Processor Turbo Frequency | | 3.3 GHz |
| | **Memory** | |
| L0 Cache Size | | 1.5 KB |
| L1 Cache Size | 64 KB | 64 KB |
| L2 Cache Size | 512 KB | 256 KB |
| L3 Cache Size | | 20 MB |
| Max Memory Size | 8 GB | 384 GB |
| Max # of Memory Channels | 16 | 4 |
| Memory Type | GDDR5 | DDR3 |
| Max Memory Bandwidth | 352 GB/s | 51.2 GB/s |
| | **Virtual Memory** | |
| # of TLB-s | 3 | 3 |
| Page Sizes Supported | 4KB, 64KB, 2MB | 4KB, 2MB, 4MB, 1GB |
| | **Features** | |
| DMA | Yes | Yes |
| FMA | Yes | No |
| Scatter/Gather | Yes | No |
| Streaming Stores | Yes | Yes |

**Table 2.5:** *Intel Xeon Phi compared to Intel Xeon summary*

# Chapter 3

# Software Architecture on Xeon Phi

One of of the main benefits with the new MIC architecture is that there are no new programming paradigms. That is, there are no new programming languages or programming language extensions introduced just for the MIC. All the familiar programming models, programming languages and tools found on other Intel products are fully compatible on the Phi. Not only does this provide a friendly coding environment, but it allows easy porting of old programs to the MIC without introducing new bugs.

In this chapter we shall briefly look into the Phi's operating system, then further examine the various parallel programming tools supported, two of which will be used later in the thesis. We will also introduce four types of programming models, *Processor Native*, *Offload*, *Symmetric* and *Coprocessor Native*.

## 3.1 Operating System

Running on the coprocessor is an **Operating System (OS)** based on the standard Linux kernel source code. There are as few changes possible to the original source code, but some have been made in order to adapt it to the coprocessor. This allows the coprocessor to support normal OS functions such as task creations, scheduling and memory management. In figure 3.1 on the following page some of the main components for the coprocessor's software architecture are illustrated. A lot of drivers and other tools have been left out due to being irrelevant for this thesis. A fully detailed overview can be found in Intel's own system software manual for the Phi. [8]

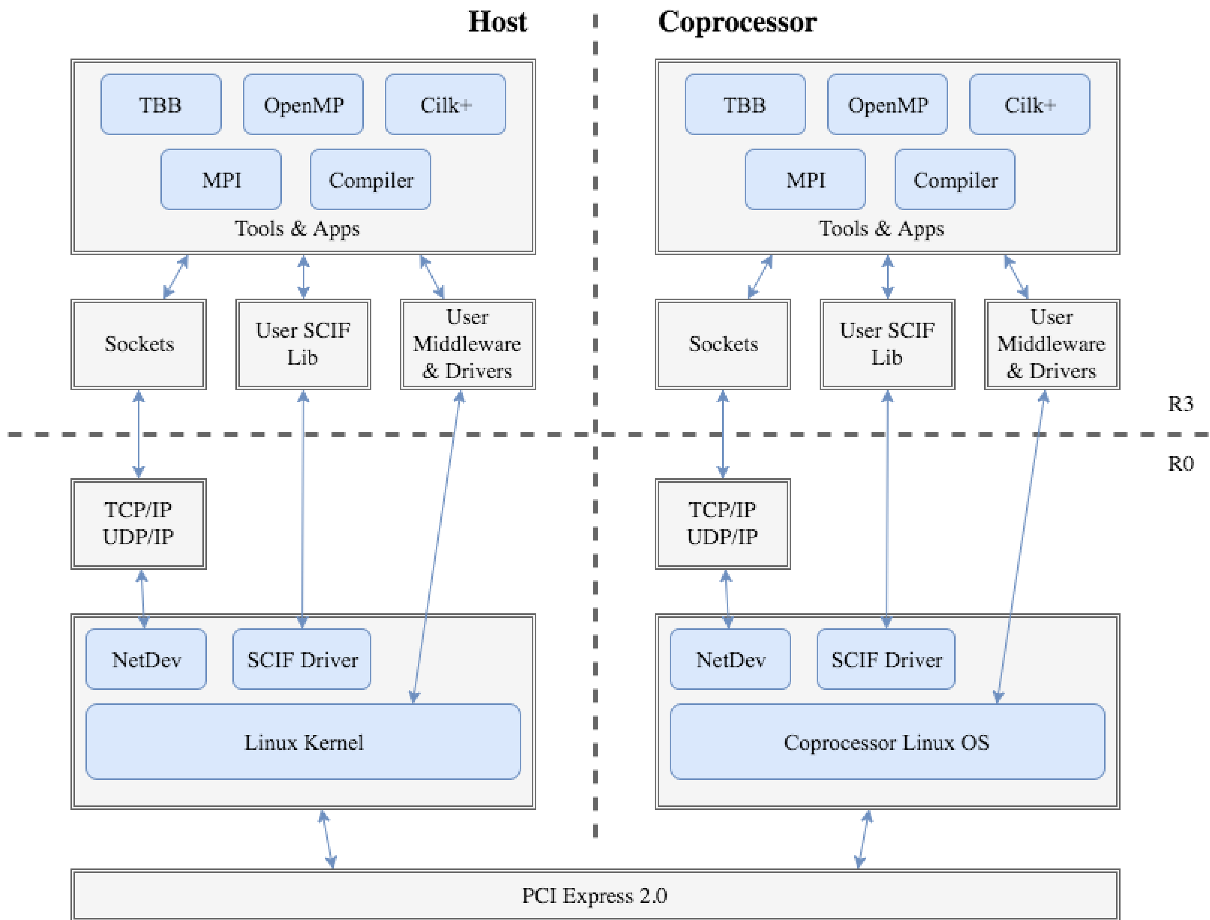


**Figure 3.1:** *Software Architecture Overview*

The figure is divided in four; a left and right side as well as top and bottom halves. On the right side is the coprocessor's architecture and on the left side the host's. The dashed line between top and bottom mark the distinguish between kernel level (R0) and user level (R3) protection rings in Linux. As normally do user level application code and system interfaces run inside ring 3 and more trusted system level functions and drivers run inside ring 0. At the very bottom we find the PCI express bus, which is the main communication medium between the host and coprocessor. The first thing to notice is the symmetry between the two sides. By replicating most of the host's software architecture onto the coprocessor's architecture yield identical functionality, which enables system developers to engage the coprocessor in the same way they engage a regular processor. Most interesting are the various tools and apps supported in ring 3, the symmetric

communications interface and the support to operate as a network node.

### 3.1.1 Symmetric Communications InterFace

SCIF is the backbone for all communication going between the host and coprocessor. It works as a middleware between the software oriented tools and apps found in ring 3 and the physical PCIe bus found in ring 0. By providing an uniform symmetric API it allows the host and coprocessor to communicate easily. SCIF takes advantage of the inherent reliability of the lower layer PCIe and operates on data packets without the need for additional meta-data. As such, it is not a replacement for higher level communication, but rather provide an abstraction from the hardware to these higher levels API-s.

SCIF also utilize the DMA capabilities found on the coprocessor for high bandwidth transfer when larger data blocks need to be transferred and it allows all of the coprocessor memory to be visible to the host or vice versa. This provide the ability to map memory of the host processor or memory on the coprocessor into the address space of a process running on respectively the coprocessor or host processor. These abilities are also scalable, meaning that they still exist in an environment with multiple host processors and multiple coprocessors used.

### 3.1.2 Network Abstraction

Another neat component is the *NetDev driver*. It allows the coprocessor to operate as a network node supporting the TCP/IP protocol. This is done by emulating the PCIe bus to appear as an Ethernet device to the IP layer of the networking stack. With the coprocessor visible as a network node one can access it remotely through *SSH* or *Telnet* and through *Network File System* get access to the host or remote file systems.

### 3.1.3 Tools & Apps

As shown from the *Tools & Apps* box in R3 in figure 3.1 on the preceding page, there are a great number of tools and language extensions available on the MIC. We can further organize these options regarding to threading and vector options, and sort by the level of control provided. Generally more control gives higher complexity. Each and every tool are targeted for all or some of the programming languages *C*, *C++* and *FORTRAN*. In this thesis only C will be used.

The easiest among the options for both vector and task parallelism is **Intel Math Kernel Library (MKL)**. It is an API used to compute highly vectorized and threaded Linear Algebra, Fast Fourier Transforms (FFT), Vector Math and Statistics functions. Once a function is called it will automatically manage task division, thread spawning and gathering of data into one answer. It will also take full advantage of the VPU-s available if they are needed.

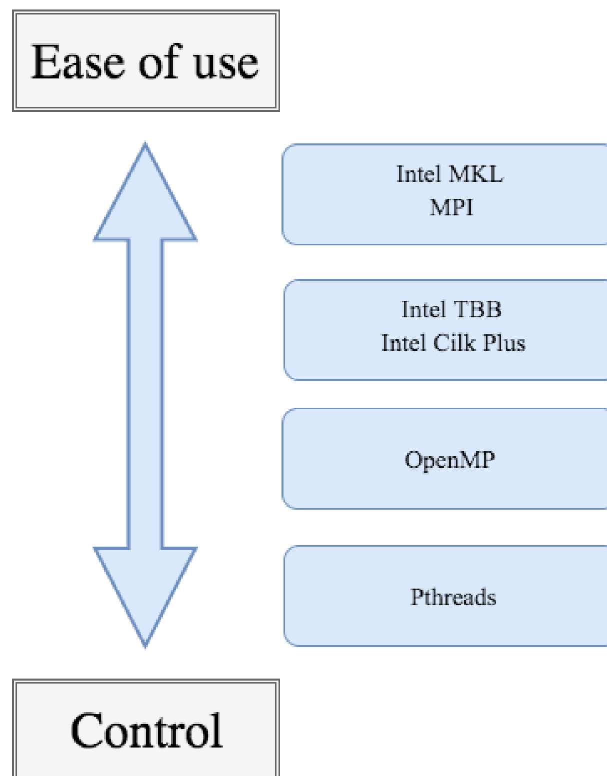


**Figure 3.2:** *Threading Tools for Xeon Phi*

The other options for spawning of threads and task based parallelism are **Intel Cilk Plus**, **Message Passing Interface (MPI)**, **Open Multi-Processing (OpenMP)** and **Pthreads** when using C as the programming language. **Intel Threading Building Blocks (TBB)** are only available for C++. All options are listed in figure 3.2.

### Cilk Plus

The second easiest language extension is the Cilk Plus language extension. For task parallelism only three new keywords are implemented. These

are `cilk_spawn`, `cilk_sync` and `cilk_for`, respectively managing thread creation, thread waiting and loop parallelism.

### OpenMP

Providing a set of compiler directives, library routines and environment variables OpenMP seek to provide task parallelism, mainly for loops. The directives are denoted as

```
#pragma omp construct [clause [clause] ...]
```

and operate on structured blocks, that is blocks containing one or more statements with one point of entry at the top and one point of exit at the bottom. OpenMP rely on a shared address model and threads communicate by sharing variables. It is an abstraction of the underlying low-level pthreads´ API. If not carefully developed, unintended sharing of data can cause race conditions. To control race conditions there have been added directives for synchronization as well.

### MPI

MPI is a specification for a communication protocol used in parallel computing, that defines a standard syntax and semantic for a core of library routines. An MPI program launches a number of processes as demanded by user. These are known as ranks, each with their own address space and unique id. The ranks can communicate either point-to-point or through collective communication via a transaction called *message passing*, in which a copy of the data is transferred from one rank to the others. Although MPI mainly focus on direct communication between the ranks, there are some support for a shared memory model included in version *MPI-3*. [9]

### Pthreads

This is a very low-level API providing full control over management of threads, mutexes and locks, condition variables and thread synchronization. It is the programmer's responsibility to ensure that no data races occur and that the threads communicate as expected.

For vectorization the other options are **Cilk Plus: Array Notation**, **Automatic Vectorization**, **Semiautomatic Vectorization with directives**, **Open Computing Language (OpenCL)**, **Vector Classes** and **Vector Intrinsics** as shown in figure 3.3 on the next page.

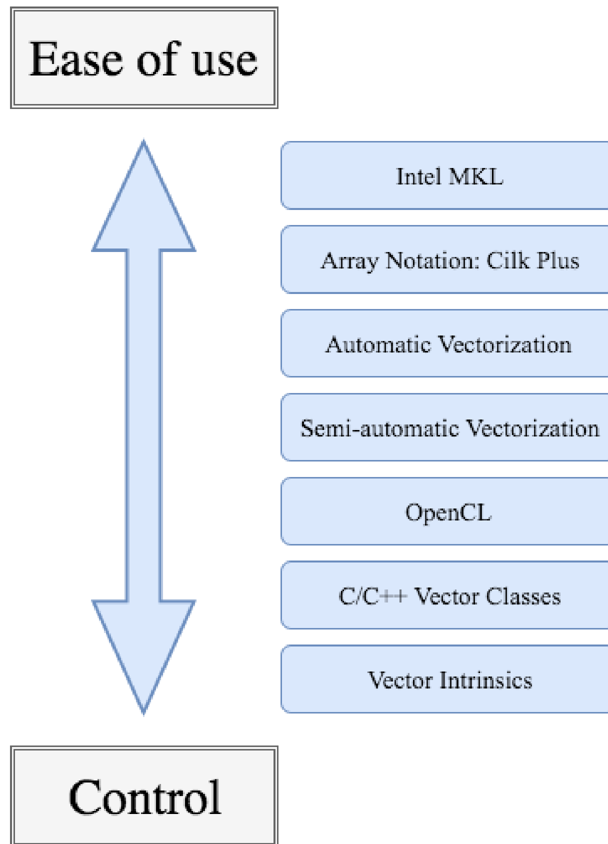


**Figure 3.3:** *SIMD Tools for Xeon Phi*

### Cilk Plus: Array Notation

Cilk Plus is also the second simplest language extension with regard to vectorization. It introduces the syntax `[:]` that delineates an array section. One can then work on vectors with the following syntax:

```
Array-Expression[lower-bound : length : stride]
```

The `lower-bound` and `length` defines the portion to perform the operation on, with `stride` stepping in the indexes. This allow abstraction of vector operations and helps the compiler identify regions of code that should be vectorized. Common operations such as adding two vectors or adding, subtracting, multiplying or dividing an array by a scalar variable are supported.

### Automatic -and Semiautomatic Vectorization

With automatic vectorization the programmer write code in a vectorizable friendly form and the compiler will vectorize the code for SIMD execution without hints. For the compiler, program correctness is paramount and it will make conservative assumptions. That is, function calls to non-vectorized functions, branches in code, non-contiguous memory access pattern, loop dependencies and pointers are all areas where the compiler will be conservative. Branches and loop dependencies can cause discontinuities in SIMD execution and the compiler must assure that pointers do not overlap in memory.

With semi-automatic vectorization, annotations or directives are placed in the code to loosen the assumptions the compiler makes when attempting vectorization. These can indicate mandatory action that must be performed or just be hints for the compiler. The use of such directives in wrong areas can convince the compiler to vectorize where one should not, creating unintended consequences.

### OpenCL

OpenCL is a specification that defines a standard for parallel programming across multiple platforms. The language itself, *OpenCL C Language*, is based on C99 standard and it need to be supported on the underlying operating system and hardware. OpenCL provide a heterogeneous programming model that seeks to take advantage of all system resources, mainly CPU-s and GPU-s, but also any type of hardware compliant with the specification. Its primary use are number crunching, which makes it good for video/image/audio processing, simulations and scientific calculations, financial models and data-parallel algorithms that are computationally significant.

### Vector Classes

These are special classes that provide abstract operations on the underlying SIMD instructions. They are more flexible than Cilk Plus: Array Notation, but does not offer the same simple syntax to delineate a vector. The vector classes are primary divided in Integer and Floating-point vector classes, with a variety of sizes for both.

#### Vector Intrinsic

These are very low-level function that provide direct access to many of the various vector instructions supported on Intel products without the need of Assembly code. There are support for the new instruction set IMCI introduced with Phi as well as AVX and SSE support.

Though, it is important to understand that these tools are not exclusive of one another. Which frameworks to use are very application dependent and sometimes two or more must be combined together to achieve the desired result.

## 3.2 Programming Models

As seen in the previous section several different programming tools and paradigms are supported on the MIC, offering a trade-off in simplicity for user control. In addition to this fine stepped *depth*, the underlying software architecture also allows for a *breadth* variety in programming models. Depending on the level of parallelism in the code, there are four different way to run the application: *Native on host processor*, *Offload*, *Symmetric* and *Native on coprocessor*.

### 3.2.1 Native on host processor

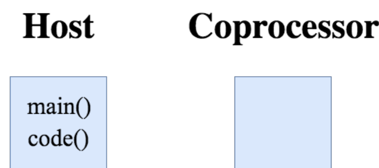


**Figure 3.4:** *Native on host processor model*

With this model all code run on the host processor as illustrated in 3.4. This will typically be used in general parallel computing with lot of sequential code blocks. We will not discuss this approach any further.

### 3.2.2 Offload

In an offload model most code will still run on the host processor, but there exists certain blocks of code that are highly parallizeable. These section

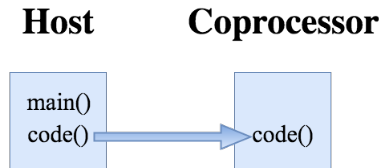


**Figure 3.5:** *Offload model*

will be transferred to and executed on the coprocessor, before the result is sent back to the host processor. The action is illustrated in figure 3.5.

