Institut für Informatik

Lehrstuhl für Rechnertechnik und Rechnerorganisation / Parallelrechnerarchitektur

Techniques for adapting Industrial Simulation Software for Power Devices and Networks to Multi- and Many-Core Architectures

Dipl.-Math. Univ. Thomas Müller

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Martin Bichler
Prüfer der Dissertation:	1. Univ.-Prof. Dr. Arndt Bode
	2. Univ.-Prof. Dr. Hans-Joachim Bungartz
	3. Prof. Dr. Carsten Trinitis, University of Bedfordshire, United Kingdom

Die Dissertation wurde am 22. Januar 2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 9. März 2014 angenommen.

Abstract

Simulation software has been widely used in academic and industrial environments for a long time. In recent years, however, the available hardware characteristics have changed significantly and rapidly. Several years ago, true parallel processing was only available using clusters or expensive workstations, whereas today systems with multiple processor cores are well established and even mass market laptops provide multiple cores.

Contrary to performance increase due to rising processor clock frequency in the past, existing software typically does not automatically benefit from these additional cores and several other improvements. Instead, it has to be adapted to properly benefit from the increased compute power and to efficiently use today's multi- and many-core architectures.

This thesis devises techniques to cost-effectively conduct these adaptations to increase the efficiency of industrial high voltage engineering applications on multi- and many-core architectures and help to leverage their full potential. This is done using several real world exemplary simulation software packages provided by industry partner ABB.

The presented techniques include strategic changes to data structures and algorithms to improve time complexity by subtly inserting caches or extending data access methods where appropriate. Other changes improve performance by accounting for characteristic properties of processor hardware, such as caches or branch prediction – ultimately by dynamically generating optimized and highly problem specific code.

Additionally, the implications of reimplementing an application based on new and improved theoretical methods regarding performance as well as cost-effective development are discussed and evaluated.

Finally, various application characteristics and runtime parameters affecting parallel efficiency are illustrated on both general purpose multi-core processors as well as a Xeon Phi system representing a many-core architecture.

Acknowledgment

Many people have contributed to the success of this work. First of all, I would like to thank Prof. Dr. Arndt Bode for giving me the opportunity to do my doctorate at his chair, the Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR), for his continuous support and the freedom he granted me for my research. I am also grateful to Prof. Dr. Carsten Trinitis, who supervised my work, provided valuable feedback and directions. Furthermore, I am thankful to my colleagues for many fruitful discussions, leading to new ideas and perspectives, as well as an enjoyable work environment.

I am also grateful for the cooperation with ABB, which allowed me to conduct my research on real life industry applications and environments. In this context, I specifically want to thank Dr. Andreas Blaszczyk for his input and feedback, as well as for managing and coordinating the cooperation.

Finally, I would like to thank my family and friends for their continuous support over the years.

Contents

Chapter 1

Introduction

1.1 Motivation

For some years now parallel processing using an ever growing number of processor cores is a significant trend and the resulting multi- or even many-core processors represent a significant shift in processor development and architecture.

Most of the time industrial environments pay high attention to cost effectiveness, even regarding research and development. Because of this existing software is often only changed when necessary, such as to implement new functionality or to fix bugs. Until a few years ago this was mostly fine even for computationally demanding applications such as simulation software.

While compute clusters and expensive workstations in principle made parallel processing possible, performance improvement was still mostly realized by increasing the base clock frequency of the processor. This had the advantage that existing software automatically got faster, mostly without the necessity to adapt or restructure the application itself. So if more performance was needed, i.e. more complex problems needed to be solved or the computation of an existing problem had to be faster, it was typically sufficient to buy a newer processor.

This is no longer true though. Nowadays improvements of the serial computational performance of processors are no longer as significant. Instead, research and development concentrates on improving overall performance by enabling and encouraging increasingly parallel processing. However, existing serial applications no longer automatically get faster due to these advancements, but have to be modified to take advantage from the increased compute power.

From now on time and effort need to be invested into adapting and restructuring software to properly benefit from the increased compute power provided by modern multi- and many-core processors and further development in the future. A phrase was coined by Herb Sutter [77] in 2005 to signify this change: *The free lunch is over*.

When reworking applications with respect to these technological changes, different objectives have to be considered. On the one hand absolute runtime performance, such as the time someone has to wait for the results of a computation, is an important criterion. On the other hand the time and costs invested into improving and speeding up software cannot be ignored, neither in academia nor in corporate environments. Recently, due to a consistently increasing electricity rate power efficiency is becoming an important attribute as well.

These objectives are partially conflicting with each other and need to be properly balanced with respect to the specific task at hand. Investing several man-months to speed up an application by only a few percent typically only pays off for important and critical applications that are either very long running or are executed very frequently. This is especially true for low-level optimizations that improve performance on very specific hardware, as these optimizations are often ineffective on other or future architectures.

Various techniques to efficiently improve the runtime performance of industrial simulation software are devised and evaluated in this thesis, while also taking cost effectiveness into account. These techniques include strategically placed and confined modifications as well as large scale changes and the selection of appropriate runtime parameters during execution. They apply to threaded applications running on individual multiprocessor workstations as well as applications executed on compute clusters using a message passing scheme for parallelization. Even for parallelized applications the sequential performance of individual threads or processes is of high importance, so improving this by taking advantage of typical hardware characteristics of modern processor architectures is also within the scope of this thesis.

The industrial simulation software packages investigated in this thesis are real life applications used by industry partner ABB and originate from the area of power devices and networks. They serve as a representative sample of software actively used in the field and each of them is used to exemplarily present and evaluate specific performance improvement techniques.

The first application performs a thermal simulation of a power transformer. Partly based on source code written during the late 1980s and early 1990s and originally not parallelized, it is representative for sequential applications with a dated core that fail to efficiently utilize modern processors due to their internal structure and design.

Next are a set of applications computing the electrostatic field of high voltage devices. While all basically compute the same electrostatic field, they are based on different mathematical foundations and parallelization paradigms. These applications represent the outcome of a decision to completely rewrite an application from scratch for improved performance and resource usage, providing the opportunity to ponder the respective advantages and disadvantages and perform a benefit cost analysis.

Lastly, an application dealing with the reliability of power networks and the effects of possible disruptions in it is selected from a third business area. It is representative of a computation that can be parallelized in a straightforward way, but encounters difficulties in practice when scaling up to high numbers of threads.

The hardware systems used in this thesis to execute the simulation software packages and to highlight and evaluate the presented techniques and improvements are two multi-core computer systems as well as a many-core based architecture, specifically a Xeon Phi accelerator card. All three systems are based on the well-established x86 architecture and are presented in more detail in Section 3.3.4.

1.2 Thesis Outline

Following this introductory chapter, a short survey of the history of microprocessors is presented in Chapter 2. It summarizes how their computational capabilities as well as their complexity have increased over time, details some of their characteristic traits affecting performance and introduces the many-core architecture, featuring a significant number of processor cores for massively parallel processing. Additionally, the memory hierarchy is briefly covered and related to development in processor performance. Afterwards, Chapter 3 covers some basic principles, from the big *O* notation to describe growth rates and different parallelization techniques to performance evaluation.

Each of the next three chapters then covers one of the areas from which the simulation software packages have been chosen. Specifically, Chapter 4 deals with the application for simulating the thermal behavior of power transformers. A step by step tutorial on improving reliability and performance is presented, including adapting data structures and functions to take the characteristics of current processors into account, as well as introducing dynamic code generation and parallelization.

Chapter 5 then discusses four applications and a strategically modified version of one of them to compute the electrostatic field of high voltage devices. The respective advantages and disadvantages of the different approaches and their efficiency regarding modern processor architectures are highlighted.

The power networks simulation is investigated in Chapter 6 and reasons for the reduced parallel efficiency are identified. Based on this, techniques to enhance it are presented, which cover strategic modifications to the application as well as improving the runtime parameters during execution.

Chapters 4 to 6 are mostly self-contained and focus on the relevant applications, while Chapter 7 provides an extensive and general conclusion covering all three areas and the techniques and findings from the previous chapters. Finally, the last chapter contains a short synopsis as well as an outlook on related future research topics.

1.3 Cooperation

The chair for *Computer Technology and Computer Organization*[1] has a long standing cooperation with ABB that goes back to the 1990s and, over the years, provided contacts with several different departments of ABB all over the world.

One of the first joint projects was the configuration and installation of a Beowulf cluster [75] in 1999, the first cluster at ABB [16]. At that time compute clusters only rarely got utilized in industry.

Later, this cooperation culminated into the four-year *CASOPT: Controlled Component- and Assembly-Level Optimization of Industrial Devices*[2] project, which started in 2009 under the umbrella of the *Seventh Marie Curie Framework Program* (FP7)[3], funded by the European Union.

The CASOPT project consortium consisted of industry partner ABB and the academic partners University of Cambridge, Technische Universität Graz and the Computer Technology and Computer Organization chair of Technische Universität München.

Over many years, the cooperation provided an invaluable opportunity to work on joint projects and exchange experience and know-how between industry and academia.

[1]`http://www.lrr.in.tum.de` (retrieved January 2014)
[2]`http://www.casopt.com` (retrieved December 2013)
[3]`http://cordis.europa.eu/fp7/home_en.html` (retrieved December 2013)

It also made it possible to use real life applications, that are actively used by numerous engineers all around the globe, for research and, by means of various student projects, to let students and scientists catch a glimpse of corporate research.

This cooperation provided the basis for research and experiments conducted in this thesis, as well as access to the simulation software used as representative sample.

Chapter 2

Multi- and Many-Core Architectures

This chapter presents a short survey of the history of microprocessors and how their performance as well as their complexity have increased over time. At first this is solely done by raising clock frequency and by adding architectural improvements. However, for some years now the trend has shifted to introducing parallel processing by using a growing number of processor cores and vector units.

Also, the hierarchy of the memory subsystem from caches integrated into processors up to non-uniform memory access at the main memory level and how it affects the achievable compute power is considered.

An excellent and detailed presentation of computer architecture in general can be found in Hennessy and Patterson [38]. The following summary highlights the most important aspects and provides the basis for the remainder of the thesis.

2.1 Moore's Law

In 1965 Gordon Moore pointed out, that up to this point the number of transistors on integrated circuits roughly doubled approximately every year and he expected it to continue to do so for the next 10 years [60]. This was later revised to doubling approximately every *two* years [61]. In 1970 the term *Moore's Law* was coined to describe Moore's observation and prediction, which is still used as reference point today.

Initially, Moore's Law was an observation and attempted a limited forecast into the future. However, as it proved to be correct, it became a guideline for industry, research and marketing. In a way it can be regarded as a self-fulfilling prophecy.

Moore's Law has been declared obsolete several times during the past, as further miniaturization was deemed unfeasible, but technological advances have so far managed to keep it alive and valid. Nevertheless, at least miniaturization will reach an ultimate physical limit when the size of structures on a microprocessor will approach the size of atoms.

2.2 Sequential Processing

Until the mid-2000s new processor designs were almost always attended by a significant increase in base clock frequency. Additionally, new processor generations featured performance improvements through e.g. automated prefetching of data from memory, pipelining and out-of-order execution to name but a few.

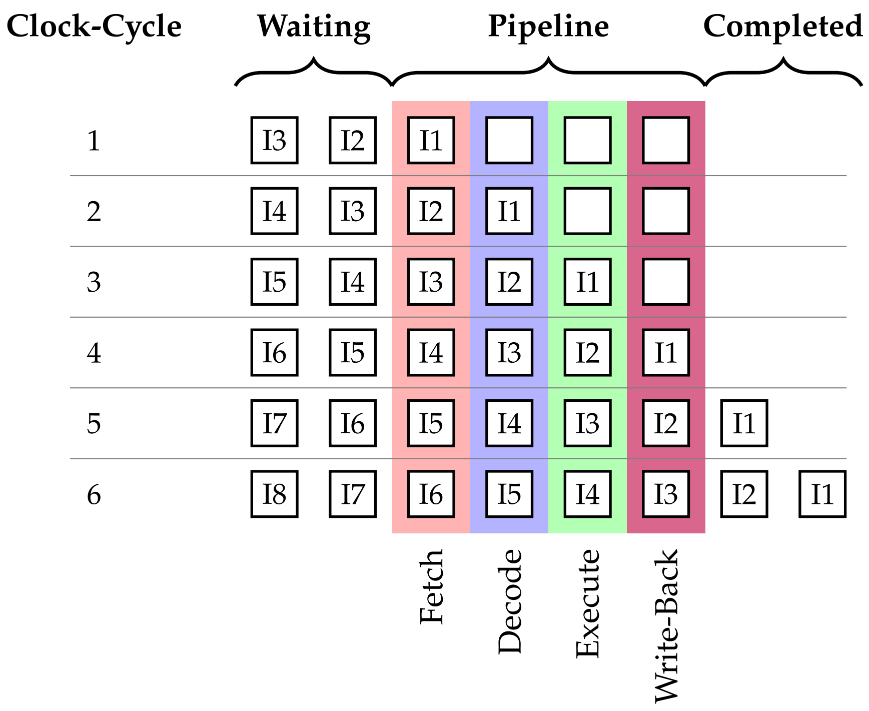


Figure 2.1: Processor Pipeline

When running at very high clock frequency, processors are no longer able to fully load, decode and process a single instruction during one cycle. To nevertheless leverage the high frequency, processing an instruction is split into multiple steps and processed using a pipeline, where each stage of the pipeline is able to process a specific step of an instruction [38, appx. C].

Common steps in processing an instruction are *fetch*, *decode*, *execute* and *write-back*, but many more are possible. As visualized in Figure 2.1, a pipeline featuring four stages can then process four different steps of four different instructions at each stage, virtually processing the instructions in parallel.

At clock-cycle one the first instruction is fetched and enters the pipeline. During the next clock-cycle instruction one is decoded, while the second instruction is fetched and so on. At the forth clock-cycle, all pipeline stages are filled and four instructions are being processed concurrently. Afterwards, processing of one instruction is completed at every clock cycle as long as the pipeline remains filled.

When processing two dependent instructions, however, this might not be possible. If the second instruction requires the result of the first, processing of the second one has to be stopped until the first one is completely processed, i.e. the pipeline is *stalled*, resulting in reduced performance.

Similarly, if the executed code branches based on some condition, the processor does not yet know which instructions to load into the pipeline, possibly resulting in a pipeline stall. To mitigate this issue, the processor tries to predict which branch is taken based on statistical data from previous executions and speculatively starts processing the appropriate instructions. If the prediction is correct, execution can be continued as usual, otherwise the speculative execution has to be aborted and processing of the correct instructions has to be started. This technique is called *branch prediction* and helps to keep the pipeline filled.

In practice processor pipelines consist of considerably more stages than the four exemplarily mentioned above. Intel's *NetBurst* architecture, the basis for the Pentium 4 processor series, was designed for very high clock frequency and subsequently featured up to 31 stages in its last revision. Its successor, the redesigned *Core* architecture, reduced the number of stages to fourteen to mitigate the adverse effect of branch mispredictions among others [42, 43].

All in all, these stages are not exclusively used to allow for increased clock frequency, but also to incorporate and support additional architectural features, such as *out-of-order execution* [50], which automatically rearranges the processing of individual instructions to improve overall utilization of the processor. Instructions are no longer necessarily processed in their original order. The processor may move and delay an instruction waiting for a data item to be retrieved and instead execute other independent instructions until the data item is available. This hides memory access latencies and further helps to keep the instruction pipeline filled.

2.3 Parallel Processing

All architectural improvements and the base clock frequency increase discussed above have one thing in common: They result in an automatic and significant performance improvement for software that is executed on a new processor architecture, typically without even the need to recompile it. For instance, users who required more performance to achieve shorter application runtime could buy a newer processor and the application would automatically run faster.

However, at one point when trying to increase clock frequency even further beyond 3 to 4 GHz, engineers started to struggle with increased power consumption leading to significant heat dissipation and difficulties to efficiently cool the processor. To nevertheless be able to further increase overall performance and adhere to Moore's Law, the main focus for research and development shifted towards parallel processing.

2.3.1 Classification of Parallel Processing

Parallel processing was used to speed up computations from the early beginnings of microprocessors. As early as 1966 Michael J. Flynn presented a formal classification of different types of parallel processing, known today as *Flynn's taxonomy* [25, 26]. However, at that time, parallel processing or even computerized processing at all was far from being mass market, but almost exclusively special-purpose applications.

Flynn used the number of instruction and data streams used in parallel during processing as reference and distinguished four different types: The sequential *Single Instruction Stream, Single Data Stream (SISD)* type executes a single stream of instructions which operate on a single stream of input data.

A *Single Instruction Stream, Multiple Data Stream (SIMD)* system on the other hand executes a single stream of instructions simultaneously on multiple input data streams in parallel, such as computing the sum of two vectors of a certain length with a single instruction. Although less commonly used, a system providing *Multiple Instruction Stream, Single Data Stream (MISD)* can for instance be used for fault-tolerance by independently executing instructions on the same data on different parts of the system and verifying the validity of the result by comparing the individually computed ones.

Finally, the fourth classification is *Multiple Instruction Stream, Multiple Data Stream (MIMD)*, supporting the simultaneous execution of different instruction streams operating on multiple data streams.

2.3.2 Single Instruction, Multiple Data (SIMD)

The first widely-used SIMD instruction set for the commonly used x86 architecture was *MMX* by Intel in 1997 [43]. It supported integer operations on eight 64-bit registers, each of which could be used to store and process a single 64-bit integer or a vector of two 32-bit integers, four 16-bit integers or eight 8-bit integers concurrently.

It was the first step to introduce the single instruction, multiple data concept to the mass market and, in 1999, was followed by the *Streaming SIMD Extensions (SSE)*, adding 128-bit registers and support for floating-point operations. Since then, new processor designs have almost always included new instructions to support additional and more efficient operations using these registers. In 2011 the *Advanced Vector Extensions (AVX)* were released, extending the size of the registers to 256 bit.

These extensions, also called vector instructions, can significantly improve peak performance when processing floating point data, as is commonly used for scientific computations. Contrary to earlier improvements in processor designs utilizing SIMD instructions requires explicit support by the application.

Furthermore, to efficiently use vector instructions, the data to be processed has to be appropriately placed in memory, which may require extensive reorganization of data structures within an application, limiting the immediate performance gain for existing applications.

2.3.3 Multiple Instruction, Multiple Data (MIMD)

Integrating vector registers and instructions into the processor to support SIMD operations was the first step to parallel processing within a single mass-market processor. Consequently, the next step is support for MIMD like operations, i.e. multiple instruction streams are processed in parallel.

Simultaneous Multi-Threading (SMT) The *Hyper-Threading Technology (HT)* introduced by Intel in 2002 is an implementation of *Simultaneous Multi-Threading (SMT)*, supporting the concurrent execution of two independent instruction streams. To the operating system a single HT enabled processor is presented as two *virtual* or *logical cores* on which it can execute applications in parallel – be it different applications or a single application employing multiple threads.

However, only certain parts of the processor's internals, such as registers, are duplicated to support Hyper-Threading, resulting in shared usage of the other parts. Shared resources are managed by pausing the execution of one application's instruction stream, once both instruction streams require simultaneous access to a non-duplicated part of the processor, such as floating point computation.

Because of this, applications that heavily use floating point computations, such as scientific applications, often do not benefit significantly if at all from Hyper-Threading or simultaneous multi-threading in general. However, the precise performance behavior depends on the specific nature of an application.

Multi-Core To further improve a processor's peak performance and reduce the drawbacks of simultaneous multi-threading, true multi-core technology was available when dual-core processors were presented to mass-market in the mid-2000s.

Similar to simultaneous multi-threading the processor is represented as multiple cores to the operating system, but here the processing core is duplicated as a whole and every core represents a fully functional and independent processor. These cores are normally called *physical cores*, to distinguish them from the logical cores featured by simultaneous multi-threading (Hyper-Threading). As it is no longer necessary to pause processing of an instruction stream to accommodate shared resources, this technology significantly improves performance for almost every application, even numerically intensive ones that have an emphasis on floating point operations.

The multi-core approach is typically coupled with providing SIMD registers and instructions, further increasing the achievable performance, when every physical core may operate on multiple data streams at once.

Over the years multi-core processors have become commonplace. Desktop, notebook as well as smartphone and tablet processors feature up to four physical cores, while more expensive processors targeted for high performance servers provide up to 12 physical cores. Also, this multi-core technology is often coupled with simultaneous multi-threading, as its integration is cost-effective and provides an additional performance advantage for some types of applications.

Multiple Sockets Aside from putting multiple cores into a single processor, it is of course also possible to put several processors into a single system. These systems are classified by the number of sockets they provide for processor installation, common are two or four sockets. While such a configuration is rarely used for desktop systems, it is very common for high-performance servers and special-purpose workstations.

By combining both approaches and installing multiple multi-core processors into a single system, it is possible to construct systems with 40 or more physical cores and even twice as many logical cores when additionally enabling Hyper-Threading.

2.3.4 Many-Core Architecture

As detailed above, the current trend in research and development is towards processors being equipped with an increasing number of cores as well as improved SIMD operation. The *Many Integrated Core Architecture (MIC)* introduced by Intel serves as an example for this trend by significantly increasing both the number of cores and the amount of data the SIMD instructions can process at a time. It is based on the same fundamental instruction set as the general purpose processors to ease porting to and executing applications on it.

The first commercially available product based on this architecture was the *Xeon Phi* in 2012, an extension card to be installed in a host computer system and used as an accelerator in addition to the main processors. Besides the actual MIC-based processor, the card also contains local memory and runs its own operating system, an adapted variant of *Linux*.

The processor on the Xeon Phi card features 60 physical cores and supports SIMD operations on 512-bit wide registers and data vectors, double the size of AVX available on general purpose processors today. Furthermore, it supports 4-way simultaneous multi-threading, i.e. to the operating system every physical core is visible as four logical cores, resulting in a total of 240 logical cores available for computation.

However, due to technical and power consumption constraints, the individual core is much simpler than general purpose processors: Its clock frequency of about 1 GHz is significantly lower than the 3 GHz or more general purpose processors often run at.

Also, a lot of architectural enhancements developed over time are missing. This includes hardware prefetchers responsible for automatically preloading data from memory, branch-prediction as well as out-of-order execution. Without these features, memory access latencies are not automatically mitigated by reordering instructions or preloading data. Instead, execution is halted until the data is available. At the same time, the lack of speculative execution due to branch-prediction decreases the efficiency of the processor's pipeline.

These limitations also are the reason for supporting 4-way Hyper-Threading in the first place. When the execution of an instruction stream executed on a logical core has to be halted, the shared resources can be utilized by the other three logical cores, thereby increasing overall efficiency.

All in all, while the single-core performance of Xeon Phi is not competitive with a modern desktop or server processor, it still offers noticeable higher theoretical peak compute power by supporting large vector processing and 60 cores with 4-way Hyper-Threading per core. To benefit from this architecture though, one needs an application that can be efficiently parallelized.

The Xeon Phi accelerator card supports different usage modes. One mode of operation is called *offloading*, where the application itself is started on the host system, but specifically marked and prepared parts of the application are automatically transferred to the Xeon Phi for computation [41].