### 3.2.3 Symmetric

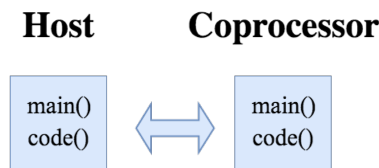


**Figure 3.6:** *Symmetric model*

As seen in drawing 3.6 is the application launched on both the host and coprocessor in a symmetric programming model. Here some sort of cooperative communication is also establish between the two.

### 3.2.4 Native on coprocessor

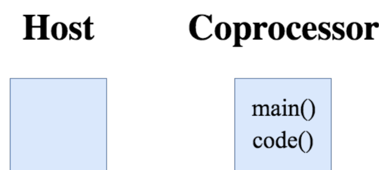


**Figure 3.7:** *Native on coprocessor model*

Last, there is the option of native on coprocessor only model. With this model all program code will be compiled native for -and run on the coprocessor. This approach is desirable only if the code is highly parallizeable and with minimum sequential code and synchronization between the threads. Figure 3.7 show the idea of this method.

## 3.3 Summary

This chapter started off by briefly glancing the OS and some of the components found in the software architecture on the MIC. Further we examined several MIC compatible programming tools including MPI, OpenMP, Cilk Plus and OpenCL, before we in the last section looked at the various programming models supported on the MIC. In the following chapters some of these tools and paradigms will be employed to solve the problems.

# Chapter 4

# Matrix Multiplication

The first problem used in the comparison of the host and coprocessor will be the classic multiplication of two matrices. A matrix is a table of data, or a two-dimensional array, represented by rows and columns. For a matrix of dimensions `n` $\times$ `m`, there are `n` rows, each with `m` columns as illustrated below.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix}$$

Arbitrary data may be stored in the matrix, but in order for multiplication of two matrices to be allowed, elementary arithmetic must be defined between the data elements themselves. Elementary arithmetic involves addition and multiplication among others. In this problem the data type **double** will be used, which has a well defined elementary arithmetic. To avoid overflow caused by multiplying large numbers together, all numbers will be from range 0–1. C is the programming language the problem will be solved in and OpenMP will be the mean of parallelism tool. Only relevant snippets of code are displayed throughout the sections and the full source codes can be found in the Appendix.

## 4.1 Problem Definition

Matrix multiplication is a binary operation taking two matrices and producing one new matrix. There is no unique definition of multiplying two matrices, but we will use the most common one known as the *matrix*

*product.* [10] Given matrix **A** of dimensions n $\times$ m and matrix **B** of dimensions m $\times$ k we will compute the product $\mathbf{C} = \mathbf{AB}$, which will have dimensions of n $\times$ k.

$$\begin{pmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,k} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,k} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix} \cdot \begin{pmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,k} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,k} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,k} \end{pmatrix}$$

The idea is that all m elements along the rows of **A** will be multiplied with the m elements down the columns of **B**. For row i and column j, the multiplied results are summed and become the result for entry $\mathbf{C_{i,j}}$:

$$C_{i,j} = \sum_{k=1}^{m} A_{i,k} \cdot B_{k,j}$$

We will narrow the problem down to operating on square matrices only, that is matrices of equal dimensions n $\times$ n. This implies that all matrices **A**, **B** and **C** will have dimensions n $\times$ n.

## 4.2 Naive Solution

The first implementation and starting point will be a parallel version that computes the summation above for all pairs of i, j $\in [0, n]$, corresponding to all entries $\mathbf{C_{i,j}}$. This is done with a triple nested loop. The first goes through all rows i in **A**, then for each of those rows, all columns j in **B** are iterated over in the second loop. The last and innermost loop sums up the products as it moves along the current row's and column's elements.

```
void multiply(double *A, double *B, double *C, int n){
  int i, j, k;

// Start a team of threads to compute the C elements.
#pragma omp parallel
#pragma omp for collapse(2) private(k)
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {

      // Calculate C[i, j] = A[i, k] * B[k, j],
```

```
      // for all 0 <= k < n
      C[i * n + j] = 0.0;
      for (k = 0; k < n; k++) {
        C[i * n + j] += A[i * n + k] * B[k * n + j];
      }
    }
  }
}
```

**Listing 4.1:** *Snippet from `mm_simple.c`*

The code is listed in 4.1 on the facing page. Some programming oriented aspects of the function needs explanation. The function itself takes three matrices `A`, `B` and `C` and the dimension parameter `n`. All matrices are dynamically allocated at runtime using `malloc()`, but flatten from 2D to 1D, with rows stored consecutively. By doing so, each matrix is stored as a consecutive block of memory and we can access it with a single pointer. Accessing become a little more tricky though, but can be calculated by the formula:

$$\mathrm{Matrix_{2D}[i,j] = Matrix_{1D}[i \cdot n + j]}$$

Next are the two OpenMP directives `#pragma omp parallel` and `#pragma omp for collapse(2) private(k)`. `#pragma omp parallel` will start up a team of threads, each executing the following structured block in parallel. How many threads are defined by the control variable `nthreads-var` associated with OpenMP, which may be set at runtime with the function call `omp_set_num_threads(int n)` or exported as an environment variable `OMP_NUM_THREADS=n` before running the program. The number of threads created are also dependent on available resources and in the case where resources are sparse a smaller number of threads will be used. However, there will never be more threads than defined in `nthreads-var`. After a `parallel` region there is an implicit barrier synchronizing all threads before the master thread may continue.

The next block of code, `#pragma omp for collapse(2) private(k)`, tells the compiler to collapse the next two loop (`i` and `j`) into one loop and then apply the OpenMP `omp for` work division mechanism to split the loop calculations among the current available threads. Conceptually, the `for` loop changes to a single loop that executes as `for(ij = 0; ij < n * n; ij++)` with the associated similar mapping for the use of `i` and `j` inside the loop body. This enables each thread to be assigned larger portion of

data, since the workload split among the threads increased from n to n × n. The private(k) clause will let each thread get its own copy of the variable k, removing race-conditions that else might occur on this variable. The variables i and j also need to be local to each thread for the same reason, but this is automatically handled by collapse itself.

Let us compile and run the code native on both the host Xeon and the Phi. We will be using Intel's own compiler, namely icc. To specify the use of OpenMP the additional flag -openmp (or -fopenmp with gcc) must be included. To let the compiler optimize as much as possible, the flag -O3 is added as well. The compilations are identical except for the Phi, where an additional -mmic flag must be added to specify that this code is targeted for the coprocessor only. The compilation also include all necessary source files and yields an output file defined by the flag -o. If not stated, all source code files in this chapter will be compiled in the same manner. We use the following commands:

```
$ icc -openmp -O3 mm.c mm_simple.c -o host_simple.out
$ icc -mmic -openmp -O3 mm.c mm_simple.c -o mic_simple.out
```

The output from running on both architectures with dimensions of n=1000, 5000, 10000, 15000 is listed in the plot 4.1 on the next page. Only the median value out of three runs are used for each value of n along the x-axis. The y-axis shows the run times in seconds, scaled logarithmically with values approximately doubling at each step. The same plot settings will be used throughout the thesis. We see that the Phi's performance is terrible. Even though all 240 threads are being used, it takes much longer time to finish than the host Xeon with only 32 threads running. As n increases the runtime ratios between the host Xeon and the Phi do too, resulting in quite a big gap. For n=15000, the Phi did not even finish after two hours of running and the program was aborted. There are several reasons for this behaviour, some of which will be addressed in the following section.

## 4.3 Optimization

In the following subsections the code will be further analyzed, with the intention to utilize the resources found on both architectures better. Four optimization methods will be applied to the code. They are *transposing*,

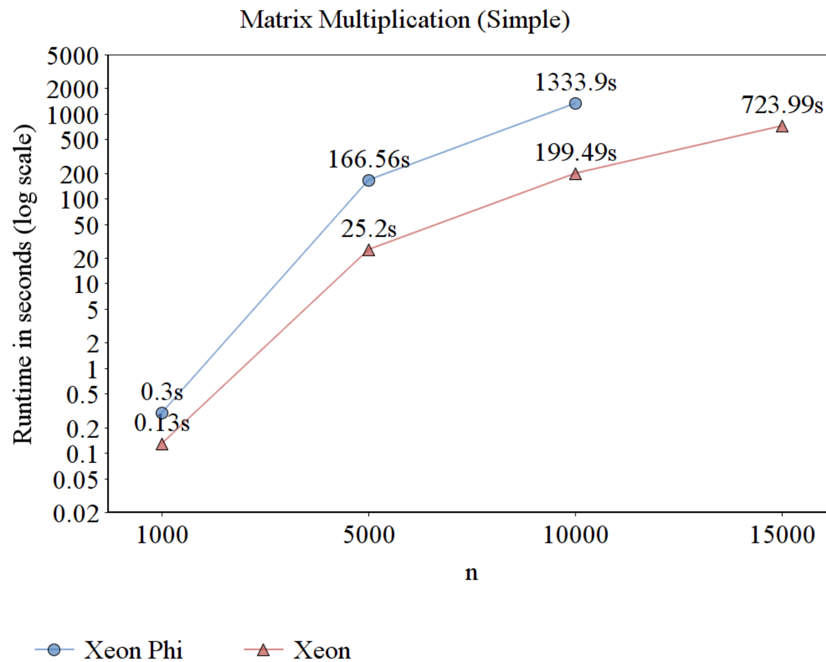


**Figure 4.1:** *Matrix Multiplication - Simple Solution Runtimes*

*vectorization*, *blocking* and *alignment*. The optimizations will be built on each other, meaning that previous techniques are included in each step of optimizing. For example, when applying blocking, vectorization and transposing are applied too.

### 4.3.1 Transposing Matrix B

Looking at the line `C[i * n + j] += A[i * n + k] * B[k * n + j];` in the innermost loop of listing 4.1 on page 38, or more specifically how matrix `B` are being accessed, one notice that each element are retrieved by some multiple of `n`. Even though this pattern is predictable it has a huge downside. Prefetch instructions can not be issued, or will load unnecessary data, due to memory being accessed in a non-continually way, causing cache miss on almost every single load instruction even for small values of `n`. To avoid this a technique known as transposing will be applied to matrix `B` before multiplying to increase memory locality. Transposing matrix `B` will create a new matrix $B^T$, in which all rows from `B` become columns and all columns from `B` becomes rows. For bigger values of `n` this transformation should not effect the run time by much due to transposing having a time complexity of $O(n^2)$, whereas the multiplication itself takes

`O(n³)`. However, it might be done in parallel by distributing the rows of `B` among the threads, each swapping only relevant elements from their own rows. The code in listing 4.2 illustrate how this is done in OpenMP. Again a new team of threads are started with the clause `#pragma omp parallel`, causing the next block `#pragma omp for private(j, temp)` to run in parallel. The `for` clause divides the workload of the next loop among the threads and due to `private(j, temp)` all threads get their own copy of the variables `j` and `temp` to avoid race-condition that might occur in these.

```
void transpose(double *matrix, int n){
  int i, j;
  double temp;

// Start a team of threads to transpose the matrix
#pragma omp parallel
#pragma omp for private(j, temp)
  for (i = 0; i < n; i++) {
    for (j = i + 1; j < n; j++) {
      temp = matrix[i * n + j];
      matrix[i * n + j] = matrix[j * n + i];
      matrix[j * n + i] = temp;
    }
  }
}
```

**Listing 4.2:** Snippet from `mm_transpose.c`

With matrix `B` transposed, the innermost loop of listing 4.1 on page 38 may now be expressed as `C[i * n + j] += A[i * n + k] * B[j * n + k];` On each iteration for `k`, values of matrix `B` are now accessed continuously instead of multiples of `n`. This should allow the compiler to insert prefecth instructions, thus avoiding many cache misses.

Let us compile and run the code again with dimensions of `n=1000, 5000, 10000, 15000`, as listed in the plot 4.2 on the next page. This is an immediate overall decrease in run times on both the Phi and the host Xeon. As `n` moves through the values `1000, 5000, 10000`, performance on the Phi improves by respectively 75%, 90% and 92% from the simple solution. The host Xeon has decreased run times as well, with respectively 14%, 65% and 64% for the first three values of `n` and 65% for `n=15000`. Two interesting observations arise with these results. First, the rate of how the running time improves seems to steady out with increasing values of `n`. One

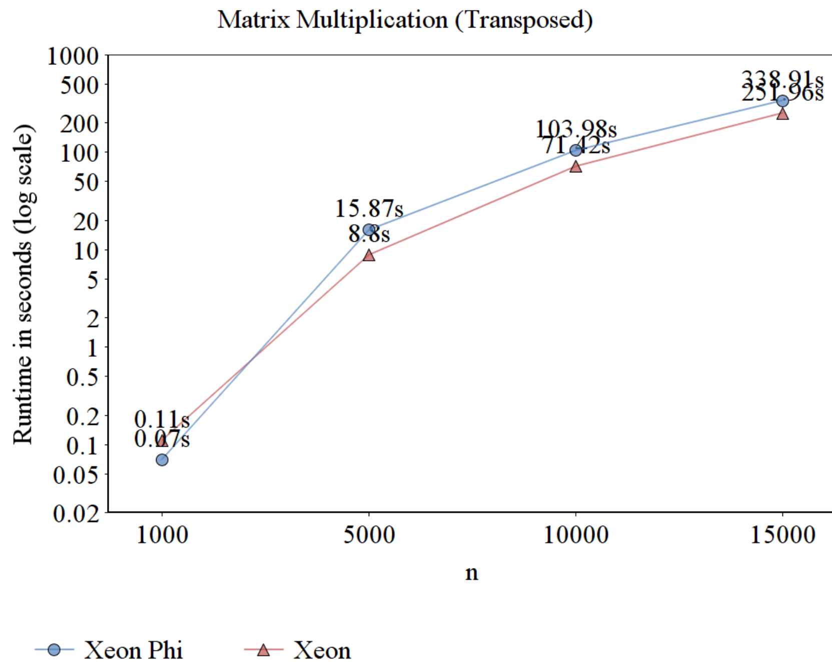


**Figure 4.2:** *Matrix Multiplication - Transpose Runtimes*

possible explanation is that somewhere between the two lowest values of `n`, the number of prefecth instructions reach its maximum due to a constraint bound by the cache sizes. The cache can only hold a certain amount of data, which will limit the number of prefetch instructions. Should prefetch instructions be issued to rapidly, cache lines could be evicted even before being used. This is not a desired behaviour. The second observation of interest is related to the run times for the two lowest values of `n`. For `n=1000` the Phi is faster than the host Xeon, but for `n=5000` the opposite is true. How could this be? Again the cache sizes and the total number of caches play an important role. From earlier (see section *2.1.3 on page 12*) we saw how the Phi-s L2 caches were twice the size of the host Xeon's. This allows more prefetch instructions on the Phi, given that the hardware can support it, which might have a bigger impact on the run time for smaller values of `n`.

### 4.3.2 Vectorization

There are still room for improvements. Compiling with the option `-opt-report=3` will generate a report over which sections of code were vectorized and how they were vectorized. The report will also include

information about which sections were not vectorized. To utilize the VPU-s should be essential for performance, both on the Phi and the host Xeon. In this particular case we must ensure that the innermost loop in function `multiply()` is being vectorized. That is, the multiplication of elements from matrix `A` and `B` are independent and can be done simultaneously in the VPU-s as described in subsection 2.1.5 on page 22.