Another mode of operation is to run an application natively on the Xeon Phi. The application is copied into the cards local memory and subsequently executed on the locally running operating system. As this represents the most straightforward way to execute something on the Xeon Phi, this mode is used in this thesis.

2.4 Memory Hierarchy

2.4.1 Memory Wall

As seen in the previous sections, overall performance of processors has improved significantly over the years. But to be able to leverage that performance, it is necessary to be able to provide input data fast enough to the processor to keep it going.

However, the computing performance of processors has improved much faster than the typical main memory access latency, which represents the time it takes to request and transfer data from main memory to the processor. The term *Memory Wall* was coined by Wulf and McKee [86] to describe this growing disparity of processor performance and main memory speed.

In the 1990s, typical memory latency was in the order of about 15 processor cycles, which means that after the data was requested from main memory, instructions to actually process the requested data had to be delayed for at least 15 cycles. Today the latency has increased to 200 and more cycles, resulting in an even higher number of delayed instructions. Additionally, the number of operations a processor is able to process during each cycle increased, especially when considering SIMD instructions.

All in all, waiting for data from main memory results in a significant and increasing waste of computational power.

2.4.2 Uniform vs. Non-Uniform Memory Access

In systems with multiple processor sockets the type of connection between individual processors and main memory is vital. Typically *Uniform* and *Non-Uniform Memory Access* architectures are distinguished.

Within a *Uniform Memory Access (UMA)* system, as outlined in Figure 2.2, all processors uniformly share and access the available physical memory. This means that the time it takes to transfer data from memory is independent of the data location in memory and the processor it is transferred to. Moreover, the available memory bandwidth is also shared among all processors.

On the other hand, in a *Non-Uniform Memory Access (NUMA)* architecture the memory access time does depend on the location of data in memory and which processor requested the data. There, each processor has *local* and *non-local* memory, and access to local is faster than to non-local memory. The exact difference in access latency depends on the precise architecture design. The advantage of this layout is that it allows all processors to simultaneously utilize the full memory bandwidth to the respective local memory, improving the overall amount of data that can be transferred from memory at a given time.

Typically, the available physical memory is split into as many equally large chunks as the number of available processors and every chunk is connected to a single processor. Additionally, there is a connection between the processors. However, memory access latency is not solely determined by whether the relevant memory chunk is local or not, but additionally by the structure of the processor interconnect.

Figure 2.3 shows a system with four processors and a ring connection between the processors. From processor 0's point of view, only memory 0 is local. When it accesses data from non-local memory 2, the request and the subsequent data has to be transferred across the connection between processor 0 and processor 2. Furthermore, when accessing data from memory 3, the transfer has to pass across both processors 2 *and* 3 (or processors 1 *and* 3), adding additional latency to the memory request. This effect can be avoided by adding an additional cross connection between processors 0 and 3 and processors 1 and 2.

When developing a parallelized application that is expected to be executed on a NUMA architecture the software engineer, to achieve optimal performance, has to take care that as many memory accesses as possible reference local memory. A common

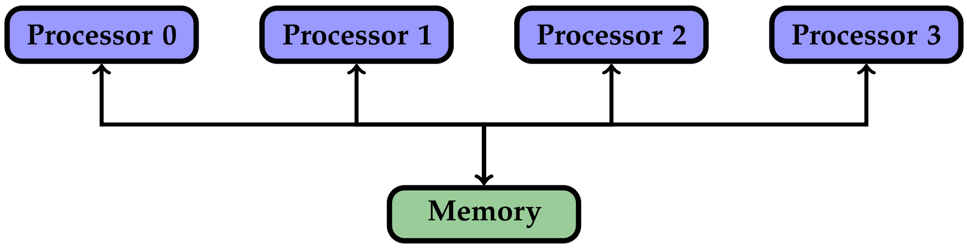


Figure 2.2: Uniform Memory Access (UMA)

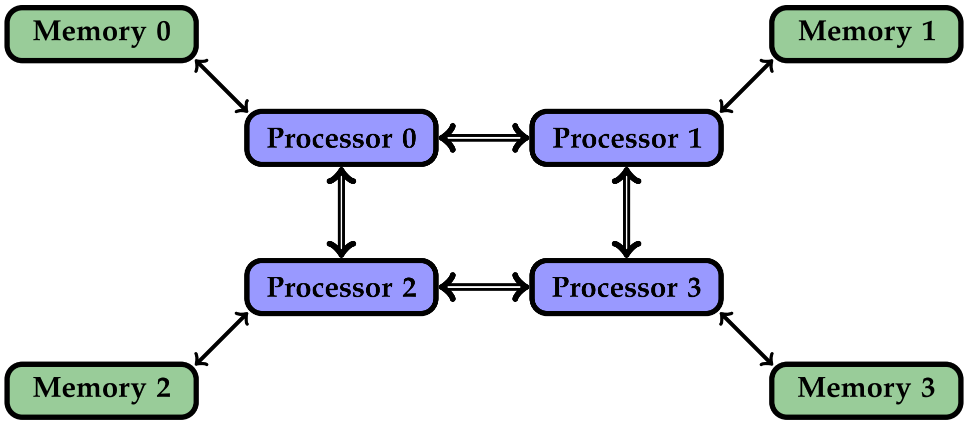


Figure 2.3: Non-Uniform Memory Access (NUMA)

source for reduced performance is an initialization or startup phase during which most of the memory later required is allocated and which is executed by a single thread.

If not explicitly configured otherwise, a *first-touch* policy is typically employed by the operating system, i.e. memory is allocated local to the processor executing the startup phase. This results in increased memory access latencies for threads later executed on different processors.

A more efficient alternative is to initialize the required threads before executing the startup phase and to ensure that every thread allocates the memory it will be using later on.

2.4.3 CPU Caches

To mitigate the growing memory access latency, processor manufactures started to embed additional memory into or very close to the processor to serve as caches. Due to cost and space restrictions these caches are considerably smaller than main memory, but can be accessed with significantly less latency, while at the same time providing higher transfer bandwidth.

The caches itself are arranged in a hierarchy, with the *Level 1 (L1)* cache being the smallest but fastest one. Further levels generally increase in size, while access performance decreases. The cache at the highest level, L2, L3 or nowadays even L4, is normally also called *Last Level Cache (LLC)*.

In modern multi-core processors, there typically is an L1 and an L2 cache per physical core, and an L3 cache that is shared by all physical cores. Typical sizes for the L1 cache are 64 KiB and 256 KiB for the L2 cache, while the L3 cache holds several MiB.

To reduce management overhead and increase transfer efficiency between caches and main memory, data is transferred in blocks called *cache lines*. While the exact size of a cache line depends on the processor, common values are 32, 64 or 128 bytes.

When data is requested from main memory, a full cache line containing the requested data is transferred from memory into the processor cache. To create space, another, already stored cache line has to be removed from the cache (called *eviction*) and the relevant cache line is selected by a replacement policy implemented into the processor. Ideally, one would evict the cache line that is least likely to be used in the future. As the processor cannot know this, a *(Pseudo) Least Recently Used (LRU)* scheme is often employed instead, although other methods such as *Dynamic Insertion Policy (DIP)* [69] or *Re-Reference Interval Prediction (RRIP)* [48] are possible as well.

The organization of data as cache lines also improves performance of applications that exhibit locally grouped data access patterns. If an application needs two data items stored closely together in memory, chances are good that the second data item is part of the same cache line as the first item. Then, after the cache line was transferred to the cache at the access of the first item, access to the second item is fast, as it is already stored in the cache. This is known as *spatial locality* and has significant impact on the efficiency of caches [20].

To further reduce decreased performance due to waiting for data requested from main memory, most processors employ a *hardware prefetcher*, which tries to automatically preload data from main memory that is anticipated to be used in the future. This is done by trying to recognize repeated memory access patterns of an application and using those to predict the data required in the future. The transfer of this data from memory may then be initiated ahead of time.

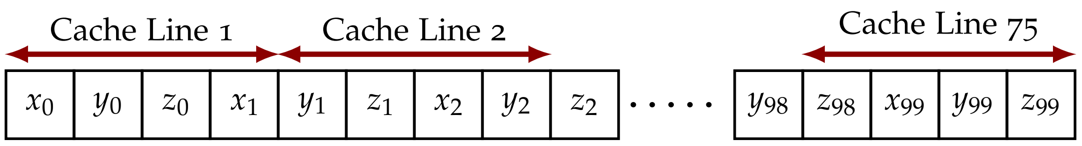
If predicted correctly, the data is already stored within the fast processor cache at the moment it is required for processing, thereby hiding the memory access latency. On the other hand, performance can also be negatively affected in case of a misprediction. A cache line that is transferred from memory and is not going to be used will needlessly evict another, possibly still to be used, cache line.

2.4.4 Cache Efficient Memory Access

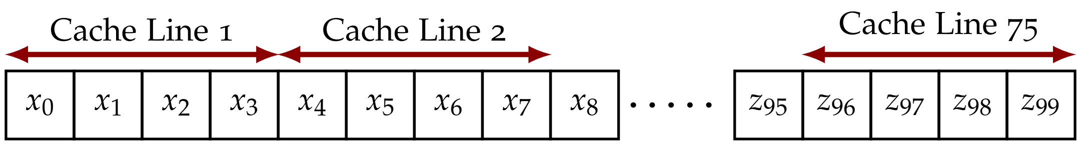
Nevertheless, to obtain maximum performance, it is also the responsibility of the software developer to appropriately design an application and its data structures to efficiently use the processor's caches. This typically includes grouping and storing data in such a way that the required data is scattered across as few cache lines as possible to improve cache efficiency. Often, this boils down to choosing the optimal data structures for the task at hand.

This is best illustrated by an example: Considering a set of 100 points in three-dimensional space, each consisting of an x, y and z coordinate, $p_0, p_1, \ldots, p_{99}$, $p_i = (x_i, y_i, z_i) \in \mathbb{R}^3$, two different basic schemes to store them in memory come to mind:

The first one stores one point with its three coordinates after the other, while the other one groups together the coordinates of the individual points. These layout schemes are typically called *Array-Of-Structs (AoS)* and *Struct-Of-Arrays (SoA)*, as they result from using the corresponding data structures of the C programming language and are both depicted in Figure 2.4


Cache Line 1
Cache Line 2
Cache Line 75
x_0 y_0 z_0 x_1 y_1 z_1 x_2 y_2 z_2 ..... y_{98} z_{98} x_{99} y_{99} z_{99}

(a) Variant 1: *AoS*


Cache Line 1
Cache Line 2
Cache Line 75
x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 ..... z_{95} z_{96} z_{97} z_{98} z_{99}

(b) Variant 2: *SoA*

Figure 2.4: Storage layout for points in three dimensional space

Let us assume an application needs to compute the sum of all x coordinates $\sum_{i=0}^{99} x_i$ and that a cache line holds exactly four coordinates. To consecutively store all 300 coordinates 75 cache lines are necessary. When processing all x coordinates using the first storage variant *AoS*, all 75 cache lines need to be transferred from memory, as each one contains at least one point's x coordinate.

In contrast, when utilizing the second scheme *SoA*, only the first 25 cache lines contain x coordinates, resulting in three times less data transferred from main memory as well as improved spatial locality, thereby increasing overall efficiency.

Chapter 3

Basic Principles

Some of the most basic principles when dealing with performance on computer systems are recapitulated in this chapter.

It begins with a formal notation describing a characterization of the growth rate of an algorithm and afterwards shortly discusses two essential parallelization schemes. Finally, the chapter closes with some details about performance evaluation in general and the specific hardware systems utilized for evaluation in this thesis.

3.1 Big O Notation

To make use of increasing computing capabilities, input data sets tend to grow over time to solve a problem that was previously infeasible or to improve the accuracy of an existing computation.

Because of this, when considering the performance of algorithms to process or store data, it is more interesting to know how it characteristically behaves for varying input data than the absolute time it takes to process a data set of fixed size.

To classify an aspect of this behavior of an algorithm, the *big O notation* is used, which was published by Bachmann [5] in 1894. Formally, for functions $f(x)$ and $g(x)$, one writes

$$f(x) = O\left(g(x)\right)$$

if and only if there exists a positive constant value c and a constant value x_0 such that

$$|f(x)| \leq c|g(x)| \quad \forall\, x > x_0.$$

Informally, this means that, except for some constant factor, the function $g(x)$ represents an upper bound to the growth rate of $f(x)$.

Let us consider the multiplication of an n dimensional matrix $M \in \mathbb{R}^{n\times n}$ and a vector $v \in \mathbb{R}^n$ as an example. The calculation performed to compute the individual components of the result vector w is

$$w_i = \sum_{j=1}^{n} m_{i,j}v_j = m_{i,1}v_1 + m_{i,2}v_2 + \cdots + m_{i,n}v_n, \quad i = 1,\ldots,n$$

For every component w_i of w a total of n multiplications have to be performed and the results need to be summed up. Overall, this results in $2n-1$ operations per component of w and a total of $2n^2 - n$ operations to compute the full vector w. The matrix-vector-multiplication can therefore be classified as $O(n^2)$ for dimension n.

Using simple reference functions, such as n^2, makes it easy to quickly assess the growth rate of an algorithm. In case of vector-matrix-multiplication, a 10-fold increase in dimension would result in roughly 100 times more operations to compute the resulting vector. The notation $O(1)$, which is also used in this thesis, expresses a constant upper bound that is independent of the input data size.

It is important to note that this notation only specifies an upper bound, while the actually observed growth rate can be lower. If additional or more precise bounds are required, other notations exist as well, such as Ω to specify a lower bound, or Θ, which is used to specify both a lower and upper bound. These notations, including the big O notation, are called *Landau Symbols* and are named after the German mathematician Edmund Landau who suggested the little o notation in addition to Bachmann's [55].

In Cormen et al. [19] these notations are introduced in more detail and their relation to each other is discussed. Moreover, a multitude of different algorithms covering many different application areas are presented and their asymptotic behavior is analyzed and compared, including, but not limited to, searching and sorting algorithms, various data structures with different properties as well as advanced design and analysis techniques. All in all, this makes it an excellent work of reference.

3.2 Parallelization Techniques

A fundamental question when trying to speed up the computation of a problem by parallelizing it is how to distribute and transfer data amongst the different streams of execution. Two different paradigms to answer that question are *shared memory* parallelization on the one hand and *message passing* on the other hand.

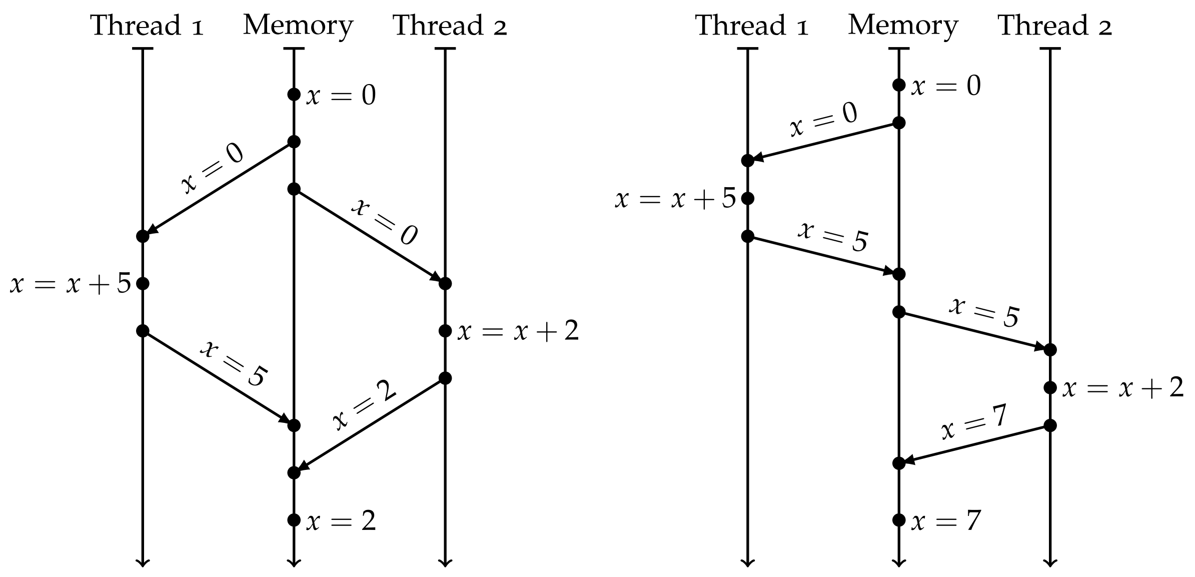
3.2.1 Shared Memory

When employing shared memory parallelization, an application features multiple execution streams, called *threads*, running in parallel on different cores within a computer system. Every thread has full access to the application's global data and is able to process and modify it without restrictions, hence shared memory.

At first glance, using shared memory parallelization is easy to get started, as it is not necessary to think about how to distribute data for parallel processing. However, great care must be taken to avoid conflicting access to the same data.

Figure 3.1 shows such a conflicting access, with two threads incrementing a variable stored in global memory to update a counter e.g. representing the number of data items processed by the threads. In 3.1a both threads load the original value 0 from shared memory, increment a locally stored copy and subsequently write the incremented value back to shared memory. As *Thread 2* performs its write back at a later point, it overwrites the increment computed by *Thread 1*, which is then lost. In 3.1b, however, access to the global variable is properly serialized by ensuring that *Thread 2* defers loading the variable until *Thread 1* properly stored its own computation.

This dependency of the globally computed result on the (possibly random) execution order of the threads is known as *race condition*. In this illustration, the timing of the



(a) Conflicting Access (b) Serialized Access

Figure 3.1: Race condition when updating global value

threads leads to a wrong result without raising any errors, making it hard to find the source of the problem. Such race-conditions can also happen during access to relevant data structures, possibly resulting in application crashes.

At the same time, shared memory parallelization is typically confined to a single computer system. The number of processor cores provided by that system limits the achievable speedup and therefore its suitability for very large scale parallelization.

When using shared memory parallelization, one uses an application programming interface (API) and libraries that provide certain functionality to create threads and control their execution. At the very bottom level, these APIs are provided by the operating system, such as *POSIX Threads (Pthreads)* [46, 18], available on most Unix-like systems.

An alternative are more high-level APIs providing a consistent interface across multiple operating systems, such as *OpenMP* [65], published by the OpenMP Architecture Review Board. It uses annotations inserted into the source code of an application, to specify how the compiler should generate parallelized code. It is possible to mark sections of code to be executed in parallel, including how to handle parameters and shared data. When the compiler processes such a section, it automatically extracts the section's content to a newly generated function and then adds additional code at the original location to call and execute the externalized function multiple times in parallel.

Other annotations include synchronization commands, such as barriers, which force all parallel execution streams to pause at the point of the annotation, until every single execution stream has reached this point. This can be used to avoid conflicting accesses by ensuring that at the end of a parallelized computation stage, all results are available before proceeding to the next stage to further process the results.

In summary, OpenMP provides a straightforward way to add parallel processing to an application using the shared memory paradigm, typically without the need to rewrite large sections of the application.

3.2.2 Message Passing

The second parallelization paradigm requires the programmer to explicitly handle data exchange between parallel execution streams. This data exchange is realized by passing messages from one execution stream to the other, coining the name *message passing*.

Instead of being able to directly access all data, the individual execution streams only have access to their own local memory. If data from another execution stream is required, it has to be explicitly transferred from the other stream's private memory.

While this may look like a constraint at first, it does have several advantages. First, it is considerably harder to accidentally create conflicting accesses. As other execution streams do not have direct access to private data and data structures, they cannot mistakenly tamper with them.

Also, message passing makes very flexible configurations possible. Parallelization is not limited to a single system, but may instead utilize multiple systems connected by some kind of network used to transfer messages. Such a pool of systems is called a *computer* or *compute cluster* and distributing the parallel processing across its nodes makes the message passing scheme appropriate for large scale parallel processing.

The *Message Passing Interface (MPI)* reference by Snir et al. [73] constitutes a standardized, well-defined API, providing functions to send and receive messages, as well as synchronize different execution streams. By using these functions, programmers can parallelize their applications using the message passing paradigm. MPI in itself does not represent an actual implementation, but is instead the "template" all implementations must adhere to. Notable implementations are *OpenMPI* [31] or *MPICH* [35], which are both freely available as open source.

As long as the application programmer abides by the MPI standard, the parallelized application should work correctly on either conforming MPI implementation. This allows for very specialized implementations, which are tailored to specific hardware configurations and network types.

3.3 Performance Evaluation

3.3.1 Key Aspects

When evaluating the performance of algorithms, applications and the like, it is important to state the key aspects one wants to consider. Naturally, one of the first things that comes to mind is the absolute time one has to wait from the start of a computation until the result is returned, also called *wall-clock time*. However, when assessing performance, there are other important aspects as well.

One of them is the amount of memory that is required during the computation. At first sight, this merely determines if it is feasible to perform the computation on a given computer system or how much memory a newly procured one must provide. But memory consumption is also influenced by design choices and implementation details of an algorithm or application.

Often, different performance metrics influence each other and trying to improve one adversely affects the other. Two algorithms can have very different memory requirements, even though both of them basically compute the same result. At the same time, the algorithm with the lower memory footprint might be slower and needs more time to compute the result. In this case, it is necessary to prioritize and decide which aspect is more important or critical for the given situation.

Another often used performance metric for parallel processing is the processor or *CPU time*. It is used to show the amount of time the computation is actively using the processor and excludes time spent waiting for e.g. data loaded from hard disk or over a network. CPU time of an application running on a single processor is always less or equal than wall clock time. When multiple cores are used, however, total CPU time will typically be significantly higher than wall clock time and increase with the number of cores used.

3.3.2 Performance Counter

To help developers in evaluating the performance of their software, processor manufacturers started to include special-purpose registers, called *hardware performance counters*, into their processors.

Once configured and activated, the processor uses them to store the count of specific hardware-related events, such as cycles elapsed or instructions processed. As gathering and storing these statistical values is directly performed by the processor, it can be used to analyze virtually every application without any noteworthy effect on the performance itself.

The specific types of events supported by a processor and how to configure and activate them depends on the precise processor model and underlying architecture and is documented in the technical documentation of the processor manufacturer, such as the *System Programming Guide* [45] for Intel processors or the *BIOS and Kernel Developer's Guide* [1] by AMD.

Commonly supported events include the aforementioned cycles and instructions, as well as more complex ones such as cache or branch misses. The cache misses event counts the number of data loads which could not be satisfied by any processor cache, but instead resulted in an actual transfer from main memory. The branch misses count is the number of times the processor wrongly predicted the branch to be taken at a conditional instruction, typically resulting in performance loss.

By regularly reading those values from the hardware counters and comparing them to the state of the running application, it is possible to create a profile of the application. This profile can then be used to e.g. identify regions of the application that are slowed down by a high number of cache-misses. Subsequently, a software developer can investigate if using a different data structure or algorithm improves performance at this point.

It is important to note that the precise meaning of an event type is also hardware dependent and cannot directly be compared amongst different processors. The number of cache-misses reported by the performance counters might or might not also include data items automatically prefetched from memory, and the precise implementation of the prefetcher may differ considerably between different processors, especially from different manufacturers.