However, looking at the report in the region describing function `multiply`, four lines of interest show up:

```
remark #15344:  loop was not vectorized:
  vector dependence prevents vectorization.
remark #15346:  vector dependence:
  assumed FLOW dependence between C line 24 and C line 24
```

In short, the compiler were not able to vectorize the multiplication line due to dependencies. There is one type of problem that arise with the function. Pointers to matrices are being passed as parameters, giving the compiler a hard time figuring out whether these pointers overlap in memory or not. In such cases the compiler will not vectorize unless it can guarantee correct result after vectorization, which in this case it could not. We can override the compiler in several different ways. For instance, the compiler directive `#pragma simd` force the compiler to vectorize the next loop no matter what, or the directive `#pragma ivdep` makes the compiler assume that no vector dependencies exists. In this case we will add the keyword `restrict` to the pointers being passed to function `multiply()`. `restrict` hints the compiler that the following variable do not overlap in memory with any other variables. The new function signature is given as: `void multiply(double *restrict A, double *restrict B, double *restrict C, int n)`.

```
$ icc -openmp -restrict mm.c mm_vec.c -o host_vec -opt-report=3
$ icc -mmic -openmp -restrict mm.c mm_vec.c -o mic_vec -opt-report=3
```

Compiling again with `-opt-report=3` one can see that the loop has been vectorized, with an estimated potential speedup of 15 (for the MIC, only 5.5 on the Xeon due to higher vector loop cost). Since `restrict` is a standard from C99, the flag `-restrict` or `-std=c99` must be included in the compilation as well:

```
LOOP BEGIN at mm_vec.c(25,13)
  remark #15300:  LOOP WAS VECTORIZED
  remark #15442:  entire loop may be executed in remainder
  remark #15448:  unmasked aligned unit stride loads:  2
  remark #15475:  -- begin vector loop cost summary --
  remark #15476:  scalar loop cost:  24
  remark #15477:  vector loop cost:  1.250
  remark #15478:  estimated potential speedup:  15.030
  remark #15488:  -- end vector loop cost summary --
LOOP END
```

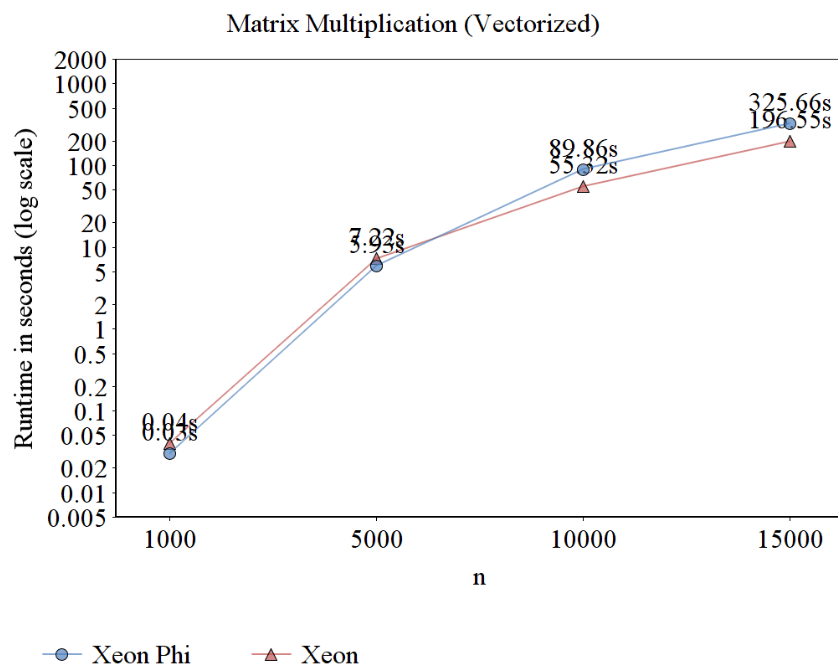


**Figure 4.3:** *Matrix Multiplication - Vectorization Runtimes*

The output of running the programs with vectorization ensured are listed in plot 4.3. Even though the run times benefit from vectorization, the numbers do not nearly match the estimated potential given by the vectorization report above. Compared to the results from the transposed version, run

times on the Phi have decreases by roughly 57%, 62%, 14% and 4% for respectively dimensions of `n=1000, 5000, 10000, 15000`. The host Xeon has improved overall run times too, with decreases of 63%, 16%, 22% and 22%. With vectorization enabled the Phi was able to perform better than the host Xeon for the two lowest values of `n`, with the same order of performance boost for these values. As the Phi both have bigger and many more VPU-s than the host Xeon this was as expected. However, when `n` became `10000` the speedup subsided and for `n` equal to `15000` there were barely any performance gain at all. For the last two values of `n` the host Xeon did not experience the same drop in performance as the Phi. As `n` increases another issue seems to become more dominant. To answer this the memory access pattern must be reviewed once more.

### 4.3.3 Blocking

Even though transposing matrix `B` rearranged its memory access pattern to a consecutive one, there is still one problem left untouched. The problem arise with increasing values of `n`. Both matrix `A` and `B` are being accessed from start to end one row or column at a time. For matrix `A` the same entry will be reused each time a new column in `B` is processed, which happens on every `n`'th element. The access of entries in matrix `B` have a much larger span since each column in `B` must be proceeded before the entry is being revisited, yielding a reuse on every $n^2$'th element. Potentially all values for a row in `A` could fit into cache for small values of `n`, which would allow reuse before cache lines holding these entries are evicted. For `B` this would be problematic even for small values of `n` since the whole matrix must fit into cache in order to be reused. The cache is a scarce resource, so a different approach to calculate the elements of matrix `C` could be more efficient.

Blocking, or *loop tiling*, is another way of structuring the loops to reuse data. By partition a loop's iterations space into smaller blocks it can ensure that data used within the loop stays in the cache until it is reused. The partition of the loop's iteration space leads to partitioning a larger array into smaller blocks, or in this case, partitioning the accesses of the matrices `A` and `B` into blocks of size `block_size`. Instead of computing only one entry $C_{i,j}$ at a time, one computes blocks of dimensions `block_size` $\times$ `block_size`. The updated version of function multiply is listed underneath. It is based on the blocking algorithm found on Intel's own matrix multiplication user guide. [11]

```
void multiply_block(double *restrict A, double *restrict B,
          double *restrict C, int n,
          int block_size) {
  int num_blocks = n / block_size;
  int i, j, k, ib, jb;
  double C_BLOCK[block_size][block_size];

// Startup a new thread team with divided work from
// collapsed loops 'i, j'. Each thread get their own
// 'k', 'ib', 'jb' and 'C_BLOCK' variables.
#pragma omp parallel
#pragma omp for collapse(2) private(k, ib, jb, C_BLOCK)
  for(i = 0; i < num_blocks; i++) {
    for(j = 0; j < num_blocks; j++) {

      // Reset C_BLOCK to zeros
      memset(C_BLOCK, 0, sizeof C_BLOCK);

      // Calculate C = A * B for the block
      // C_BLOCK[block_size][block_size]
      // starting at C[i * block_size][j * block_size]
      for(k = 0; k < n; k++) {
        for(ib = 0; ib < block_size; ib++) {
          for(jb = 0; jb < block_size; jb++) {
            C_BLOCK[ib][jb] +=
              A[(i * block_size + ib) * n + k]
            * B[(j * block_size + jb) * n + k];
          }
        }
      }

      // Update C with values from C_BLOCK
      for(ib = 0; ib < block_size; ib++) {
        for(jb = 0; jb < block_size; jb++) {
          C[(i * block_size + ib) * n
          +(j * block_size + jb)] = C_BLOCK[ib][jb];
        }
      }
    }
  }
}
```

**Listing 4.3:** Snippet from `mm_block.c`

The two outer loops `i` and `j` have been partitioned into blocks of size `block_size` and each thread will compute a `C_BLOCK` at a time, before placing the results into matrix `C`. Each `C_BLOCK` are computed block-wise,

by moving a block at a time along the rows of `A` and a block at a time along the columns (or rows in a transposed variant) of `B`. An important notice about the code is that the block size must be a divisor for the dimension parameter `n`. If not, the two outer loops will not span all rows and columns. The approach is better visualized through an example. In the following, matrices `A` and `B` are multiplied, with matrix `B` transposed. We choose `n=4`, `block_size=2` and `cache_size=4` with a LRU cache scheduling as implemented both on the Phi and host Xeon.

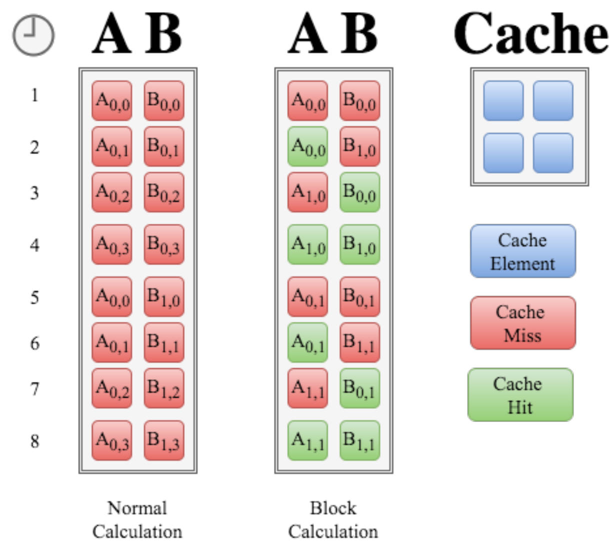


**Figure 4.4:** *Cache Reuse: Normal vs Blocking*

From figure 4.4 the simple algorithm is listed to the left and the blocked one to the right. Cache misses are marked with a red box and cache hits with a green box. The simple algorithm never gets the chance to reuse values, resulting in 100% cache miss, whereas the blocked one has a 50% cache hit ratio. The patterns are repetitive, so this percentage will be true even after both matrix product are fully computed. By choosing the right value of `block_size` one can maximize the reuse of cached entries, but finding this value theoretical is rather difficult. It will depend on the cache size, how much other data or information simultaneously stored there and the order of loop nests among others. As such, the values used in the run below are chosen empirically. Again we compile the new version and run the program with dimensions of `n=1000, 5000, 10000, 15000`. The plot is listed in 4.5 on the facing page

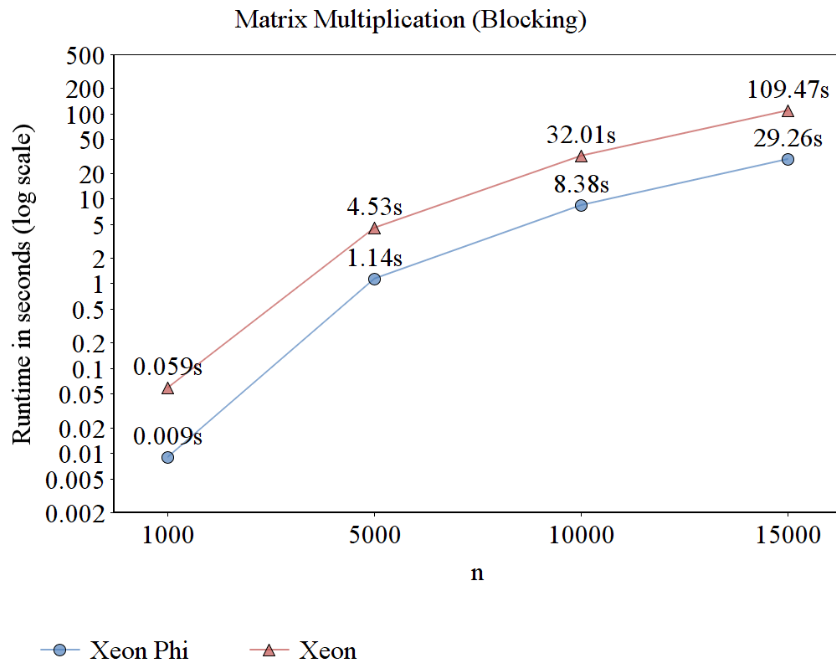


**Figure 4.5:** *Matrix Multiplication - Blocking Runtimes*

For the Phi a block size of 20 provided the best performance boost, where both lower and higher block sizes increased the run time proportionally. The host Xeon on the other hand had its performance peek at a block size of 40, but there were not that much variety from the range 20-80. One could also experiment with non square blocks, blocks with unequal number of rows and columns, but this is not covered here. In the results listed above a block size of 20 has been used on the Phi and a block size of 40 for the host Xeon. We see a clear distinction between this plot and the previous one. Not only have the performances on the Phi improved by 70%, 80%, 91% and 91% from the vectorized version, but the Phi outperformed the host Xeon on every values of `n`. It is interesting that the run times decrease more as `n` gets higher. This observation suggest that the problem arising in the previous optimization was indeed memory related.

### 4.3.4 Data Alignment

A final optimization technique that will be elaborated is alignment of data. Some vector instruction sets require to be loaded from an aligned memory address into the vector register. Failing to do so will result in extra latency. IMCI introduced with the Phi requires 64-byte data alignment, whereas

AVX found on the host Xeon has a 32-byte relaxed data alignment. An address `a` is aligned to a boundary `n` if the equation `a % n == 0` evaluates to true. Only arrays and non-scalar data structures are necessary to align and there are two ways to accomplish this.

Arrays declared as *static* or local arrays can be extended with the keywords `__attribute__((aligned(n)))` to let the first element be aligned on a `n`-byte boundary:

```
double C_BLOCK[block_size][block_size] __attribute__((aligned(64)));
```

For memory blocks allocated on the heap there are aligned versions of `malloc(size)` and `free(pointer)`, namely `_mm_malloc(size, n)` and `_mm_free(pointer)`:

```
double *A = _mm_malloc(n * n * sizeof(double), 64);
_mm_free(A);
```

Since the allocated arrays will be passed to a function `multiply(...)`, the compiler is not able to validate within the functions whether these parameters are aligned or not. It needs to be assisted with the compiler hint `__assume__aligned(pointer, n)`, which hints the compiler that `pointer` is aligned to `n`-byte boundary. The compiler need to verify that data accessed within the loop start at a multiple of some *value* too. This *value* depends on the type of the accessed data and the size of the VPU-s found on that architecture. In this problem the data type is `double`, with the host Xeon and the Phi fitting respectively four and eight of these types into their VPU-s. To inform the compiler the hint `__assume(condition)` may be used. It hints the compiler to assume the given condition to be true. One last time the source code is updated with the necessary changes:

```
void multiply_block(...) {
  ...
  // Align C_BLOCK to VEC_ALIGN-byte boundary
  double C_BLOCK[block_size][block_size]
      __attribute__((aligned(VEC_ALIGN)));

  // Let compiler know that matrixes are aligned
```

```
  // to VEC_ALIGN-byte boundary
  __assume_aligned(A, VEC_ALIGN);
  __assume_aligned(B, VEC_ALIGN);
  __assume_aligned(C, VEC_ALIGN);

  // Let compiler know that n is a multiple of how
  // many double fit in VPU
  __assume(n % (VEC_ALIGN / sizeof(double)) == 0);


#pragma omp parallel
#pragma omp for collapse(2) private(k, ib, jb, C_BLOCK)
  for(i = 0; i < num_blocks; i++) {
    for(j = 0; j < num_blocks; j++) {

      ...

      for(k = 0; k < n; k++) {
        for(ib = 0; ib < block_size; ib++) {
          for(jb = 0; jb < block_size; jb++) {
            C_BLOCK[ib][jb] +=
              A[(i * block_size + ib) * n + k]
            * B[(j * block_size + jb) * n + k];
          }
        }
      }

      ...

    }
  }
}
```

**Listing 4.4:** Snippet from `mm_align.c`

All three matrices `A`, `B` and `C` are assumed aligned to a `VEC_ALIGN`-byte boundary, which is 64-byte for the Phi and 32-byte for the Xeon. The `C_BLOCK` is extended with an attribute aligned sequence and the parameter `n` is assumed to be a value suited for the VPU-s. The variables `k`, `ib` and `jb` all start at 0 for all threads, so no assumption needs to be taken there. 0 is always a multiple for any value. For the variables `i`, `j` and `block_size` one should in theory help the compiler, but since both matrices `A` and `B` are accessed by a multiple of `n` in the innermost loop, it is sufficient providing information about `n`. From looking at the optimization report's output by the compiler when including the flag `opt-report=5` one can check that aligned accesses were generated:

```
$ cat mm_align.optrpt
...
$remark #15388:  vectorization support:
  reference A has aligned access [ mm_align.c(39,25) ]
$remark #15388:  vectorization support:
  reference B has aligned access [ mm_align.c(39,25) ]
...
```

Outputs from running with `n=1000, 5000, 10000, 15000` are shown in plot 4.6. On the Phi the benefits of aligned accesses are none, but the host Xeon has 30% overall decreased run times compared to previous results.

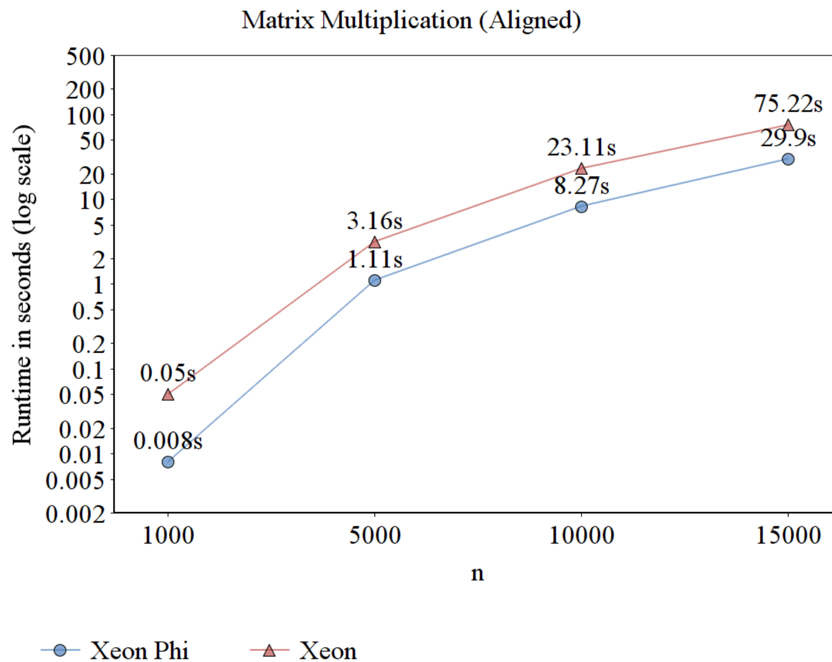


**Figure 4.6:** *Matrix Multiplication - Aligned Runtimes*

## 4.4 Discussion

Plotted in figure 4.7 on the facing page are all run times found in the previous subsections. There are four plots, one for each dimension `n=1000,`