3.3.3 Performance Analysis Tools

To save oneself the tedious task of manually interacting with performance counters, one uses existing performance analysis tools. These tools are programmed to be able to properly configure and activate the performance counters on various processors from different vendors.

Additionally, to provide the programmer with enhanced information, they typically compute additional statistics based on the raw readout of the performance counters, such as the percentage of data loads that resulted in cache misses or the average *instructions per cycle (IPC)*.

These statistical values help in assessing the overall efficiency of an application. The absolute number of cache misses often is of little significance if these cache misses only happen at a tiny fraction of data accesses, while the IPC rate provides a good indication whether the processor is used to its full potential.

Finally, if debugging information and the source code of the profiled application is available, these tools usually also provide detailed information about the number and type of events which occurred while executing a specific function or even a specific line of code. This allows for very strategic and targeted optimizations.

An often used performance analysis tool is *perf*[1], a free and open source program, which is very closely coupled with the Linux kernel and developed alongside with it. At the same time, the processor manufacturers themselves often develop and distribute proprietary closed source tools specifically tailored to their own processors, such as *VTune*[2] by Intel or AMD's *CodeXL*[3].

3.3.4 Used Hardware Platforms

In this thesis three different computer systems are used to highlight key aspects or differences in behavior or performance of the simulation software packages or parts of it. They consist of the two-socket *Westmere-EP* system, the four-socket *Westmere-EX* system and a *Xeon Phi* accelerator card.

The Westmere-EP system is a medium range server with two Xeon X5670 processors and 36 GiB DDR3 memory. Every processor runs at a base clock frequency of 2.93 GHz and features 6 physical cores as well as Hyper-Threading, resulting in 12 logical cores per processor and 24 logical cores overall.

The four-socket Westmere-EX system contains four Xeon E7-4850 processors, each running at a clock frequency of 2.0 GHz. Every processor has 10 physical and 20 logical cores due to Hyper-Threading. All in all, 80 logical cores are available. The system provides 256 GiB of DDR3 memory, which is why it has been used for the memory intense simulations in Chapter 5.

The third hardware platform to be used is an Intel Xeon Phi 5110P accelerator card providing 60 physical cores and 8 GiB of on-board GDDR5 memory. Contrary to the 2-way Hyper-Threading implemented by the general purpose processors used in the other platforms, the Xeon Phi features 4-way Hyper-Threading, resulting in 240 logical

[1]`http://perf.wiki.kernel.org` (retrieved November 2013)
[2]`http://software.intel.com/en-us/intel-vtune-amplifier-xe/` (retrieved December 2013)
[3]`http://developer.amd.com/tools/heterogeneous-computing/codexl/` (retrieved December 2013)

	Westmere-EP	**Westmere-EX**	**Xeon Phi**
architecture	multi-core	multi-core	many-core
processor cores (physical / logical)	12 / 24	40 / 80	60 / 240
base clock frequency	2.93 GHz	2.00 GHz	1.05 GHz
memory	36 GiB	256 GiB	8 GiB

Table 3.1: Hardware platforms used

cores all in all. However, the cores are clocked at a comparatively low frequency of 1.05 GHz.

Table 3.1 summarizes the most important characteristics of the individual platforms. In Chapter 2 the hardware characteristics and especially the differences between general purpose processors and the Xeon Phi accelerator were presented in more detail and context.

As discussed before, the individual Xeon Phi processor core is much simpler than a general purpose server processor and is therefore missing key features such as out-of-order execution or automatic memory prefetching. On the other hand, the Xeon Phi provides 60 physical and – due to 4-way Hyper-Threading – 240 logical cores, 512 bit wide vector registers and instructions (as opposed to 128-bit on the Westmere systems) and high memory throughput, resulting in high processing performance regarding floating point operations.

To sum up, an application must feature a high parallel efficiency and extensively use vector instructions to fully benefit from the Xeon Phi architecture.

Chapter 4

Case 1 – Thermal Simulation

4.1 Background

Today, power transformers are typically designed and produced according to the customer's specification and specific requirements and are then produced in small quantities only – often solely a single transformer for a specific site of operation.

For many years, power transformers have mostly been designed only by experience and some fixed set of fitted formulas, which had been acquired and developed over the years. Amongst others, these formulas are used to calculate the required amount of cooling for a particular configuration.

However, this development process makes it difficult to design new generations of transformers featuring different characteristics, such as different sizes, materials and so on, as they differ too much from those that the designers and their fitted formulas are familiar with.

In practice, this lack of experience is typically compensated by an increased safety margin to fulfill the customer's specifications. To ensure sufficient safety margin, the manufacturer has to e.g. use more or more expensive materials, which increases production costs and reduces the profit.

By fully simulating the thermal behavior based on physical principles instead of using fitted formulas, the manufacturer is able to design a power transformer which satisfies the customer's specifications while at the same time production costs are kept as low as possible. A simulation based optimization procedure requires a high number of thermal simulations, so performance is a critical point.

In Blaszczyk, Flückiger, Müller, and Olsson [14] a novel approach to conduct this kind of simulation using SPICE[1] is presented. SPICE was originally presented by Nagel and Pederson [64] in 1973. The simulation solver covered in this chapter is based on the C implementation created in the late 1980s and early 1990s, specifically version 3f.5 which was released as source code in July 2007[2].

[1]Simulation Program with Integrated Circuit Emphasis

[2]`http://embedded.eecs.berkeley.edu/pubs/downloads/spice/` (retrieved December 2013)

4.2 Temperature Simulation using SPICE

4.2.1 Introduction

Originally, SPICE itself was developed to simulate electronic circuits or networks. However, by appropriately constructing network and network elements and correlating appropriate physical properties, such as voltage with temperature and electric current with power, it is possible to also simulate thermal and/or pressure networks as suggested by Gramsch et al. [32].

To do this, two distinct networks have been modeled using SPICE. One network represents the temperature at various parts of a transformer, the other represents the mass flow of the cooling fluid within the transformer. Table 4.1 shows the correlation between the different networks.

Additionally, special network elements are used to appropriately couple both networks to model the effects of various physical interactions within the transformer, like temperature propagation between different materials, movement and flow speed of fluids, cooling capabilities and so on.

Such a special network element would be a representation of a radiator, which is used to dissipate heat from the cooling fluid of the transformer into the surrounding air. Such an element would reduce the outgoing temperature (i.e. voltage) on the thermal network depending on the incoming temperature from the thermal network, mass flow (i.e. electric current) of the pressure network, ambient air temperature, surface area of the modeled radiator, etc.

Likewise there is an element to handle heat generated within the transformer, which influences the thermal and pressure network according to physical properties. Other elements handle the flow of the cooling liquid which is, among others, influenced by natural convection (depending on the temperature distribution) and, if used, pumps.

For extended details on the correlation of physical properties, the constructed network elements and the physical interactions and behaviors they simulate as well as examples refer to Blaszczyk et al. [14].

4.2.2 Operation Flow

When a new power transformer is designed, the specific requirements, such as maximum power or temperature, size constraints, as well as additional restrictions due to manufacturing or corporate policies are entered into the proprietary transformer design software system.

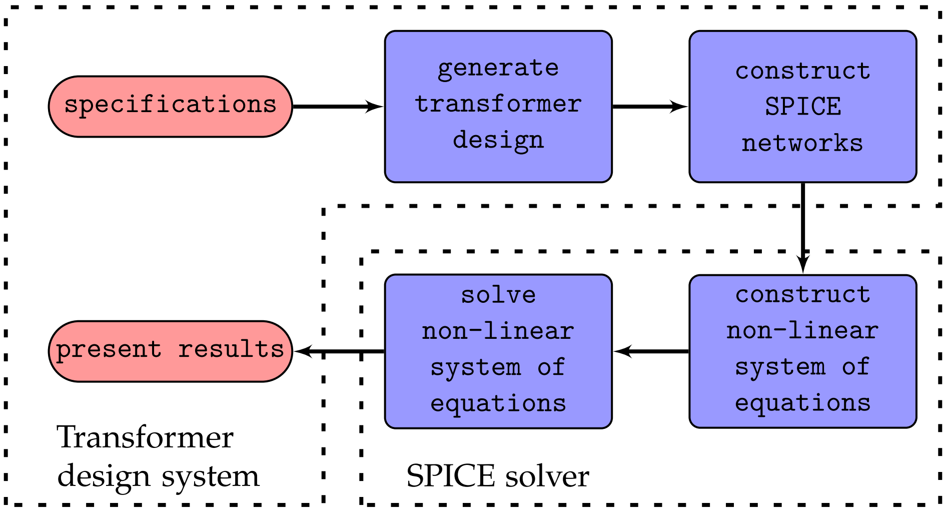
Using these specifications, the software creates a unique transformer design including materials, dimensions etc. After the design has been created, the appropriate SPICE networks are generated dynamically by the design software and fed into the SPICE-based solver to be simulated.

The solver loads the supplied network description and uses it to construct an internal representation of the equivalent electrical circuit. Complex formulas, describing nonlinear dependencies between temperatures, heat and mass flows, are attached as a separate sub-circuit library referenced by the individual network elements.

These formulas, along with constant boundary values, are used to construct a system

Electric network	Current $[A]$	Voltage $[V]$	Electric resistance $[\Omega]$
Thermal network	Power $[W]$	Temperature $[°C]$	Thermal resistance $[K/W]$
Pressure network	Mass flow rate $[kg/s]$	Pressure $[Pa]$	Flow resistance $[1/(m \cdot s)]$

Table 4.1: Correlation between quantities and units of electric, thermal and pressure networks


specifications
generate transformer design
construct SPICE networks
solve non-linear system of equations
construct non-linear system of equations
present results
Transformer design system
SPICE solver
Figure 4.1: Flowchart for simulating a new transformer design

of nonlinear equations, which is subsequently solved either directly using the Newton-Raphson method [87, 21] or, if convergence is difficult to achieve, the new fixed-point iteration method presented in Section 4.3.

Finally, the temperature distribution and cooling liquid flow within the transformer can be extracted from the results of the simulation. The workflow is represented by the flowchart in Figure 4.1.

4.2.3 Expression Tree Representation

At the inner core of SPICE the generated formulas are stored as expression trees. Every inner, non-leaf node of these trees represents a unary or binary function, e.g. $+$, $-$, $\exp()$, $\ln()$, etc. Nodes which represent binary functions like $/$ have two children – left and right – which serve as arguments to the function.

In case of a unary function like $\exp()$ the node only has a single child. Leaf nodes then either represent a constant value or a variable $(x_0, x_1, \dots)$. Finally, the tree is processed in postfix order, i.e. from bottom to top and left to right.

Figure 4.2 shows the expression tree representation of the equation

$$f(x) = e^{x_1^2+5.4} + 2x_2x_3 - x_3, \quad x \in \mathbb{R}^3 \,. \tag{4.1}$$

The Newton-Raphson method requires the Jacobian matrix, i.e. the matrix of all first-order partial derivatives of the original formulas. These are generated by explicitly and symbolically differentiating the original formulas using the well-known methods and rules of differential calculus [24].

The three first-order partial derivatives of Equation (4.1) are

$$\frac{\partial f(x)}{\partial x_1} = 2x_1 \cdot e^{x_1^2+5.4} \tag{4.2}$$

$$\frac{\partial f(x)}{\partial x_2} = 2x_3 \tag{4.3}$$

$$\frac{\partial f(x)}{\partial x_3} = 2x_2 - 1 \tag{4.4}$$

Similar to the originally generated formulas, the corresponding derivatives are stored in memory as expression trees. The derivatives of Equation (4.1) are also depicted in Figure 4.2.

Due to the chain rule in differentiation, the equation and its partial derivatives often share some expressions. In case of Equation (4.1), such a shared expression would be

$$e^{x_1^2+5.4}$$

which appears in both (4.1) and (4.2) and can be recognized by comparing Figures 4.2a and 4.2b.

As long as the variable x_1 does not change, this expression always evaluates to exactly the same result. This fact can be leveraged to speed up evaluation by caching and reusing results, as will be presented in Section 4.4.1.

4.3 Fixed-Point Iteration

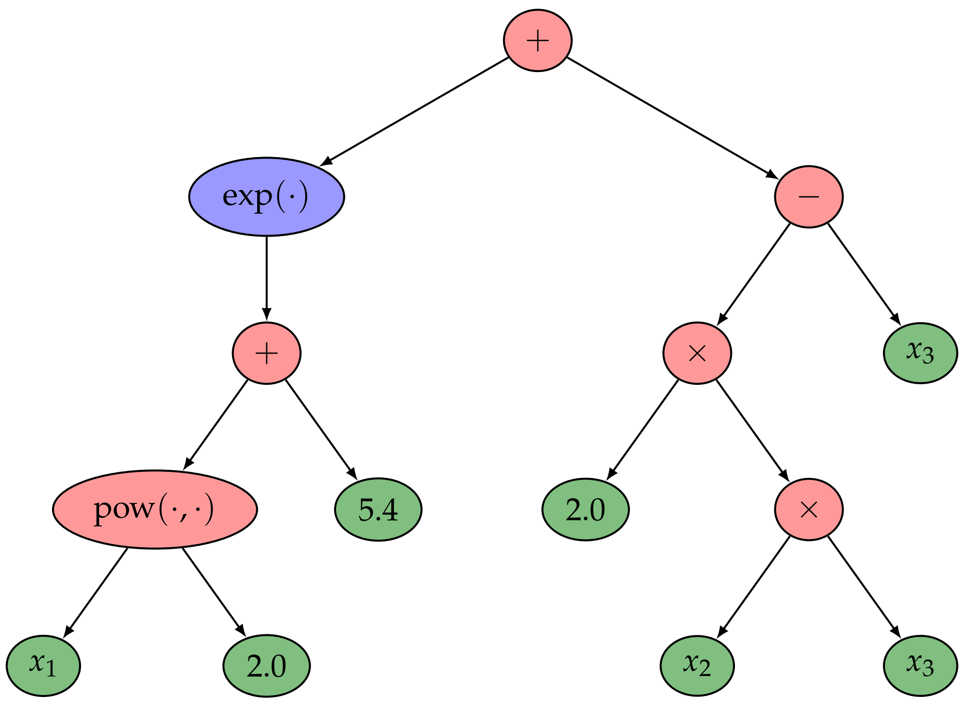
4.3.1 Separation of Networks

Due to the highly nonlinear dependencies and interactions between temperature and pressure networks, the Newton-Raphson method implemented by SPICE is typically unable to directly solve the nonlinear system of equations.

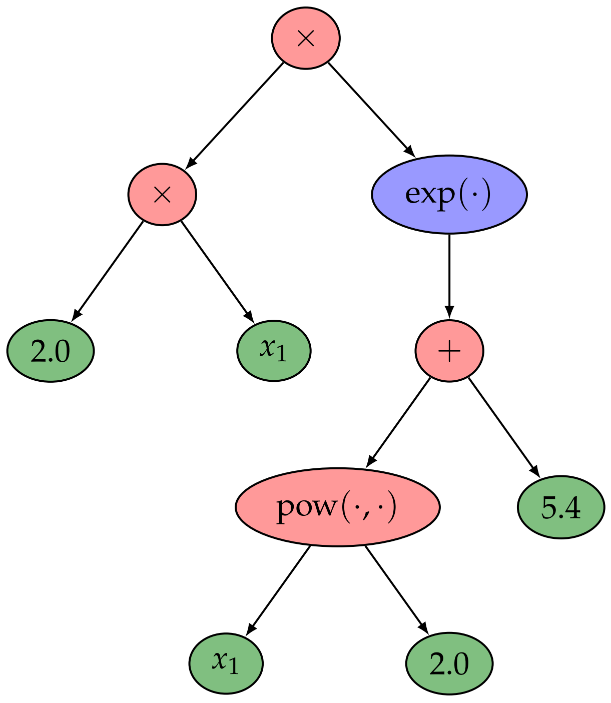
To nevertheless be able to reliably compute a solution, the coupled networks are automatically split into two separate networks – thermal and pressure/mass flow – which are independent from each other. The coupling of both networks is done by defining interface variables which substitute the direct feedback from the other network.

These interface variables include mass flow rates and velocities as a solution $x_p \in \mathbb{R}^m$ of the pressure network as well as temperatures as a solution $x_t \in \mathbb{R}^n$ of the thermal network, n, m being the number of the corresponding interface variables. Also, as the individual networks are significantly less complex than the big combined one, their simulation normally does not pose any difficulties to SPICE.

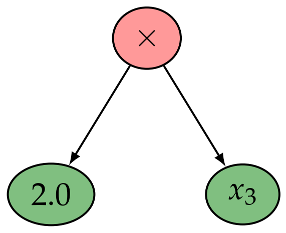
The global solution is then computed by iteratively simulating the two individual networks, while updating the appropriate interface variables at each iteration step to



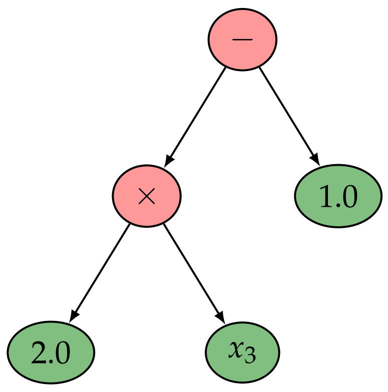
(a) $f(x)$



(b) $f_{x_1}(x) := \dfrac{\partial f(x)}{\partial x_1}$



(c) $f_{x_2}(x) := \dfrac{\partial f(x)}{\partial x_2}$



(d) $f_{x_3}(x) := \dfrac{\partial f(x)}{\partial x_3}$

Figure 4.2: Expression tree representation of Equation (4.1) and its first-order partial derivatives

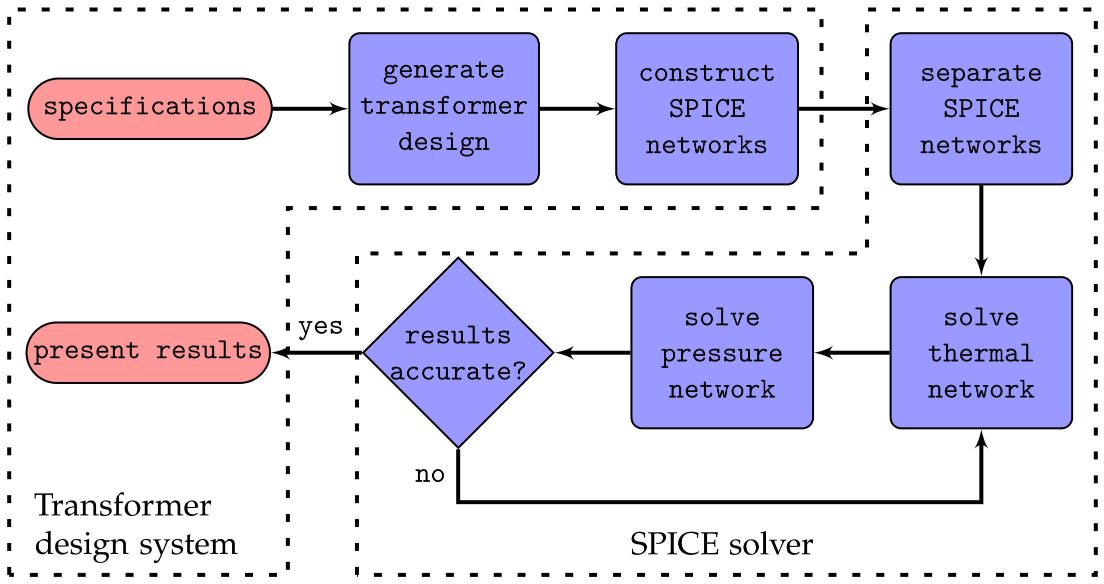


Figure 4.3: Flowchart for simulating a new transformer design with separated networks

provide the necessary feedback and interaction between both networks. The flowchart in Figure 4.3 represents the operation flow with separated networks.

This can formally be written as a fixed-point iteration where $f_t : \mathbb{R}^m \to \mathbb{R}^n$ denotes the function to compute a solution x_t of the thermal, $f_p : \mathbb{R}^n \to \mathbb{R}^m$ a solution x_p of the pressure network, and i the current iteration step index:

$$\begin{aligned} x_{p,i} &= f_p(x_{t,i}) \\ x_{t,i+1} &= f_t(x_{p,i}) \end{aligned}, \qquad x_{t,i} \in \mathbb{R}^n, x_{p,i} \in \mathbb{R}^m, i = 0, 1, 2, \ldots \tag{4.5}$$

or, more compact

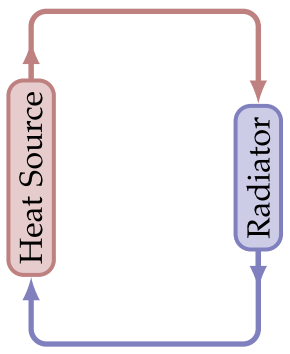
$$x_{t,i+1} = f_t(f_p(x_{t,i})), \qquad x_{t,i} \in \mathbb{R}^n, i = 0, 1, 2, \ldots \tag{4.6}$$

At the beginning of the iteration loop, the initial solution or start value $x_{t,0}$ of the thermal network must be specified. This is typically done based on rough engineering formulas for temperature calculations.

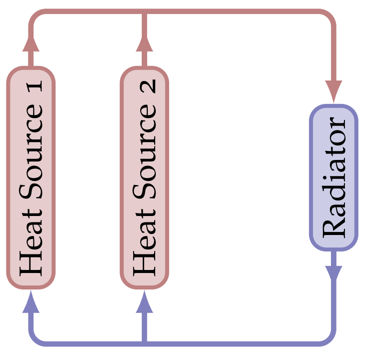
4.3.2 Convergence Considerations

As presented in Ortega and Rheinboldt [66] or Berinde [12] the convergence of a fixed point iteration like (4.5) towards a fixed point x^* is affected by the spectral radius ρ_s of the Jacobian matrix at the fixed point x^*. If $\rho_s < 1$, the iteration will converge to the fixed point x^* for a suitably chosen starting point x_0. Also, smaller values of ρ_s result in an increased convergence rate.

In Blaszczyk et al. [14] a case study with two simple transformers is carried out in which convergence and convergence rate are analyzed for a relaxed variant of the fixed point iteration (4.5). Both transformer models are symbolized in Figure 4.4. Basically, they consist of a single radiator and one (1-duct model), respectively, two (2-duct model) heat sources which are connected by cooling liquid. The heat sources increase the temperature of the cooling liquid while the radiator reduces it, as indicated by the



(a) 1-duct model



(b) 2-duct model

Figure 4.4: Simplified transformer models from case study

color of the connections inside the figure.

In the relaxed fixed-point iteration the result x_t or x_p of the network solution is not used as is, but instead the difference between the new and the previous result is calculated. Then a fraction of this difference is added to the previous result to make a step in the direction of the new result, which creates the relaxed solution x'_t or x'_p, respectively. The length of this step is controlled by the relaxation factor $\Delta \in (0;1]$.