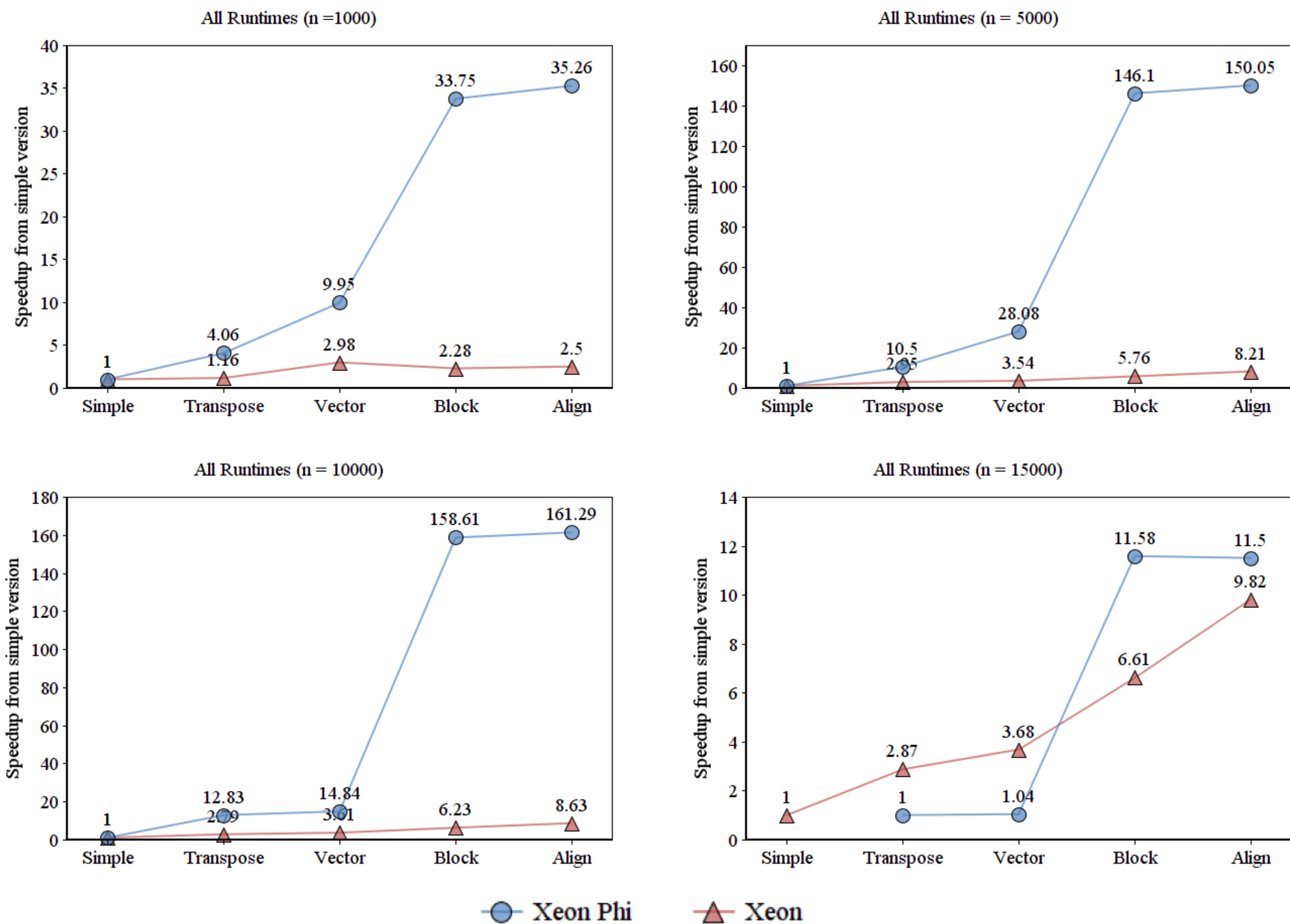


**Figure 4.7:** *Overview of run times for different sizes of n*

`5000, 10000, 15000` with optimization techniques along the x-axis and speedup (higher the better) from the simple version along the y-axis. Speedup is calculated as $\frac{\texttt{Simple Version Time}}{\texttt{Optimized Version Time}}$. Since the Phi did not complete its run for `n=15000`, speedup on this value is computed compared to the transposed version. As illustrated, matrix multiplication seems to be a type of problem suited for the coprocessor, at least with the right tunings. With memory being accessed in a smart manner, chunks of data can be kept within the big L2 caches on the coprocessor continuously feeding the VPU-s with vector instructions. The host Xeon benefits quite a bit from the optimization techniques too, but it is still outrun by the coprocessor with

roughly a factor of 3 for the biggest values of `n`. It is interesting as `n` gets higher, vectorization has less impact on the runtime. For values of `n=10000` and `n=15000` vectorization barely improves runtime on the Phi at all. For these values, memory access pattern seems to be the dominant problem. Big values of `n` causes too many cache misses, stalling cores on the Phi for many cycles as they look for the needed data in other cores' L2 caches or even further down the memory hierarchy as explained in *subsection 2.1.4 on page 20*. The host Xeon is not prone to the same extent as visualized in the plots. This might be because an extra memory layer, the L3 cache, resides on its microarchitecture, causing fewer cache misses to request data from RAM. In this example it becomes clear that transposing and blocking have the biggest impact on speedup, particularly when `n` increases. We see that utilizing the memory subsystem on the Phi is crucial for performance.

Although we did not try to invent a new better algorithm for matrix multiplication, it is interesting to compare run times for the final version with run times from a version using Intel's own commercial MKL library. Running the MKL library routine `cblas_dgemm(...)` took 6.48 and 19.65 seconds for respectively `n=10000` and `n=15000`. `cblas_dgemm(...)` is a function computing the matrix product of two matrices consisting of doubles. We see that the MKL version takes roughly 30% shorter time than our final version. The program code using the MKL library routine can be found in the matrix multiplication sections in the Appendix.

## 4.5 Summary

In this chapter a high computational, high memory usage problem was tested on both the Phi and the host Xeon. With the right tunings, matrix multiplication on the Phi outrun the host Xeon by an order of 3. The problem could be solved in a manner that reused cache and was highly vectorizable, taking full advantage of the many VPU-s and large L2 caches found on the coprocessor. We also note that transposing and blocking were the two most effective optimizing techniques.

# Chapter 5

# Quicksort

In the second problem used to compare the host and coprocessor the sorting algorithm *Quicksort* will be implemented. A sorting algorithm is by definition an algorithm taking some input and a sorting criteria, resulting in an output where elements are sorted by the criteria. The data type **int** will be used in this problem, sorting positive numbers from lowest to highest. Only relevant snippets of code are displayed throughout the sections and the full source codes can be found in the Appendix.

## 5.1 Problem Definition

Quicksort is a sorting algorithm following the divide-and-conquer design pattern [12]. The sorting problem is recursively divided into smaller sub-problems till some base case is reached, where the sub-problem can be solved directly. Each recursion consists of four steps:

1. **Check if base case is achieved**: If the base case is achieved then this branch is done. For the most basic quicksort the base case occur when only one or zero elements are to be sorted, because then there is nothing to sort. This is the *conquer* part of the design pattern.

2. **Find pivot element**: The next step is to find a *pivot* element, a value from the given sub-array that further will be used to divide the sub-array in two. Choosing a good pivot element may prevent the load balance to become askew when branching. This is not essential in the sequential solution, but when threads are introduced, dividing the workload into equal chunks is important.

3. **Partition sub-array**: The third step is to partition the sub-array based on the pivot element. All elements lower than the pivot element will be stored to the left in the sub-array and all elements greater than or equal to the right. The pivot element itself will be placed in between the two partitions.

4. **Branch out two new sub-problems**: Last, two new branches are created. One will be working on the sub-array to the left of the pivot element, the other on the sub-array to the right of the pivot element. This is the *divide* part of the design pattern. Both branches will perform point 1–4 on their sub-array.

Let us have a look at a simple example. An array of seven elements are to be sorted and for simplicity only low numbers are used. To choose the pivot element in each recursive step the value stored at the middle index in the sub-array is used. We mark unsorted elements in red, pivot elements in blue and sorted elements in green.

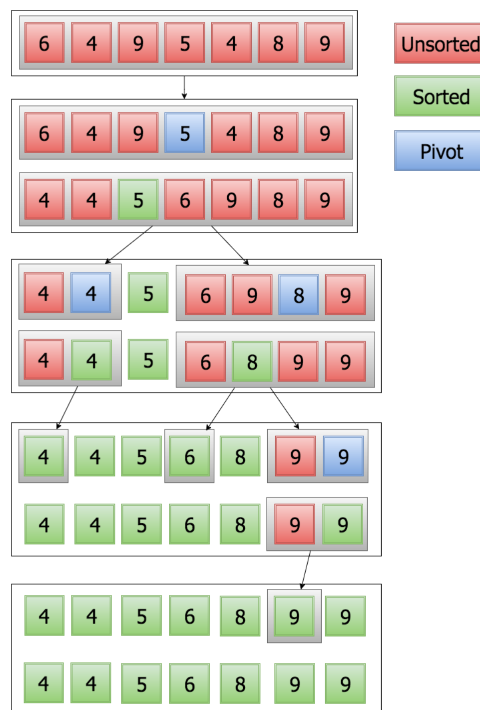


**Figure 5.1:** *Example of Quicksort*

Figure 5.1 shows the flow of the algorithm. The large white boxes are the recursive steps, the grey boxes the sub-arrays on each recursive step and the

arrows indicate where new branches are created. The first row in a white box highlights which elements were picked as the pivots on that level and the second row shows the partitioned sub-arrays. Branches created with no elements to be sorted are left out.

## 5.2 Naive Solution

We will begin with a modified, parallel version of the quicksort algorithm described above as our starting point. Because of overhead introduced with recursive calls, a direct sorting algorithm, like *Insertion sort [13]*, will be faster on very small sub-arrays. This threshold will be somewhere in the tens, but depends on the underlying architecture. The `THRESHOLD` used in our version is set to 20. Another modification is regarding partitioning of data. While partitioning there is a chance that duplicate values of the pivot element end up beside the pivot element itself. By operating with two pivot indexes, one for the lower bound and one on the upper bound of the duplicates, one might avoid unnecessary sorting. A function `partition(...)` with these adjustments is shown underneath.

```
void partition(int *a, int low, int high, int indexes[]) {
  // Pivot element set to value found at median index
  int middle_index = (low + high) / 2;
  int pivot = a[middle_index];
  int right, left;

  // Move pivot element to a[low], partition the rest
  swap(a, middle_index, low);

  // Place smaller values to left, bigger to the right
  for(right = left = low + 1; right <= high; right++) {
    if(a[right] < pivot) {
      swap(a, right, left++);
    }
  }

  // Move pivot element in between smaller and
  // bigger elements.
  swap(a, low, left - 1);

  // Skip all elements equal to the pivot element.
  right = left;
  while(right <= high && a[right] == pivot) right++;
```

```
  indexes[0] = left;
  indexes[1] = right;
}
```

**Listing 5.1:** *Partitioning from* `qs_simple.c`

Four parameters are passed to the function at each call. The sub-array to be sorted within array `a` is defined by the lower and upper bound parameters `low` and `high`. As there are no support for returning multiple values in C, an additional parameter `indexes` is needed to store the two pivot indexes. Partitioning is done by scanning `a` from `low` to `high`, swapping elements lower than the pivot to the left. The main recursive procedure calling `partition(...)` is listed below.

```
void qs(int *a, int low, int high, int depth) {
  // Number of elements below threshold? Insertion sort.
  if(high - low < THRESHOLD) {
    insertion_sort(a, low, high);
    return;
  }

  // Partition the array in the given boundaries.
  int indexes[2];
  partition(a, low, high, indexes);

  // Recursively call quicksort on left and
  // right side of pivot element.
  if(depth < max_depth) {
    #pragma omp parallel
    {
      #pragma omp single nowait
      {
        #pragma omp task
        qs(a, low, indexes[0] - 1, depth + 1);

        #pragma omp task
        qs(a, indexes[1], high, depth + 1);
      }
    }
  } else {
    qs(a, low, indexes[0] - 1, depth);
    qs(a, indexes[1], high, depth);
  }
}
```

**Listing 5.2:** *Recursive Quicksort from* `qs_simple.c`

If the size of the sub-array is lower than the defined `THRESHOLD`, the remaining elements in that branch will be sorted with insertion sort before the branch terminate. Else, the sub-array is partitioned with a call to `partition(..)` before branching into two new recursive calls. The parameter `depth` is checked against some allowed maximum depth, `max_depth`, to decide whether a sequential recursion or two new threads should be spawned. With two new threads spawned at every branching, the maximum number of threads running are $2^{max_depth}$. Spawning two threads in OpenMP with individual tasks may be achieved using the directive `#pragma omp task`. As the directive must be called within the parallel region `#pragma omp parallel`, it is necessary to control the number of threads creating tasks. `#pragma omp single nowait` ensures that only one thread from the team will execute the following structured block, that is, one thread creates the tasks and all other threads execute them as they become available. This source code and the following source codes in this chapter are compiled with the following commands:

```
$ icc -openmp -O3 qs.c qs_simple.c -o host_simple
$ icc -mmic -openmp -O3 qs.c qs_simple.c -o mic_simple
```

All keywords `icc`, `-openmp`, `-mmic`, `-O3` and `-o` have been covered in the previous chapter. Outputs from running the program with an unsorted array of sizes `n=1 million, 10 million, 100 million, 1 billion` are shown in figure 5.2 on the next page.

The plot illustrate three runs for each size of `n`. As in the previous chapter are values of `n` displayed along the x-axis and logarithmically scaled run times in seconds along the y-axis. There are quite a variety in the run times, with the host Xeon approximately four times faster on average than the Phi. Horizontal lines are added to better visualize the differences, with run time doubling at each line. It is worth noticing that the range of run times on the Phi is wider than on the host Xeon. One possible explanation is an askew workload balance between the threads caused by a poor choice of pivot element. An uneven distribution have bigger impact on the Phi because many more threads are running. Ideally the pivot element should divide

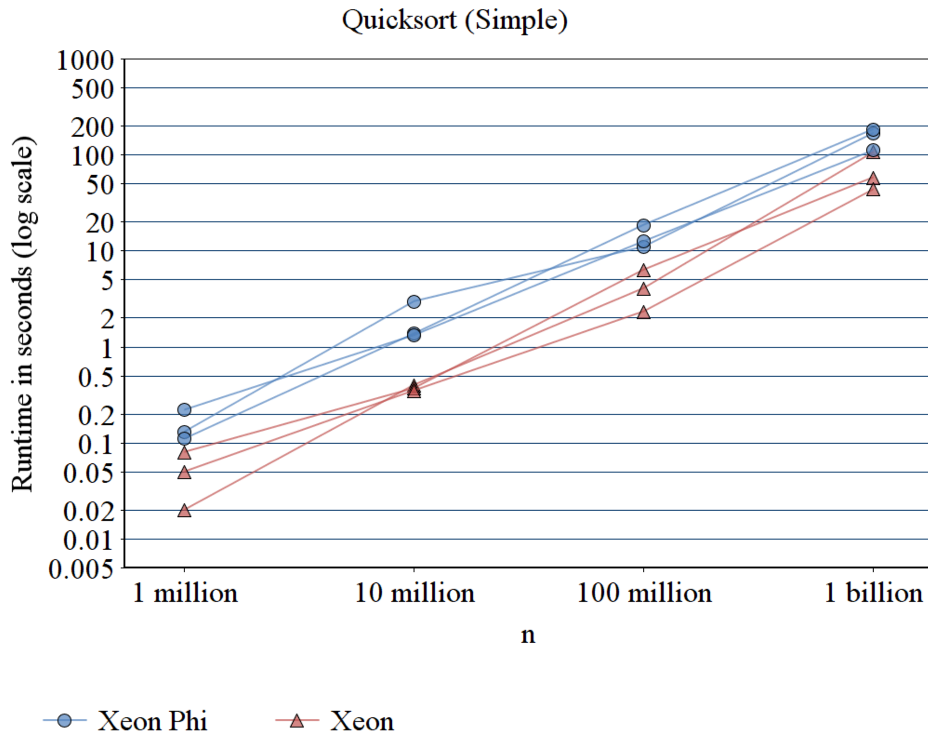


**Figure 5.2:** *Quicksort - Simple Runtimes*

the sub-array with equal amount of elements on the left and right side. With a total of 256 threads and an array of size $10^8$ perfectly partitioned at each branch, each thread would sort a sub-array of 390625 elements. However, let us assume the first pivot partitioned 90% of the elements to the left. This would cause half of the threads to sort 90% of the array, with the other half sorting the remaining 10% elements. Again in the next recursions bad pivot elements might be chosen, unbalancing the workload even further.

Another problem with quicksort is the large block of sequential code found in function `partition(...)`. This become a problem because the very first partition can only be performed by one thread and must scan the whole unsorted array of length `n` from start to stop, swapping elements along the way. Due to cores on the Phi operating at a much slower clock frequency than on the host Xeon (*see subsection 2.1.1 on page 7*), sequential code stalling all cores but one has major impact on the runtime. In addition, running only one thread per core will result in maximum 50% utilization of that core's potential as described in *subsection 2.1.2 on page 8*. Until at least 120 threads are running the cores are not fully utilized on the Phi.

## 5.3 Optimization

To improve run times, a few optimization techniques will be applied. Some of the methods introduced in the previous chapter, namely *vectorization*, *blocking* and *data alignment*, were rather general and might be used for the quicksort algorithm too. As it turned out this was not the case.

Swapping of elements are the only computational and memory changing operations performed in quicksort. From the vector report acquired when compiling with the option `-opt-report=3`, vector dependencies arise in the for-loop of function `partition(...)`. Different from matrix multiplication, these vector dependencies can not be removed. The problem is related to the algorithm itself and not compiler flaws. Firstly, a conditional branch reside within the loop body. Due to low complexity it should not be a problem for the mask registers found within the VPU-s on the Phi. However, by looking closely at the body of the `if` statement it become clear that iterations in the for loop are dependent on each other. If `swap(a, right, left++)` is executed in iteration `right`, the value of `left` are unknown till iteration `right-1` is executed. The loop body can not execute simultaneously for different values of `right`, resulting in no vectorization.

Blocking was introduced in matrix multiplication to improve reuse of cache lines and to increase data locality. The nature of multiplying two matrices allowed blocks of data to be processed at a time, fully utilizing the caches. Unfortunately, blocking can not be applied in quicksort. Since each partition rely on previous partitions, data within a single partition are only used once, thus eliminate the opportunity to use blocking.

The third option, data alignment, does not improve performance either. The unsorted array can be allocated using `_mm_malloc(n, boundary)`, providing an array of length `n`, starting at a `boundary`-byte alignment as described in the previous chapter. However, with only scalar operations performed the benefits are nowhere near that of vectorized operations. In the Appendix a version of quicksort with data alignment is provided. Running the program several times showed no performance boost whatsoever.

Two optimization methods will be applied to the code. They are choosing a *better pivot element* and a *full parallel quicksort*. The optimizations will be built on each other, meaning that previous techniques are included. For example, when applying full parallel quicksort, choosing a better pivot element is applied too.

### 5.3.1 Better Pivot Element

In the first approach we saw how the run times varied, probably because of workload balance becoming askew between the threads. As such, to establish a more reliable solution in which the workload are better divided, we will try to improve the choice of the pivot element. The simple solution chose the element found on the middle index in the sub-array, `int middle_index = (low + high) / 2`, which poorly represents the distribution of numbers. A perfect pivot element can only be obtained from an already sorted array, being the value at middle index. Choosing this element requires the main problem to be solved, so aiming for an element somewhere in the middle of the two extremes will be our goal.

How many elements must be sorted then, before we can choose the pivot element as the middle among them. Deciding this number becomes the main issue in this optimization. We will choose the number empirically as a theoretical study is rather complex. Testing with values of `5, 7, 9, 11, 13, 15, 17, 19` on an unsorted array of 100 million numbers gave the results in figure 5.3.

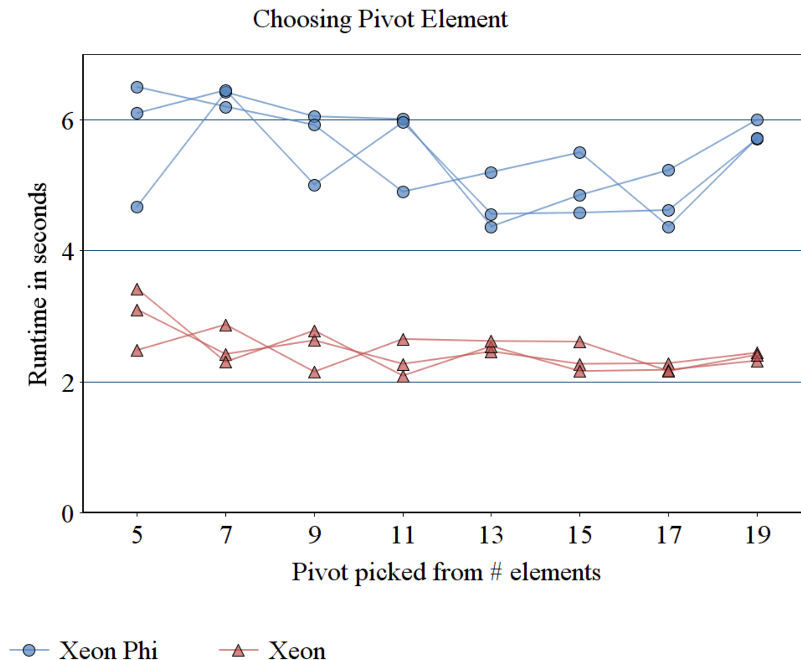


**Figure 5.3:** *Quicksort - Pivot picked from various numbers*

We see that the Phi performs best when picking the pivot element from 13, 15 or 17 sorted elements, whereas the host Xeon's performance does

not differ that much with the pivot element picked from 7 or more sorted elements. As expected, fewer threads running requires a less optimal pivot element. Plotted in figure 5.4 are the run times for the Phi and host Xeon, with pivot element picked from 13 sorted elements and number of elements `n=1 million, 10 million, 100 million, 1 billion`.

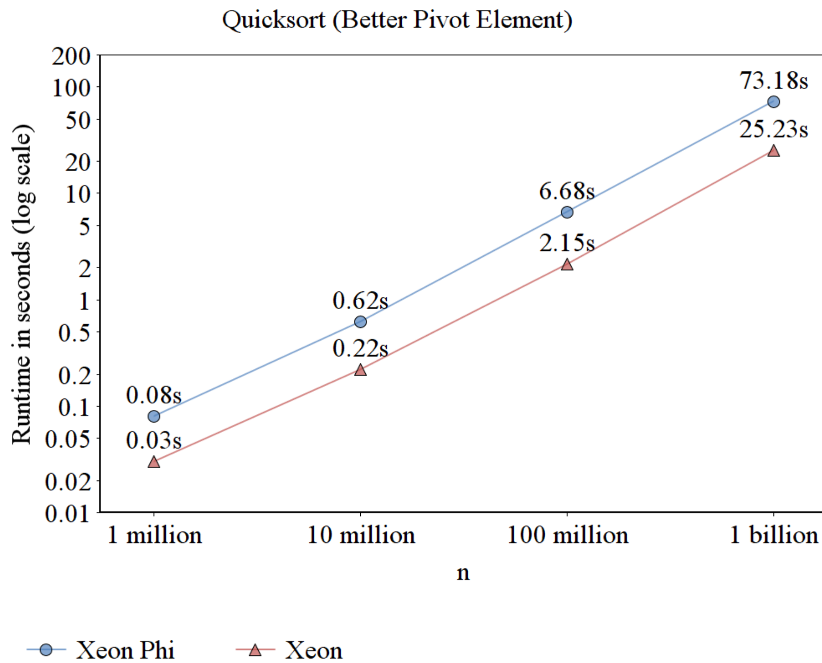


**Figure 5.4:** *Quicksort - Better Pivot Element Runtimes*

Differences in run times on the Phi have diminished for the lower values of `n`, with an overall decrease of 60% from the naive solution. Run times on the host Xeon have become more similar for low values of `n` too, with roughly 30% overall decrease. However, as only the median out of three runs are shown here, this can not be seen from the plot. For `n=1 billion` the run times on the Phi still vary somewhat, with values of `78.25`, `73.18` and `65.94` seconds. For the host Xeon all three runs ranged from `25` to `26` seconds. A possible explanation is related to an askew workload being dependent on both how the pivot element is picked and the total number of elements to sort. As the total number of element increase, the pivot element should be taken from a larger chunk of elements to better estimate the number distribution. Overall, choosing a better pivot element proved to effect the runtime as well as the stability of the algorithm.

### 5.3.2 Full Parallel Quicksort

There is still one issue left from the first approach, namely the large portions of sequential code as described earlier. To address a solution for this problem, one need to work around the quicksort algorithm from a different angle. Arne Maus [14] just did this in 2015. He developed a new version of quicksort, *Full Parallel Quicksort*. The new version has a different approach to partition the unsorted array. Instead of only a few threads doing all the work in the beginning, all threads now cooperate from the very first partition. All threads partition the whole unsorted array around the same pivot element. This is done by dividing the array among the threads and letting each thread partition its given chunk. When done, the array consists of many smaller segments with values lower than the pivot to the left and values greater than the pivot to the right. Next, each thread calculate the same common pivot index and the number of elements that thread must swap. After swapping, the algorithm has completed one original partitioning, with values smaller than the common pivot element to the left in the array and values greater than the common pivot element to the right. This approach adds additional overhead at the cost of being fully parallel, since half of the elements must be swapped twice; First local in a thread's given chunk, then global to the common pivot index. Finally, threads are divided in two halves, one half to partition the left side of the common pivot element and the other half to partition the right side of the common pivot element. This recursive process continues until an array segment only has one thread, then sequential quicksort is applied. A more detailed description is found in his paper. [15]

With the new approach, the source code become rather complex and big compared to former solutions. Multiple synchronization barriers are needed and many local temporary variables are computed by each thread at each recursion. As such, no program code will be listed here, but can be found in the Appendix under the Quicksort section. One important notice is that the number of threads must be equal to $2^{num}$ for some value `num`. This is related to how the code branches threads into two halves until only one thread remains in each branch. As such, the number of threads used on the Phi are explicitly increased to 256 instead of 240 with `export OMP_NUM_THREADS=256` before running. Let us compile and run quicksort one last time. Outputs from running with an unsorted array of sizes `n=1 million, 10 million, 100 million, 1 billion` are listed in figure 5.5 on the facing page.

The full parallel quicksort decreased run times on the Phi with another

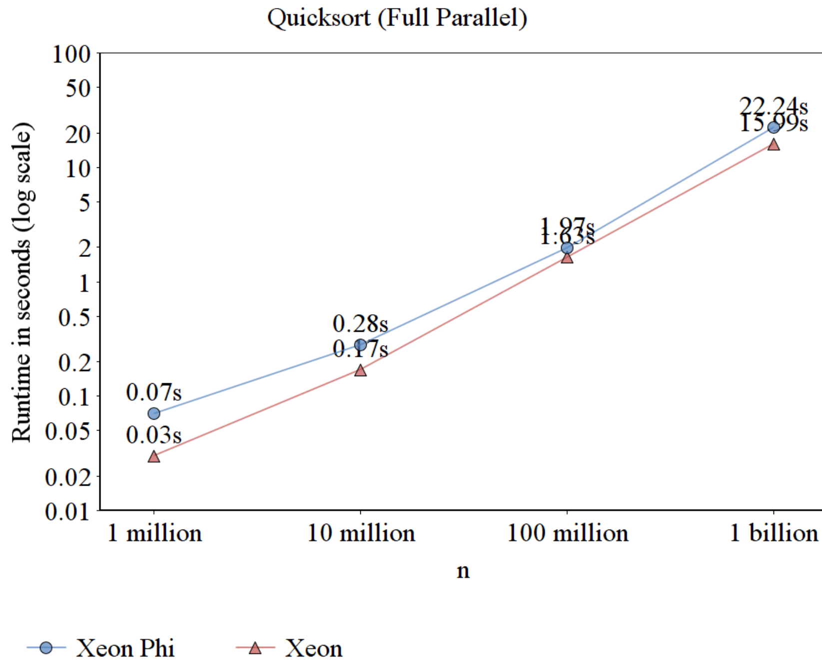


**Figure 5.5:** *Quicksort - Full Parallel Runtimes*

60–70% from the better pivot version. The host Xeon benefits too, with roughly 20–55% decrease in run times compared to the better pivot version. We see that a full parallel version tighten the gap between the two graphs, but it is still not enough for the Phi to become faster than the host Xeon. Performance on the Phi nearly tangent that on the host Xeon for `n=100 million`, but all other sizes of `n` are outperformed.

## 5.4 Discussion

Plotted in figure 5.6 on the next page are the median run times for all three versions of quicksort introduced in this chapter. Each plot represent one size of `n`. Along the x-axis we find the optimization technique used and along the y-axis the speedup (higher the better) from the simple solution. Again speedup is calculated as $\frac{\texttt{Simple Version Time}}{\texttt{Optimized Version Time}}$. Ensuring that all threads on all cores were fully occupied with a balanced workload proved to impact the run times for both the Phi and the host Xeon. It becomes clear that the Phi benefits more from both choosing a better pivot element and a fully parallel version, as discussed in the two previous subsections. One can not observe from these plots that the Phi benefits more from

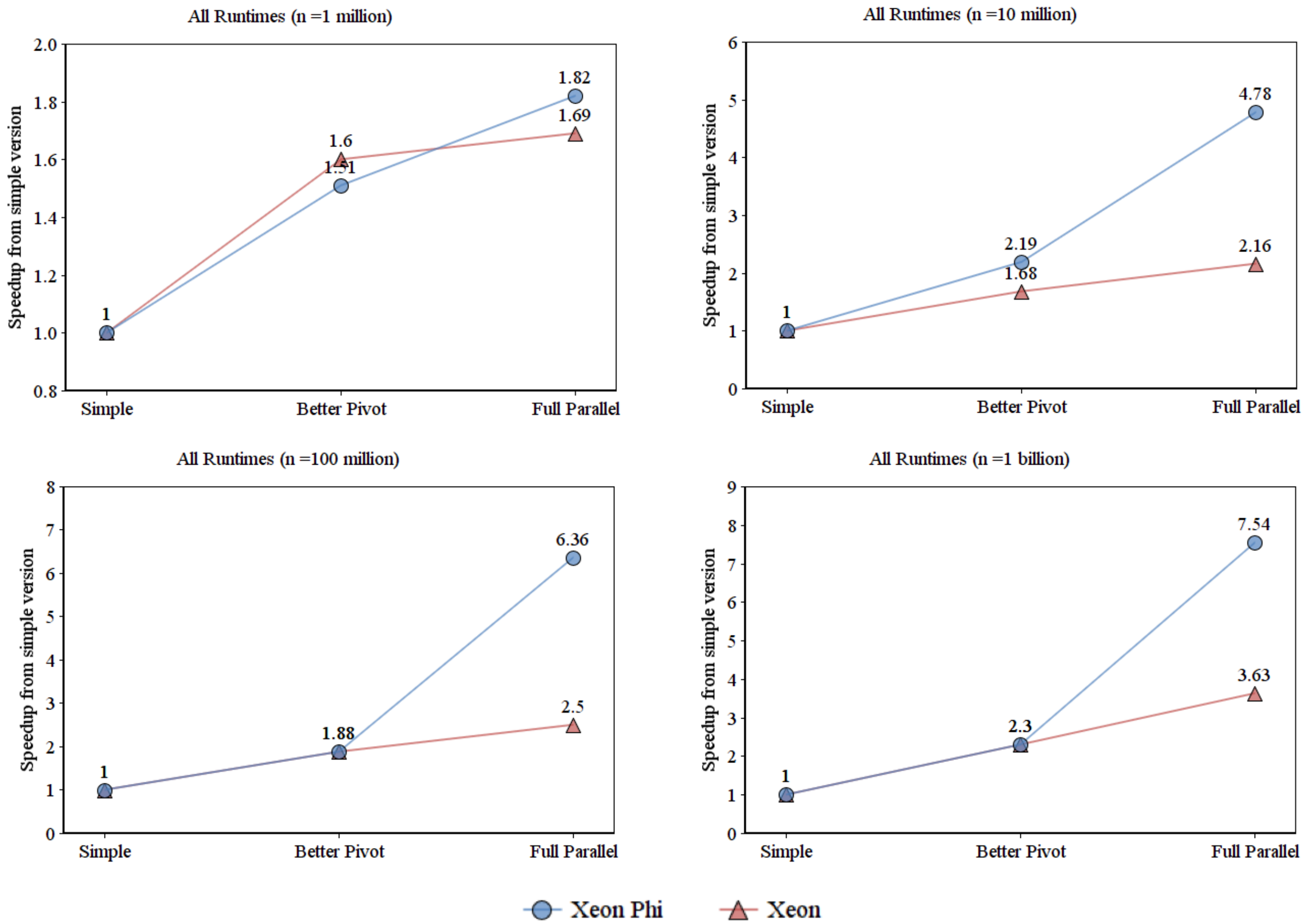


**Figure 5.6:** *Overview of run times for different sizes of $n$*

choosing a better pivot though. In fact it looks like the opposite is true, as the host Xeon have a steeper or equal incline when introducing the better pivot optimization. This is due to only the median out of three runs being shown. In reality run times varied a lot more on the Phi than the host Xeon before a better pivot element was chosen. The overall speedup for the Phi improved more than the overall speedup for the host Xeon, but the speedup taken from the median run time did not benefit more. However, we did not manage to outrun the host Xeon as with matrix multiplication. In fact, the host Xeon performs slightly better even for the final version. Memory locality and vectorization are two of the main reasons, as neither

could be adapted to the quicksort algorithm. An interesting observation is how the slopes of the graphs change as `n` increases. On the host Xeon the biggest incline is given when applying the *better pivot* optimization technique. This is true for all sizes of `n`. On the Phi the same is true for the lowest size, but as `n` increases the decline associated with the *full parallel* optimization technique become steeper. For the three biggest sizes of `n` the Phi benefits more from the *full parallel* optimization than the *better pivot* one. As mentioned earlier, this might be because a larger size increases the amount of time spent in sequential code, which is terrible for the Phi (see subsection 2.1.2 on page 8, under decoding of instructions).

Another observation of interest is if we try to overbook the number of available threads. For large arrays threads will probably spend much time waiting for data to arrive, which might allow other threads to utilize the CPU-s meanwhile. Trying to run the full parallel solution with double amount of threads and `n=1 billion` pushed run times on the Xeon down to 10 seconds. The Phi could not perform a run with 512 threads.

## 5.5 Summary

In this chapter a medium computational, medium memory consuming problem was tested on both the Phi and the host Xeon. With the right tunings, quicksort run on the Phi nearly tangent quicksort run on the host Xeon. We were not able to enable vectorization nor memory locality, which both proved to be important in the former chapter. On the other hand, the problem could be solved in a fully parallel manner without creating bottlenecks associated with sequential code that arise in ordinary quicksort.

# Chapter 6

# Traveling Salesman

The third and final problem that will be tested is the traveling salesman problem. As with matrix multiplication and quicksort is the traveling salesman problem well known. Belonging to the complexity class **Non-deterministic Polynomial Time Hard (NP Hard)**, that is, the class of all problems not yet solvable in polynomial time [16] and whose solution can not be verified in polynomial time either, the traveling salesman problems running time grows exponential with its input size. Again **int** will be used as the data type and only relevant snippets of code are displayed throughout the sections. Full source code can be found in the Appendix.