Using f'_t and f'_p to denote the relaxed solution of the network computation f_t and f_p, respectively, the relaxed variant of Equation (4.5) can be written as

$$\begin{aligned} x'_{t,i+1} &= f'_t\left(x'_{t,i}, x'_{p,i}\right) \\ x'_{p,i+1} &= f'_p\left(x'_{t,i+1}, x'_{p,i}\right)' \end{aligned} \qquad x'_{t,i} \in \mathbb{R}^n, x'_{p,i} \in \mathbb{R}^m, i = 0, 1, 2, \dots \tag{4.7}$$

with

$$f'_t\left(x'_{t,i}, x'_{p,i}\right) := x'_{t,i} + \Delta \cdot \left(f_t(x'_{p,i}) - x'_{t,i}\right), \qquad \Delta \in (0;1]$$

and f'_p accordingly. For $\Delta = 1$ this is exactly equal to (4.5) and to start the iteration loop $x_{t,0}$ is again set to a rough estimation and $x_{p,0}$ is computed using the unrelaxed function $f_p(x_{t,0})$.

The case study shows that the correct choice of the relaxation factor is crucial for both convergence and convergence rate. For the 1-duct version, it is mostly a matter of convergence rate, as $\rho_s < 1 \quad \forall \Delta \in (0;1)$, i.e. the iteration always converges when it is started from a suitable start point for every choice of $\Delta \in (0;1)$, as illustrated by Figure 4.5. The optimal choice, however, is $\Delta = 2(\sqrt{2} - 1)$, which minimizes the spectral radius ρ_s and consequently results in fastest convergence rate.

The 2-duct model is more complicated though. Convergence behavior strongly depends on the ratio of thermal power generated by the two heat sources. The fraction of overall thermal power that is generated by one of the heat sources is specified by the parameter $k_s \in (0;1)$.

Consequently, this parameter indirectly also influences the temperature increase, flow speed, etc. of the cooling fluid. However, the model is symmetric regarding $k_s = 0.5$, i.e. the convergence behavior is identical for e.g. $k_s = 0.3$ and $k_s = 0.7$. Because of this, the following analysis is restricted to $k_s \in (0;0.5]$.

Figure 4.6 shows that the principle behavior of the dependency between the relaxation factor Δ and the resulting spectral radius ρ_s is comparable to the 1-duct model. Again, the spectral radius decreases linearly to its minimum value and starts growing rapidly afterwards, but the spectral radius is no longer less than 1 for every $\Delta \in (0;1)$, demonstrating that in this case the relaxation factor not only influences the convergence rate but also convergence itself. Even worse, the permissible interval for Δ to ensure convergence shrinks noticeably with increasing imbalance in thermal power generation. The larger the imbalance, the smaller the relaxation factor has to be chosen.

Figure 4.7 shows the maximum permissible value of Δ to ensure convergence as well as the optimal value to minimize the spectral radius and therefore minimize the number of necessary iteration steps. It features a logarithmic x scale to emphasize the behavior for small values of k_s. For small values of k_s the difference between the optimal and the maximum permissible relaxation factor is getting very small.

Similar behavior was also observed with many real life models, making it extremely hard to properly select an appropriate relaxation factor. If the chosen factor is very small, the convergence rate will be unnecessarily slow, while a factor that is too large will result in no convergence at all.

4.3.3 Adaptive Relaxation

As detailed above, the correct choice of the relaxation factor is crucial. While a thorough analysis of the spectral radius of the precise fixed point iteration can demonstrate the behavior and allow the identification of an optimal relaxation factor, it is a complex and time consuming processes. Conducting such an analysis for real life problems every time they are to be simulated is infeasible in practice.

Because of this, the static fixed point iteration (4.7) has been transformed into an adaptive fixed point iteration that adapts the relaxation factor at every iteration step to ensure convergence while trying to achieve a high convergence rate.

$$\begin{aligned} x'_{t,i+1} &= f'_t\left(x'_{t,i}, x'_{p,i}\right) \\ x'_{p,i+1} &= f'_p\left(x'_{t,i+1}, x'_{p,i}\right) \end{aligned}, \qquad x'_{t,i} \in \mathbb{R}^n, x'_{p,i} \in \mathbb{R}^m, i = 0,1,2,\dots \tag{4.8}$$

with

$$f'_t\left(x'_{t,i}, x'_{p,i}\right) := x'_{t,i} + \Delta_{t,i} \cdot \left(f_t(x'_{p,i}) - x'_{t,i}\right), \qquad \Delta_{t,i} \in (0;1]$$

and f'_p accordingly.

The relaxation factors $\Delta_{t,i}, \Delta_{p,i}, i = 1,2,\dots$ are handled separately for both networks and are adapted during the iteration using the increase factor $C_{\text{inc}} \geq 1.0$ or the decrease factor $C_{\text{dec}} \in (0;1]$ depending on the previous results of the corresponding network.

$$\Delta_{t,i+i} := \begin{cases} \Delta_{t,i} \cdot C_{\text{inc}}, & \left\| x'_{t,i-1} - x'_{t,i} \right\|_2 < \left\| x'_{t,i-2} - x'_{t,i-1} \right\|_2 \\ \Delta_{t,i} \cdot C_{\text{dec}}, & \text{otherwise} \end{cases} \tag{4.9}$$

and $\Delta_{p,i+1}$ accordingly.

Both the increase and decrease factors C_{inc} and C_{dec} are heuristically adapted over time to ensure convergence.

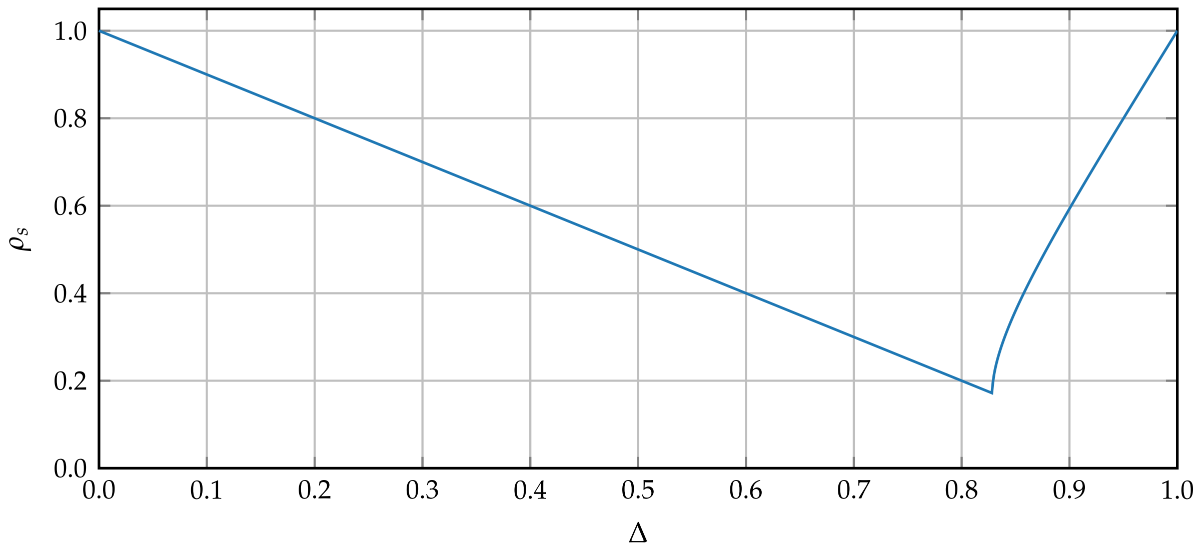


Figure 4.5: 1-duct case study model: spectral radius ρ_s

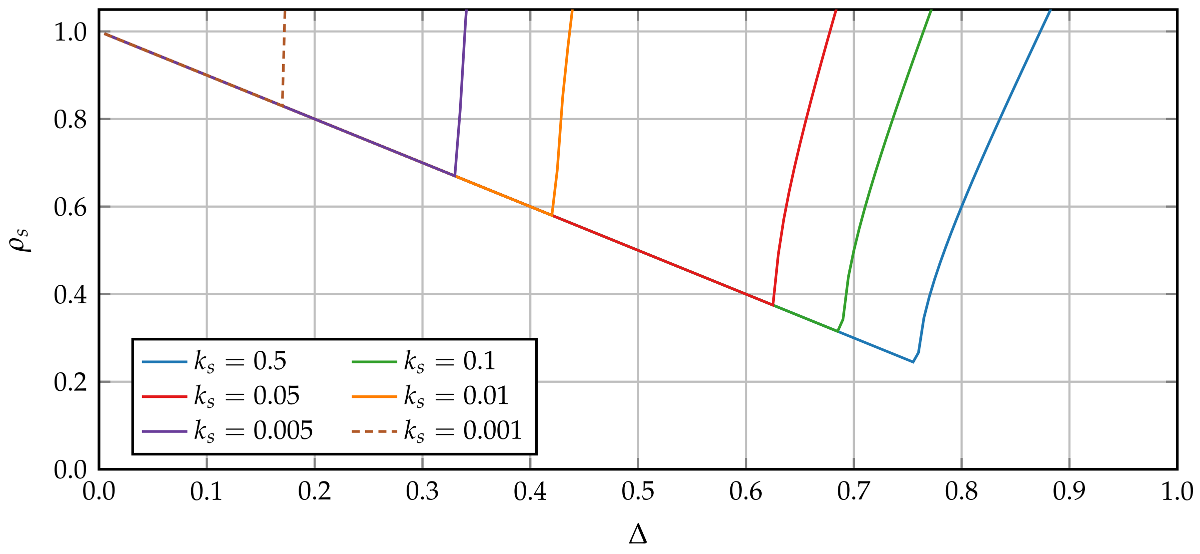


Figure 4.6: 2-duct case study model: spectral radius ρ_s

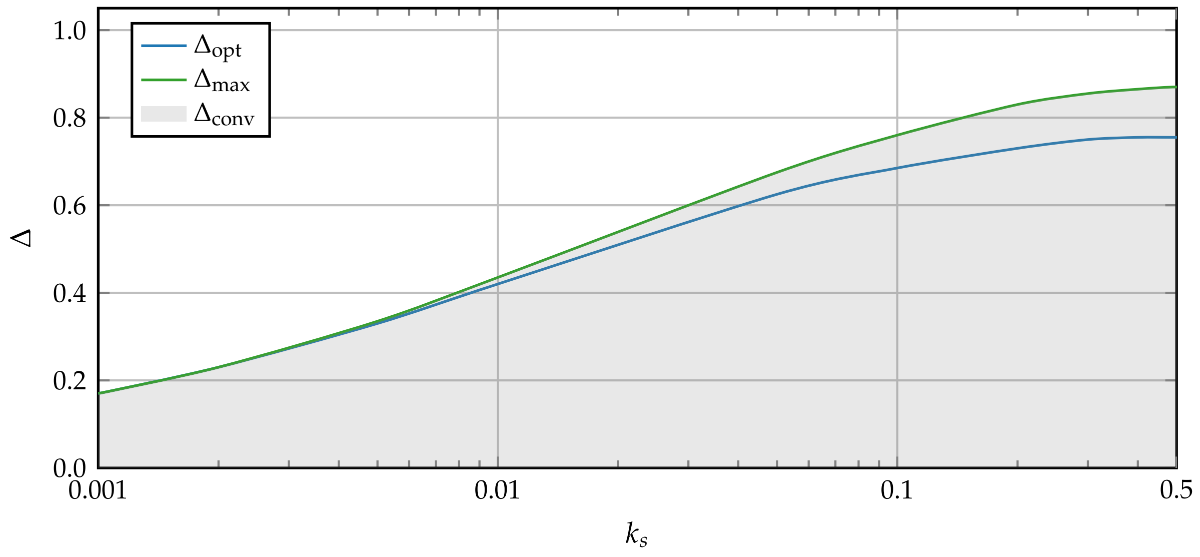


Figure 4.7: 2-duct case study model: optimal and maximum relaxation factor Δ

4.4 Algorithmic Optimization

At first glance, the change to split the networks and solve the problem iteratively as detailed above might seem like a small technical detail, but it results in a considerable increase of overall computational costs.