## 6.1 Problem Definition

**The Traveling Salesman Problem (TSP)** asks the following question: *Given a list of cities and the distances between each pair of cities, what is the shortest possible route through all cities that visits each city exactly once and end in the starting city?* Depending on whether the distance between two cities is the same in both direction, we get either a **symmetric TSP (sTSP)** or **assymetric TSP (aTSP)**. In this thesis only sTSP will be covered, meaning that the cost of going from city $A$ to city $B$ equals the cost of going from city $B$ to city $A$. All problems will also follow the triangle inequality, which states that the sum of any given two sides in a triangle must be greater than or equal to the remaining side. [17] TSP can further be visualized through a graph, with vertexes being cities and edges the cost of going between two cities. Let us have a look at a small example with only four cities.

The graph shown in figure 6.1 on the next page connects four vertexes,

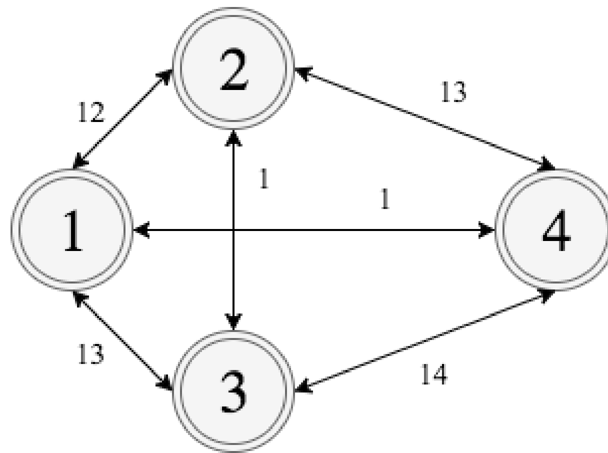


**Figure 6.1:** *Example of TSP*

or cities, labeled `1`, `2`, `3` and `4`. The edges represent the distances between the cities. What unit being used does not matter, only the relative values. In a fictive world where all cities connected directly to every other cities, we would get a complete graph where an edge existed between every pair of distinct vertexes. In reality this is not true, so TSP may result in a non-complete graph. In such cases one could assign the non-existing edges a very large value or use an algorithm that takes this into account. In our example there exist a total of six paths starting and ending in vertex `1`:

`(1)` $1 \longrightarrow 2 \longrightarrow 3 \longrightarrow 4 \longrightarrow 1$, `total cost: 28`
`(2)` $1 \longrightarrow 2 \longrightarrow 4 \longrightarrow 3 \longrightarrow 1$, `total cost: 52`
`(3)` $1 \longrightarrow 3 \longrightarrow 2 \longrightarrow 4 \longrightarrow 1$, `total cost: 28`
`(4)` $1 \longrightarrow 3 \longrightarrow 4 \longrightarrow 2 \longrightarrow 1$, `total cost: 52`
`(5)` $1 \longrightarrow 4 \longrightarrow 2 \longrightarrow 3 \longrightarrow 1$, `total cost: 28`
`(6)` $1 \longrightarrow 4 \longrightarrow 3 \longrightarrow 2 \longrightarrow 1$, `total cost: 28`

Because a symmetric graph is used, every path can be done both forward and backward with an equal total cost. Paths `[1, 6]`, `[2, 4]` and `[3, 5]` are essentially the same paths, only forward and backward. The paths with the least cost are `(1), (3), (5)` and `(6)`, which brings the total cost down to 28. In general there exists `(N-1)! / 2` different paths for sTSP with `n` cities.

## 6.2 Problem Data

Input data are obtained from **TSPLIB** [18] and thereby follow its file format. As stated on their website, this is a library of sample instances for the TSP problem from various sources and of various types. We will only be using a couple of the files, namely `att48.tsp` and `att532.tsp`. The two files define respectively the 48 capitals of the U.S and 532 locations within a city. As such, the first file has very varied distances whereas the distances in the second file are more or less of equal sizes. Our TSP solver are not able to find the optimal path among that many locations and only a subset of the locations in each file are used. Each file in the library consist of the following information:

- **NAME** : <the name of the file>
- **TYPE** : <input for what type of problem>
- **COMMENT** : <comment from author>
- **DIMENSION** : <for TSP, the number of locations>
- **EDGE_WEIGHT_TYPE** : <how to calculate distances>
- **NODE_COORD_SECTION** : <the coordinates for each location>

For the algorithms following the `DIMENSION` will be read from file, then all coordinates are read into two arrays `x` and `y` before the costs matrix `costs` is computed using the given `EDGE_WEIGHT_TYPE`. For the two files above `EDGE_WEIGHT_TYPE=ATT`, which computes coordinates in a semi-euclidean way. By using coordinates we get a complete graph and all elements of the costs matrix can be computed. Both files are added in the Appendix under the TSP section.

## 6.3 Naive Solution

The first approach would be the brute force algorithm generating all `(N-1)! / 2` permutations, then choosing the path with the lowest total cost. Such a solution yields a time complexity of `O(n!)`, which become infeasible to solve for values of `n` around 20 or higher ($20! = 2,43 \cdot 10^{18}$). Since we have no easy way of tracking which routes have been taken without storing

a massive amount of data, our algorithm is going to permute all `(N-1)!` different paths.

Unlike matrix multiplication and quicksort where task division among the threads were close to trivial, dividing the different paths among the threads in TSP were not that simple. The easiest approach would be to use `n - 1` threads for a problem of size `n`, each finding the optimal path for one of the cities going from the starting city. In a problem with four cities, three threads will then be used. The first thread will search among the paths $1 \longrightarrow 2 \longrightarrow * \longrightarrow 1$, the second among $1 \longrightarrow 3 \longrightarrow * \longrightarrow 1$ and the third among $1 \longrightarrow 4 \longrightarrow * \longrightarrow 1$. Such an approach does not scale at all when the number of available threads become more than the problem size `n`, as in the case on both the host Xeon and the Phi.

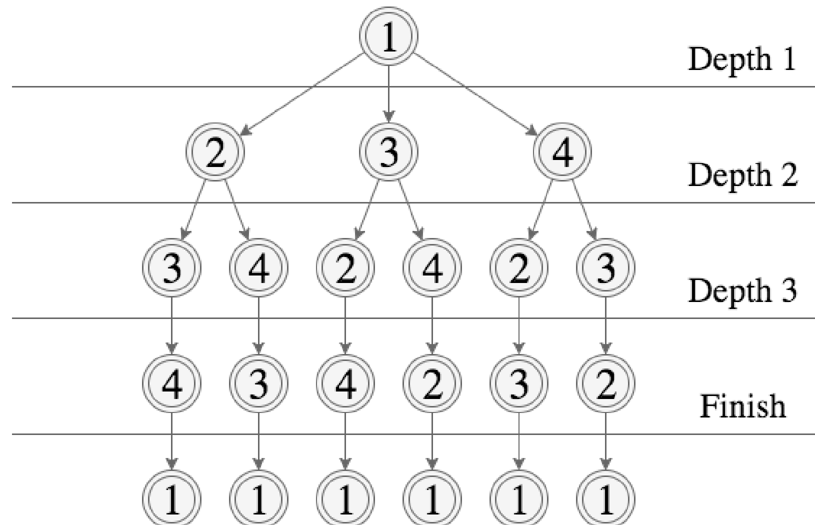


**Figure 6.2:** *Example of recursion tree*

How to distribute the work then? Assuming that we use a value of `n` such that `n! >= number of threads`, we can divide the work much clever. From the starting city there are `n - 1` different paths leading out. Going further down, there are `n - 2` outgoing paths from each of these cities. The recursive tree continues until all cities have been visited in each branch. If we use the example in figure 6.1 on page 70 we get the recursive tree shown in figure 6.2. At depth 1 there are `n - 1` tasks, or three in this example, to be computed. At depth 2 there are `(n - 1)` $\cdot$ `(n - 2)` number of tasks, or six here. In general, the number of tasks at any given depth `depth` is equal to $\frac{\texttt{(n - 1)!}}{\texttt{(n - 1 - depth)!}}$. The idea is to go down the recursion tree until we have at least as many tasks as number of threads. For each task

we will store what cities have been visited down that path, the current path chosen along with its length and which city this tasks origin from. In the source code an array of struct task, `task **tasks`, is computed with the necessary information about each task. The actual task creation is done by two large functions, `calculate_tasks(...)` and `create_tasks(...)`, which can be found in the Appendix under the TSP section.

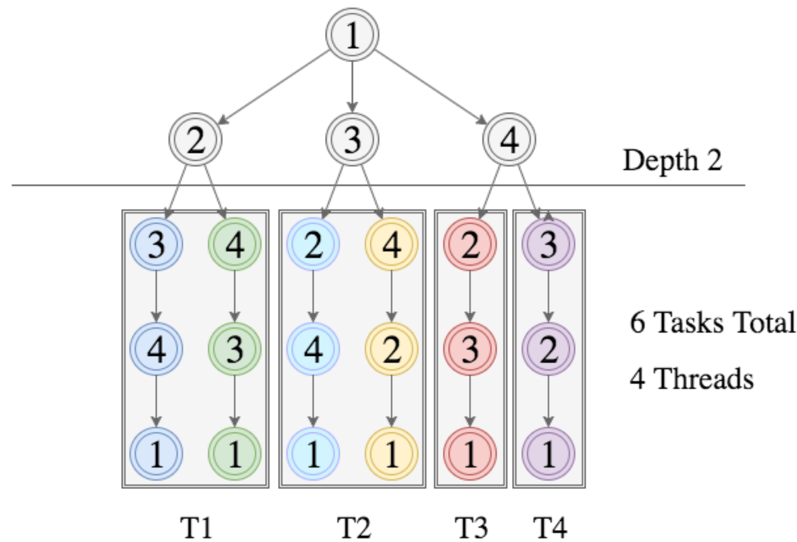


**Figure 6.3:** *Example of task division*

With all tasks collected in an array one can divide them among the threads. The fragment of OpenMP code listed below does just that. Again we use `#pragma omp parallel for` to start up a new team of threads, then split the workload among them in the following `for` loop. By adding `schedule(type [, chunk size])` we can define how the workload should be split among the threads. There are five different types; *static, dynamic, guided, auto* and *runtime*, which slightly vary from one another as explained in the OpenMP reference guide [19]. Testing showed no significant differences in run times, but letting the compiler/runtime system delegate the scheduling by choosing *auto* were overall slightly faster. Continuing using the example from figure 6.1 on page 70 a possible outcome of how tasks are divided among four threads are shown in figure 6.3. The first thread are assigned the two first task, the second thread the two next tasks, the third and fourth thread then get one task each.

```c
// Complete tasks
int i;
#pragma omp parallel for schedule(auto)
for(i = 0; i < num_tasks; i++) {
  calculate_optimal_tour(n, tasks[i], tasks[i]->from);
}
```

**Listing 6.1:** *Dividing tasks from `tsp_simple.c`*

With tasks divided each thread will try finding an optimal route. A call to the function `calculate_optimal_tour(int n, task *task, int from)` continues down the recursion tree of the given task trying to find a new optimal path. The function needs three parameters; the number of cities `n`, the task `task` to be performed and the previous city `from`. Every time a new shorter path is found, the global best path and best length are updated. Since all threads have access to these variables, it is necessary to update within a lock to avoid race conditions. The function is listed below.

```c
void calculate_optimal_tour(int n, task *task, int from) {
  int i;

  if(task->current_level == n) {
    // Found a route through all cities,
    // if it's better note that in best_so_far path.
    omp_set_lock(&lock);

    if(task->current_len + costs[from][0] < best_len_so_far)
    {
      best_len_so_far = task->current_len + costs[from][0];
      memcpy(best_way_so_far, task->current_way,
          sizeof(int) * (n + 1));
    }

    omp_unset_lock(&lock);

  } else {
    // Recursively visit each remaining cities.
    for(i = 0; i < n; i++) {

      int dis = costs[from][i];

      if(dis != 0
        && !task->visited[i]
        && task->current_len + dis < best_len_so_far) {
```

```
        task->current_len += dis;
        task->current_way[task->current_level] = i;
        task->visited[i] = 1;
        task->current_level += 1;

        calculate_optimal_tour(n, task, i);

        // Backtrack
        task->current_len -= dis;
        task->visited[i] = 0;
        task->current_level -= 1;
      }
    }
  }
}
```

**Listing 6.2:** *Calculate Optimal Tour from `tsp_simple.c`*

A simple cutoff is added to the function as well. Should the current path at any point become longer than the existing shortest path there is no point to continue down that branch. This condition is captured by the expression `task->current_len + dis < best_len_so_far` in the `if` branch of the outer `else` statement. This source code and the following source codes in this chapter are compiled with the commands:

```
$ icc -openmp -O3 tsp.c tsp_simple.c -o host_simple
$ icc -mmic -openmp -O3 tsp.c tsp_simple.c -o mic_simple
```

All keywords `icc`, `-openmp`, `-mmic`, `-O3` and `-o` have been covered in previous chapters. Outputs from running the program with number of locations equal to `n=15, 16, 17, 18` on both scattered and compact locations are listed in figure 6.4 on the following page.

The first thing to notice is the differences in run times for `att48.tsp` and `att532.tsp`. The compact city problem takes much longer time, with runs on the Phi and host Xeon taking roughly 17 and 5-12 times longer than the U.S capitals problem set. This is due to less variation in distances between city locations, causing fewer branches in the recursion tree to be cut off. Beside the differences in run times we also notice that the Phi is outperformed for all runs but two.

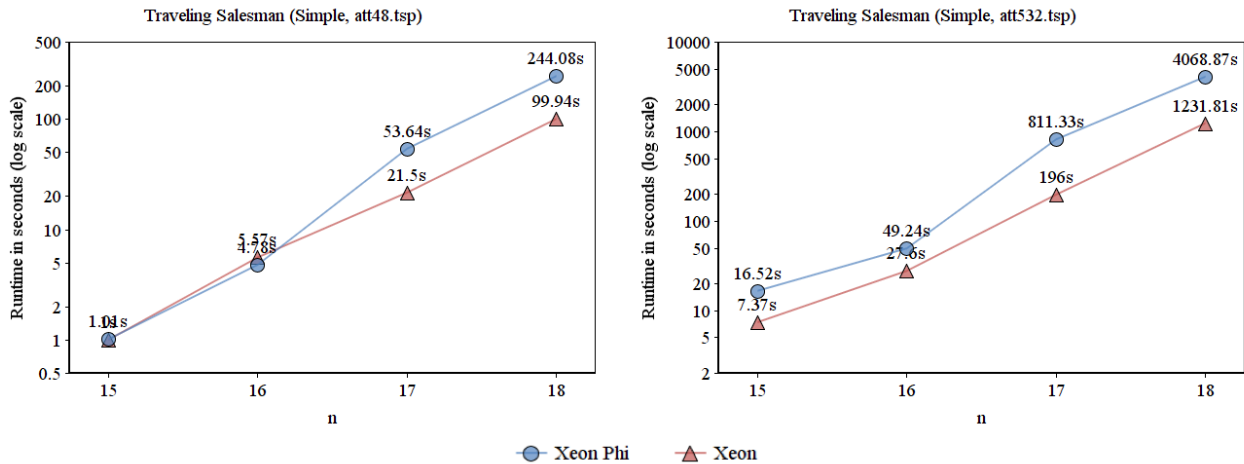


**Figure 6.4:** *TSP - Simple Runtimes*

## 6.4 Optimization

To improve run times, a few optimization techniques will be applied. As in the two previous chapters we could try to adapt the algorithm for some of the general optimization techniques, namely *vectorization*, *blocking* and *data alignment*. As with Quicksort it turned this could not be done.

Except calculating the costs table, the only computations done are recursive calls and accumulating of distances as the algorithm goes down the different branches in the recursion tree. Neither of these requires a lot of arithmetic operations and the vector processing units can not be utilized.

Blocking was introduced in matrix multiplication to improve reuse of cache lines and to increase data locality. The nature of multiplying two matrices allowed blocks of data to be processed at a time, fully utilizing the caches. Unfortunately, blocking can not be applied in TSP. TSP stores very little information at a time, just enough to keep going down the branches in the recursion tree. Alongside the costs table of size $n \cdot n$ this includes the current branch's length, its current path, what locations it has visited and what location was previously visited. A quick estimate assuming four threads per core are being used, the costs table is shared and each int takes four bytes gives:

$$\texttt{Usage Per Core (in bytes):}\quad n^2 \cdot 4 + 4 + n \cdot 4 + n \cdot n + n$$

For instance, 20 locations equals a total usage of only 1768 bytes, which is roughly 1,8KB. Each L1 cache on the Phi are able to store 32KB of instructions and 32KB of data, so even with additional meta data and program data memory should not be an issue. With all necessary data residing within the cache at once there is no point to apply blocking. The third option, data alignment, is of no use neither. With neither vectorization or heavy use of memory present, data alignment has little purpose.

The starting algorithm already fully utilize all available threads. As such only two optimization methods will be applied to the code, both related to directly enhance the algorithm itself. They are choosing a *Better Starting Length* and choosing a *Better Cutoff*. The optimizations will be built on each other, meaning that when a better cutoff is applied, choosing a better starting length is applied too.

### 6.4.1 Better Starting Length

Looking into the code the optimal length starts at `INT_MAX`, which is a very high number. Since our cutoff technique depends on the length of the shortest path, choosing a better starting length might improve performance. What length should be chosen? We will simply use a greedy algorithm [20] to compute the shortest path from the starting location through all other locations, then in the end add on the distance from the end location to the start location. This can be computed in $O(n^2)$ as choosing the next city requires iterations over all non-visited cities so far. Along with the length for the route the actual path is saved too. To do so the function `find_greedy_tour(int n, int *best_way_so_far)` listed below will be used. Other than what is mentioned above the function will save the shortest path found in the parameter `best_way_so_far` provided by caller.

```
int find_greedy_tour(int n, int *best_way_so_far) {
  int walks, from, best_to, len, i, best_len_so_far;
  int visited[n];

  len = INT_MAX;
  walks = from = best_to = best_len_so_far = 0;

  // Reset visited and best path so far
  memset(visited, 0, sizeof(int) * n);
  memset(best_way_so_far, 0, sizeof(int) * (n + 1));

  while(++walks < n) {
```

```
    // Find shortest path from 'from' to an unvisited 'i'.
    // Always choose 'i' with lowest value to 'from'.
    best_to = 0;
    len = INT_MAX;

    for(i = 1; i < n; i++) {
      if(i != from && !visited[i] && costs[from][i] < len) {
        best_to = i;
        len = costs[from][best_to];
      }
    }

    visited[best_to] = 1;
    best_way_so_far[walks] = best_to;
    best_len_so_far += len;
    from = best_to;
  }

  // Add on cost from end to start
  best_to = 0;
  best_way_so_far[n] = 0;
  best_len_so_far += costs[from][0];

  return best_len_so_far;
}
```

**Listing 6.3:** *Choose better starting path from* `tsp_greedy.c`

Compiling and running the program for both `att48.tsp` and `att532.tsp` with number of locations `n=15, 16, 17, 18` gave the results found in figure 6.5 on the next page. Improving the starting value seemed to have no significant impact on the total run time for neither compact nor scattered locations when the total number of cities `n` is small. A possible explanation might be because `n` is relatively low compared to the number of threads running. With many threads running down different branches in the recursion tree at once, a new optimal path will be found almost immediately if `n` is small enough. If only a few threads were running they might not find a good path that quickly, which would require more branches to be visited in the start. In such a run one could expect the greedy version to be somewhat faster. However, this would vary from problem set to problem set as finding a good path early on also rely on which branches are being visited in the start. In a problem set where the simple version encountered a short path early on, the run times would probably not differ. However, for the two biggest values of `n` when using the problem set `att532.tsp` one clearly see that run times on the Phi have improved. With roughly 15%

decreased run time for both values of `n` we see how a good starting length may improve run times.

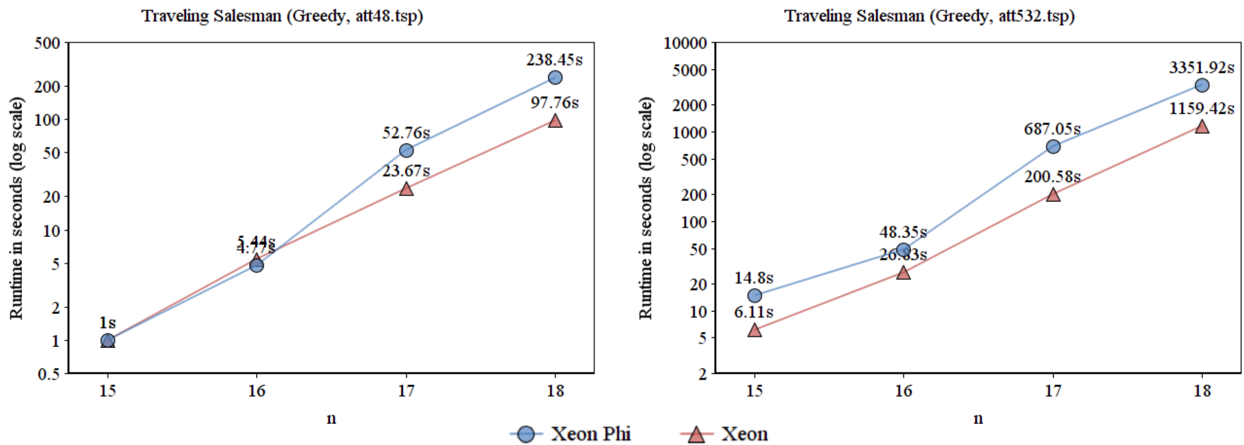


**Figure 6.5:** *TSP - Better Starting Length Runtimes*

### 6.4.2 Better Cutoff

A second improvement to the algorithm would be lower the number of computed branches in the recursion tree. This requires a better estimate for when a branch might be cut off. Till now a branch have been cut off whenever it grew longer than the already shortest path. How can one choose a better cutoff and guarantee that the shortest possible path will be found? One idea is to create a cutoff vector as large as the number of cities `n`. Each entry at index `i` will store the sum of the `i` shortest distances found in the costs matrix. This creates a vector, `cut_offs`, where the lowest distance is found at index zero, the sum of the two lowest distances at index one and so on. We get a vector containing the mathematical minimum distances sets. When going down a branch in the recursion tree one can then check for cutoff by using this vector. Given the length of the shortest path found so far, `best_len_so_far`, a current path with `j` cities left to visit and its current length `current_len`, a cutoff can be made whenever `current_len + cut_offs[j] >= best_len_so_far`. The equation guarantee correctness since `cut_offs[j]` contains the theoretical minimum sum of `j` distances in the costs table, no matter which cities are being used. We know that the sum of a current paths remaining distances can not be shorter than this value, allowing us to terminate the current branch. Creating the cutoff vector requires $O(n^2 \cdot \log n)$ time complexity

and should have no impact on the run time. Each cost in the costs matrix must be visited and whenever a cost shorter than the largest of the shortest costs is found, it is insertion sorted down the cutoff vector. Findings of such costs decreases with the number of checked distances, hence the log `n`. Let us again compile and run the code for both files `att48.tsp` and `att532.tsp` for number of cities `n=15, 16, 17, 18`. The outputs are plotted in figure 6.6.

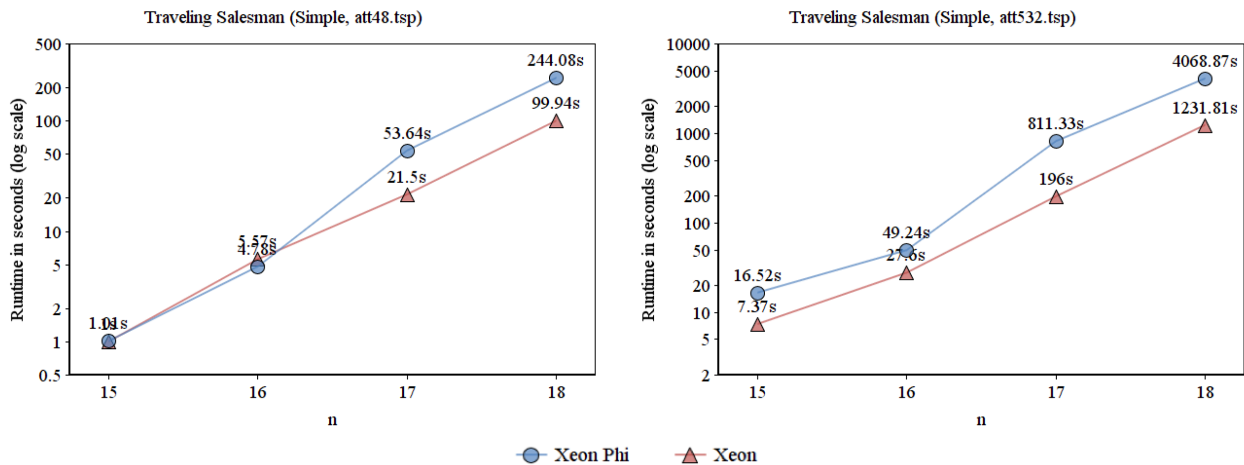


**Figure 6.6:** *TSP - Better Cutoff Runtimes*

Run times have improved significantly on both the Phi and host Xeon for both types of problem sets. The only exception is when the number of cities `n=15` in the `att532.tsp` problem set, where run time has increased for the host Xeon. Trying to run this specific situation several more times gave a wide range of run times, `8.97s, 7.44s, 3.50s, 10.99s, 5.35s, 3.67s, 9.86s, 3.35s`. This behaviour is probably a result of to few locations `n` in combination with the problem set being compact. Sometimes a very good path may be found early on by one of the threads, causing many cutoffs. Other times a less optimal path is found, which does not allow the same number of interceptions. Since the problem size is small finding the better of the two become crucial. A second observation is that run times on the U.S states problem set have decreased more than the city problem set. Since distances vary a lot in the U.S states problem it follows that a good cutoff will impact this problem set more than a problem set with many equal distances.

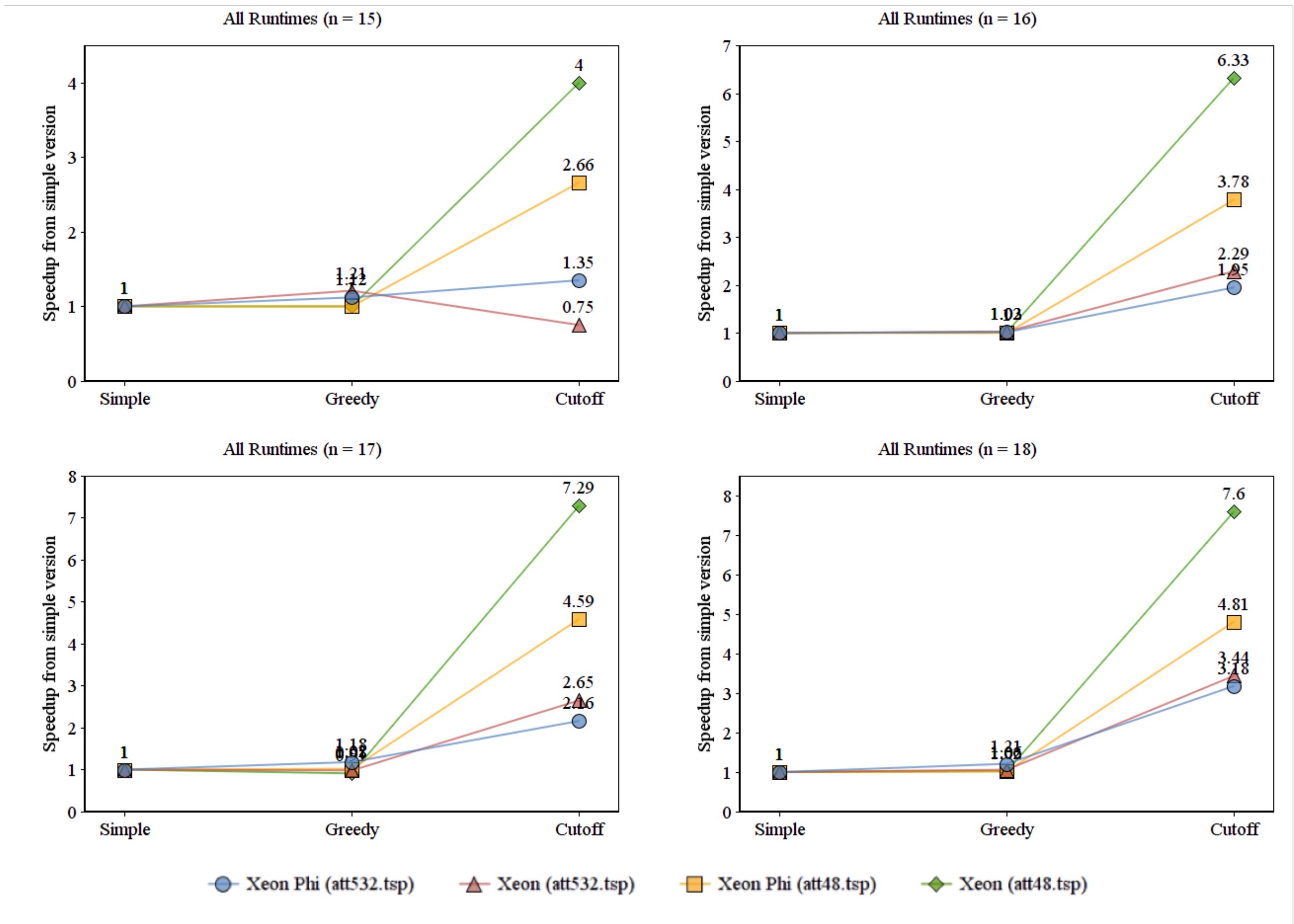


**Figure 6.7:** *Overview of run times for different number of cities* $n$

## 6.5 Discussion

Plotted in figure 6.7 are the median run times for all three versions of TSP introduced in this chapter. Each plot represent one value for `n`. Along the x-axis we find the optimization technique used and along the y-axis the speedup (higher the better) from the simple solution. Again speedup is calculated as $\frac{\texttt{Simple Version Time}}{\texttt{Optimized Version Time}}$. Speedup for both problem sets `att48.tsp` and `att532.tsp` are included in each plot. Even though we managed to utilize all threads available on both architectures the Phi was not able to outperform the host Xeon. In fact the Phi was outrun by a factor

ranging from roughly 1.5–4 for the last version, depending on the value of `n`. We also note that the host Xeon benefits the most from the optimization techniques, with the biggest inclines as `n` increases. A possible theory is related to the ALU-s. Since there are no vectorization and very few cache misses, each thread will keep feeding arithmetic operations to the ALU-s in the core it reside in. As described in subsection 2.1.2 on page 8 there are two ALU-s per core on the Phi and the execution of instructions follow an in-order execution model. The host Xeon can compute three arithmetic operations at once per physical core, but it schedules the operations in an out-of-order execution model. With many instructions queuing the ALU-s on the Phi threads might halt, waiting for the results. This may indeed happen on the host Xeon as well, but because the core operate at a higher clock frequency and schedules the instructions in a smart manner, the waiting time per instruction may decrease. This is however hard to verify as decreasing the number of threads will impact what global optimal paths are being found. Fewer threads will most likely create fewer tasks, which in turn changes how the global optimal path changes. Running the U.S capitals problem set for `n=18` with half the number of threads on both architectures gave more or less the same results whatsoever. This support the idea that threads are kept waiting for results from the ALU-s, but it seemed to have equal impact on both the Phi and the host Xeon.

It is also important to note that speedup depends on the problem set itself. Using a compact city location problem caused fewer optimal paths to be found, resulting in less speedup than the scattered U.S capitals problem set. The number of threads running may also directly impact the run time a non parallel manner. More threads allow more tasks to be checked at once, which can impact the run time significant. For instance, running `att48.tsp` for `n=18` on the host Xeon with 17 and 18 threads took respectively `13.15` and `55.88` seconds. A problem of size 18 only creates 17 task when 17 threads are running, but when 18 threads are running the total number of tasks increases to $17 \cdot 16 = 272$ ensuring that each thread gets at least one task.

## 6.6 Summary

In this chapter a high computational, low memory usage problem was tested on both the Phi and the host Xeon. The problem could be solved in a fully parallel manner, but lack of vectorization as the only real computation were recursive calls and simple arithmetic operations. Implementing a better cutoff enhanced the run times on both architectures, but the coprocessor

was still outperformed by the host for both a compact and a scattered problem set. Creating enough tasks were of great importance for the run time too, as a new global optimal path could be found faster.

# Chapter 7

# Conclusion & Further Work

In the previous chapters we tested three different types of problems:

- **Matrix Multiplication**: A high memory consuming, high computational problem. As the dimensions of the matrices scale, the needed memory and arithmetic operations do too. Even for relatively small dimensions storing all three matrices in a Phi-s L2 cache become infeasible.
- **Quicksort**: A medium memory consuming, medium computational problem. Memory usage only scale linear in this problem as opposite to the matrix multiplication of order squared polynomial. A fair lot of arithmetic operations are done, but they only included comparison of numbers and swapping values of memory locations.
- **Traveling Salesman**: A low memory consuming, high computational problem. In this problem we saw that very little memory was needed and we could possible fit the entire algorithm within the L1 cache of both versions of the Xeon. The problem was also computing intense, with accumulation of lengths being the main arithmetic operation.

## 7.1 Results

Benchmarking a simple parallel solution was the starting point for all three problems. Then we tried to adapt two classes of optimization techniques, the general ones which were *memory access pattern*, *memory alignment* and *vectorization*, and methods that were rather algorithm specific. As each

problem started with a parallel solution we already utilized all threads on both versions of Xeon. We did not try to invent new better algorithms for solving the problems, but simply used their basic templates then introduced some optimizations. We then ran each new optimized version and compared run times to the host Xeon on four different input sizes. Listed below are the progress to the final optimized versions, run with the biggest input parameter in their sets. For more detailed results, review the specific problem's chapter.

### 7.1.1 Matrix Multiplication

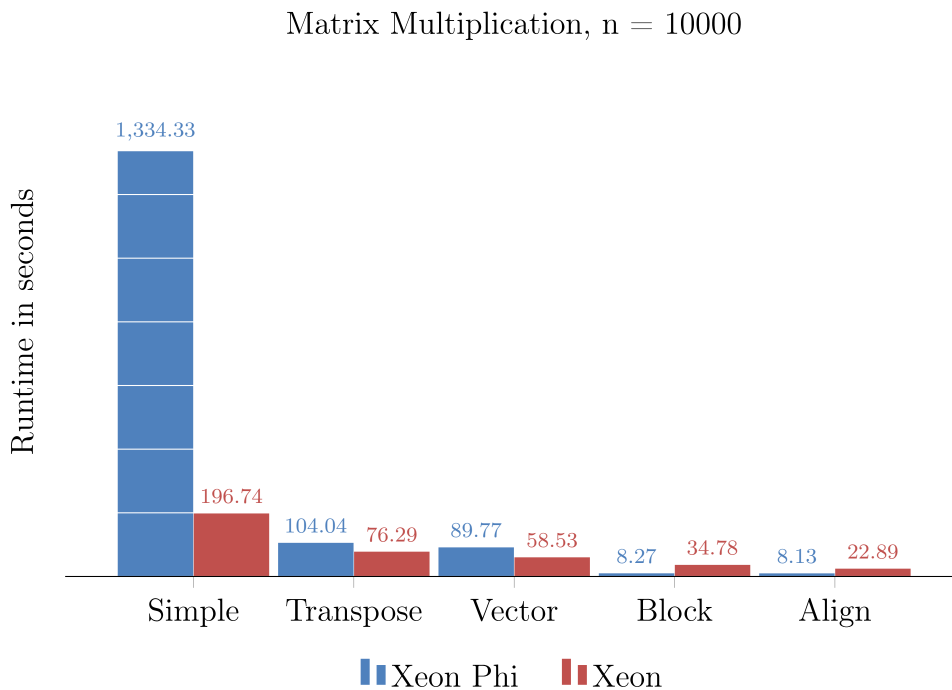


**Figure 7.1:** *Matrix Multiplication Progress, n = 10000*

As seen from the plot 7.1 the Phi outperformed the host Xeon for this problem. Since the Phi did not finish its run for `n = 15000` the second highest value is plotted. This was a kind of problem where we were able to utilize memory locality, take full advantage of the vector processing units and ensuring all threads were working at all times. What is interesting from these results are how small vectorization and memory alignment impact the run times. It might be that a different arrangement of the optimization techniques would yield other results, but looking at this specific case

we clearly see that the improvements in memory usage have the biggest influence.

### 7.1.2 Quicksort

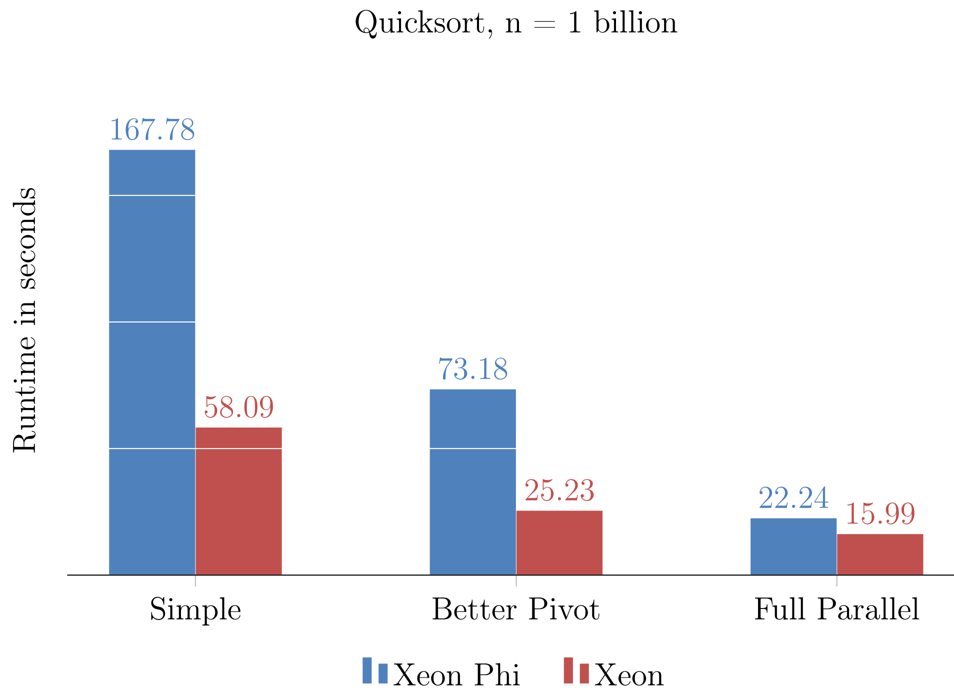


**Figure 7.2:** *Quicksort Progress, n = 1 billion*

For this problem, the Phi was outperformed by the host Xeon as seen from the plot 7.2. For Quicksort we were not able to apply vectorization nor memory locality. Since the only arithmetic operations performed were comparison and swapping of values, which were dependent on previous results, they could not be processed in the vector processing units. How memory was being accessed were unpredictable, which removed the possibility to block the algorithm, and left us only with opportunities to improve the algorithm itself. Improving the choice of pivot element had somewhat equal impact on both versions of Xeon, while a full parallel solution improved run times on the Phi the most.

### 7.1.3 Traveling Salesman

As seen from the plot 7.3 on the next page did the Phi not outperform the host Xeon for this problem either. As with Quicksort we were not able

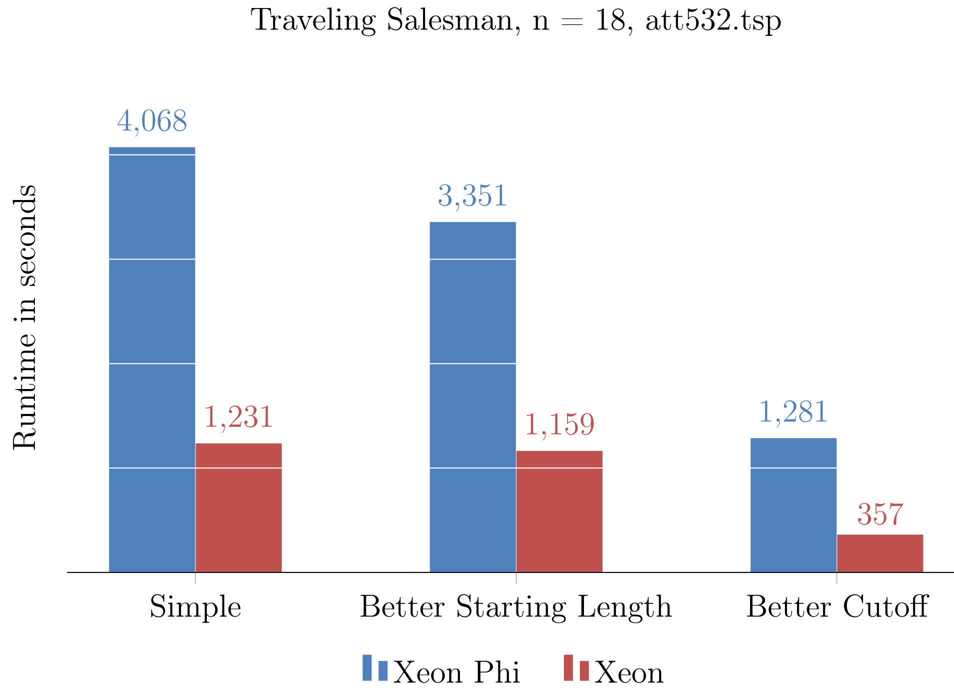


**Figure 7.3:** *Traveling Salesman Progress, n = 18, att532.tsp*

to apply vectorization nor memory locality on TSP. Again only dependent scalar operations were performed removing the possibility of using vector operations. Memory locality could not be adapted either as all necessary data already resided in L1 cache. With improvements only to the algorithm itself the decrease in run times for both versions of Xeon are of the same order, which is expected. It should be noted that this problem was not the best for comparing the two architectures, as run times also were dependent on the problem set, the number of threads/tasks created and what optimal paths were being found early on.

## 7.2 Conclusion

From the results we have obtained and their discussions, it looks like the Phi is suited for very specific types of problems. In order to fully utilize its potential we need a well scaling problem that is computing intensive, but without regard to the order of operations, and whose memory access pattern may be structured to take advantage of memory locality. Algorithms for such problems can exploit all cores, their vector processing units and keep necessary data within the L1/L2 caches avoiding many cache misses. As

it turned out, matrix multiplication meets these requirements. In cases where the requirements are not fulfilled, as in quicksort and TSP, using a regular CPU is enough. For these two examples the regular Xeon even outperformed the Phi. With only one out of three problems running faster on the Phi after tunings (zero before), it is fair to conclude that the Phi did not quite reach it's expectations.

## 7.3 Further Work

Although only a few problems were tested in this thesis, it pointed us somewhat in the direction for what the Phi is capable of. However, being an empirical study the best way to collect more data and results would be to further develop, test and compare other problems on the Phi and the host Xeon.

# Appendix A

# Problems Source Codes

This appendix includes compilation and links for the three problems *Matrix Multiplication*, *Quicksort* and *Traveling Salesman Problem* introduced in this thesis along with links to run times for some test runs.

## A.1 Abel computer cluster

All files have been tested on The Abel computer cluster [21], which requires log in with username and password. Once logged in, use the commands found below to access the proper computing nodes.

- Xeon (host) : `ssh c19-[17-20]`
- Xeon Phi (coprocessor) : `ssh c19-[17-20]-[0-1]`

In order to compile with `icc` and run OpenMP programs Intel's own library must be loaded. This is achieved by the command

```
module load intel/2016.0
```

when logged in on the host Xeon. All compilation must be done from the host Xeon. Further, before running programs on the Phi its environment must be defined. Both `LD_LIBRARY_PATH` and `PATH` must be constructed with the right libraries. For this thesis the following exports should cover the necessary libraries:

- `exportLD_LIBRARY_PATH=/cluster/software/VERSIONS/intelmpi.intel-5.0.2/mic/lib:/cluster/software/VERSIONS/intel-2016.0/compilers_and_libraries_2016.0.109/linux/compiler/lib/intel64_lin_mic:/cluster/software/VERSIONS/intel-2016.0/compilers_and_libraries_2016.0.109/linux/mkl/lib/intel64_mic:/cluster/software/VERSIONS/intel-2016.0/compilers_and_libraries_2016.0.109/linux/ipp/lib/intel64_mic`
- `exportPATH=$PATH:/cluster/software/VERSIONS/intelmpi.intel-5.0.2//mic/bin:/cluster/software/VERSIONS/intel-2016.0/compilers_and_libraries_2016.0.109/linux/bin/intel64_mic:/cluster/software/VERSIONS/intel-2016.0/compilers_and_libraries_2016.0.109/linux/mpi/intel64_mic/bin:/cluster/software/VERSIONS/intel-2016.0/bin:/cluster/software/VERSIONS/intel-2016.0/bin/intel64_mic:/cluster/software/VERSIONS/intel-2016.0/mkl/tools`

Once the module is loaded on the host Xeon and the programming environment setup on the Phi, we are ready to compile and run the programs. Compiling is defined in the next sections and running the program is trivial, simply use `>$ ./<program> <input>`.

## A.2 Matrix Multiplication Links

All Matrix Multiplication related code can be found at http://folk.uio.no/joakikr/master/matrix/. A version using Intel's built-in MKL library is also included. Compiling:

- **Simple** (*mm_simple.c*)
  - Xeon Phi : `icc -mmic -openmp mm.c mm_simple.c -o mic_simple.out`
  - Xeon : `icc -openmp mm.c mm_simple.c -o host_simple.out`
- **Transposing** (*mm_transpose.c*)
  - Xeon Phi : `icc -mmic -openmp -O3 mm_transpose.c -o mic_transpose.out`
  - Xeon : `icc -openmp -O3 mm.c mm_transpose.c -o host_transpose.out`
- **Vectorization** (*mm_vec.c*)

  - Xeon Phi : `icc -mmic -openmp -O3 -restrict mm.c mm_vec.c -o mic_vec.out`
  - Xeon : `icc -openmp -O3 -restrict mm.c mm_vec.c -o host_vec.out`

- **Blocking** (*mm_ block.c*)
  - Xeon Phi : `icc -mmic -openmp -O3 -restrict mm.c mm_block.c -o mic_block.out`
  - Xeon : `icc -openmp -O3 -restrict mm.c mm_block.c -o host_block.out`

- **Aligned** (*mm_ align.c*)
  - Xeon Phi : `icc -mmic -openmp -O3 -restrict mm.c mm_align.c -o mic_align.out`
  - Xeon : `icc -openmp -O3 -restrict mm.c mm_align.c -o host_align.out`

## A.3 Quicksort Links

All Quicksort related code can be found at `http://folk.uio.no/joakikr/master/quicksort/`. Compiling:

- **Simple** (*qs_ simple.c*)
  - Xeon Phi : `icc -mmic -openmp -O3 qs.c qs_simple.c -o mic_simple.out`
  - Xeon : `icc -openmp -O3 qs.c qs_simple.c -o host_simple.out`

- **Better Pivot Element** (*qs_ pivot.c*)
  - Xeon Phi : `icc -mmic -openmp -O3 qs.c qs_pivot.c -o mic_pivot.out`
  - Xeon : `icc -openmp -O3 qs.c qs_pivot.c -o host_pivot.out`

- **Full Parallel** (*qs_ fp.c*)
  - Xeon Phi : `icc -mmic -openmp -O3 qs.c qs_fp.c -o mic_fp.out`
  - Xeon : `icc -openmp -O3 qs.c qs_fp.c -o host_fp.out`

- **Aligned** (*qs_ align.c*)

- Xeon Phi : `icc -mmic -openmp -O3 qs.c qs_align.c -o mic_align.out`
- Xeon : `icc -openmp -O3 qs.c qs_align.c -o host_align.out`

## A.4 Traveling Salesman Problem Links

All TSP related code can be found at `http://folk.uio.no/joakikr/master/tsp/`. Compiling:

- **Simple** (*tsp_simple.c*)
  - Xeon Phi : `icc -mmic -openmp -O3 tsp.c tsp_simple.c -o mic_simple.out`
  - Xeon : `icc -openmp -O3 tsp.c tsp_simple.c -o host_simple.out`
- **Better Starting Length** (*tsp_greedy.c*)
  - Xeon Phi : `icc -mmic -openmp -O3 tsp.c tsp_greedy.c -o mic_greedy.out`
  - Xeon : `icc -openmp -O3 tsp.c tsp_greedy.c -o host_greedy.out`
- **Better Cutoff** (*tsp_cutoff.c*)
  - Xeon Phi : `icc -mmic -openmp -O3 tsp.c tsp_cutoff.c -o mic_cutoff.out`
  - Xeon : `icc -openmp -O3 tsp.c tsp_cutoff.c -o host_cutoff.out`

# Bibliography


[1] Wikipedia. Triangle inequality. https://en.wikipedia.org/wiki/Triangle_inequality. [Online; accessed 11-Apr-2016].

[2] Juan M. Cebrián, Lasse Natvig, and Jan Christian Meyer. Performance and energy impact of parallelization and vectorization techniques in modern microprocessors. *Computing*, 96(12):1179–1193, 2013.

[3] IntelPR. Intel reveals details for future high-performance computing system building blocks as momentum builds for intel xeon phi product. http://tinyurl.com/intel-knh-announced, November 2014. [Online; accessed 20-Jan-2016].

[4] Intel. Intel xeon phi coprocessor system software developers guide. *Manual*, page 15, March 2014.

[5] Micron Technology. Gddr5. https://www.micron.com/en/Products/DRAM/GDDR5. [Online; accessed 04-Feb-2016].

[6] Jianbin Fang, Ana Lucia Varbanescu, Henk J. Sips, Lilun Zhang, Yonggang Che, and Chuanfu Xu. An empirical study of intel xeon phi. *CoRR*, abs/1310.5842, 2013.

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

[8] Intel. Intel xeon phi coprocessor system software developers guide. *Manual*, page 37, March 2014.

[9] Mikhail Brinskiy and Mark Lubin. An introduction to mpi-3 shared memory programming. https://software.intel.com/sites/default/files/managed/eb/54/An_Introduction_to_MPI-3.pdf. [Online; accessed 27-Apr-2016].

[10] Wikipedia. Matrix multiplication. https://en.wikipedia.org/wiki/Matrix_multiplication. [Online; accessed 05-Mar-2016].

[11] Intel. General matrix multiply sample user's guide. https://software.intel.com/sites/products/vcsource/files/GEMM.pdf. [Online; accessed 05-Mar-2016].

[12] Wikipedia. Divide and conquer algorithms. https://en.wikipedia.org/wiki/Divide_and_conquer_algorithms. [Online; accessed 15-Mar-2016].

[13] Wikipedia. Insertion sort. https://en.wikipedia.org/wiki/Insertion_sort. [Online; accessed 19-Mar-2016].

[14] Arne Maus. Arne maus. http://www.mn.uio.no/ifi/personer/vit/arnem/index.html. [Online; accessed 22-Mar-2016].

[15] Arne Maus. Full parallel quicksort. http://heim.ifi.uio.no/~arnem/sorting/ParaQuick2015/FullParallelQuicksortNIK2015.pdf. [Online; accessed 22-Mar-2016].

[16] Wikipedia. Np (complexity). https://en.wikipedia.org/wiki/NP_(complexity). [Online; accessed 11-Apr-2016].

[17] Wikipedia. Moore's law. https://en.wikipedia.org/wiki/Moore%27s_law. [Online; accessed 23-Apr-2016].

[18] Gerhard Reinelt. Tsplib. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/index.html. [Online; accessed 15-Apr-2016].

[19] OpenMP ARB. Openmp 4.5 api c/c++ syntax reference guide. http://www.openmp.org/mp-documents/OpenMP-4.5-1115-CPP-web.pdf. [Online; accessed 15-Apr-2016].

[20] Wikipedia. Greedy algorithm. https://en.wikipedia.org/wiki/Greedy_algorithm. [Online; accessed 16-Apr-2016].

[21] UiO. The abel computer cluster. http://www.uio.no/english/services/it/research/hpc/abel/. [Online; accessed 19-Apr-2016].