While the complexity to solve the individual separated thermal and pressure networks is less than with the complete combined one, having to solve them multiple times is still considerably more expensive in total. Because of this, the performance of the core simulation code is of high importance. In this section, a few common algorithmic bottlenecks are investigated and subsequently eliminated.

A set of 441 unique transformer designs consisting of real life as well as theoretical test models provided by ABB is used as baseline and reference benchmark for runtime and speedup measurements. The Westmere-EP system has been used to evaluate and illustrate the efficiency of the optimization techniques devised in the following sections.

4.4.1 Shared Results Caching

As detailed in Section 4.2, a lot of formulas are generated and stored as expression trees within the core of SPICE and need to be evaluated during each step of the Newton-Raphson iteration within SPICE. The method `PTeval_dispatch()` is recursively called to walk these expression trees during which it calls tiny helper functions to perform the actual computation depending on the node type.

These helper functions include `PTdivide()` to divide two numbers or `PTabs()` to compute the absolute value. As depicted in Listing 4.1, these helper functions are called via stored function pointers, which is why even simple mathematical operations like adding two floating point values are handled by a separate function – `PTplus()` in this case.

Figure 4.8 details the share the most relevant functions have in overall runtime and it shows that `PTeval_dispatch()` is responsible for over two-thirds of it. When also considering helper functions, such as `PTplus()`, as well as auxiliary math library functions, the evaluation of the expression trees makes up for almost 90%. The math library functions are called by the helper functions to compute more complex operations, such as `exp.L()` to compute exp() or `log.L()` for log().

Section 4.2.3 showed that during the computation of the derivatives of these formulas, a lot of shared formulas or expressions are generated. Depending on the size and structure of these formulas as well as the number of variables they depend on, the number and size of these shared expressions are significant.

The result of an evaluation of such a shared expression will always be exactly the same, as long as the value of the respective variables does not change. To improve performance, shared expressions should only by evaluated exactly once and the computed result reused whenever possible. However, the original implementation of SPICE does not do that, but does instead naively process every expression irrespective of potentially shared parts.

Implementing the reuse of shared expressions as generated during derivation is straightforward. Every time a subtree is reused during derivative computation the

```
double PTeval_dispatch(INPparseNode* node, double* vals) {
    switch(node->type) {
    case PT_CONSTANT:   return node->constant;

    case PT_VARIABLE:   return vals[node->index];

    case PT_FUNCTION:
        return node->fp1(PTeval_dispatch(node->left, vals));

    default:
        return node->fp2(PTeval_dispatch(node->left,  vals),
                         PTeval_dispatch(node->right, vals));
    }
}

double PTplus(double arg1, double arg2) {
    return arg1 + arg2;
}
```

Listing 4.1: Original implementation of PTeval_dispatch()

appropriate node – and therefore the subtree below it – within the expression tree is marked as shared. While processing the base formula, the result of a shared subtree is stored for later use.

Afterwards, when processing the derivative expression trees, all subtrees that are marked as shared do not need to be traversed again. Instead, the result previously calculated during processing of the base expression can be loaded directly. All subtrees marked as shared are guaranteed to be part of the base expression and are therefore guaranteed to have been processed already.

Figure 4.9 shows the share of the functions in overall runtime after the reuse of results of shared expressions was implemented. `PTeval_dispatch()` is still the dominating method, but overall runtime was reduced significantly by more than 75%, as depicted in Figure 4.14 at the end of this section.

4.4.2 Caches and Advanced Data Structures

Comparing the share of the functions in Figures 4.8 and 4.9 shows that the share of the function `CKTnodName()` increased from below 4% to over 17%. It has to be noted that the absolute time spent by `CKTnodName()` remains unchanged, as it is unaffected by the shared expression reuse. However, the absolute runtime of the expression tree evaluation, i.e. `PTeval_dispatch()` and its helper functions, was significantly reduced, resulting in an increasing share of `CKTnodName()`.

The method is shown in Listing 4.2 and its purpose is to locate an internal data item by its numbered identifier and return its name. As these data items are stored in a linked list, every lookup results in a walk along the list.

Looking up an item within a linked list is of algorithmic complexity $O(n)$, where n is the number of items stored in the list. In this case, every data item is looked up at least once, so overall complexity is quadratic – $O(n^2)$. Also, walking along a linked

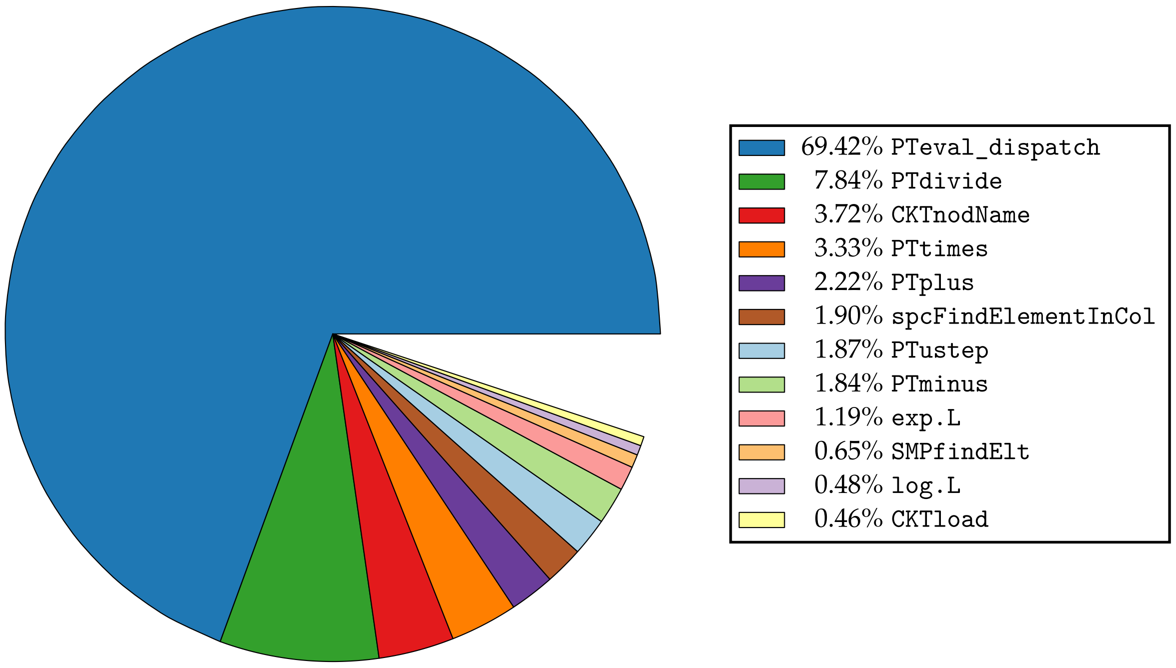


Figure 4.8: Breakdown of functions – original unoptimized code

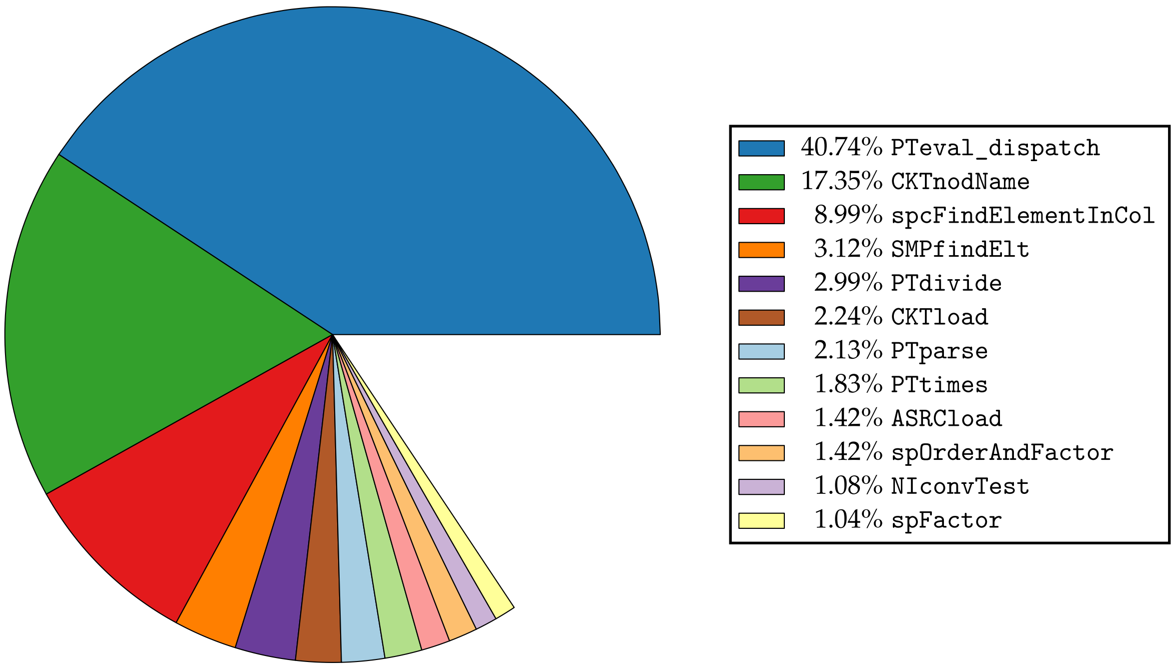


Figure 4.9: Breakdown of functions – code with shared expression reuse

```
char* CKTnodName(CKTcircuit *ckt, int nodenum) {
    CKTnode *node;

    for (node = ckt->CKTnodes; node; node = node->next)
        if (node->number == nodenum)
            return node->name;

    return "UNKOWN NODE";
}
```

Listing 4.2: Original implementation of CKTnodName()

```
char* CKTnodName(CKTcircuit *ckt, int nodenum) {
    CKTnode *node = NULL;

    if ((nodenum >= 0) && (nodenum < ckt->caches.nodeArrLength))
        node = ckt->caches.nodeArr[nodenum];

    return node ? node->name : "UNKOWN NODE";
}
```

Listing 4.3: Optimized implementation of CKTnodName()

list is not very cache efficient, as the individual data items can be scattered across memory. In the worst case, a full cache line has to be transferred from memory for every single list entry. Because of this inefficient lookup and the frequency with which this method is called, `CKTnodName()` comprises a significant amount of overall runtime, even though it is a very small and straightforward function.

The linked list and its contents are constructed at the very beginning of a simulation and are not changed afterwards. Also, the node numbers which are used as identifiers are continuously increasing, starting with zero.

Looking up the appropriate item by number can therefore be implemented by storing all items in an array, indexed by the respective number. This array can be constructed during the initial initialization phase during which the items itself are constructed. The optimized version of `CKTnodName()`, which now has constant $O(1)$ complexity, is presented in Listing 4.3.

Besides `CKTnodName()` several other occurrences of similar bottlenecks were replaced by more efficient arrays or unordered maps, which are part of the C++11 [47] programming language. While an array only supports indexing with natural numbers from 0 to n, an unordered map supports arbitrary keys, such as character strings, for identifying items as long as a function to compute a hash value of a key is provided. Looking up an entry within an unordered map can be implemented to have amortized constant cost, i.e. its complexity is $O(1)$ [28, 22].

Figure 4.10 shows the share of the functions after the optimizations in data structures and caches discussed above have been applied. `CKTnodName()`, which previously accounted for more than 17% of overall runtime, does no longer contribute any significant time to overall runtime. All in all, these optimizations further reduced absolute runtime by almost 20% as illustrated in Figure 4.14.

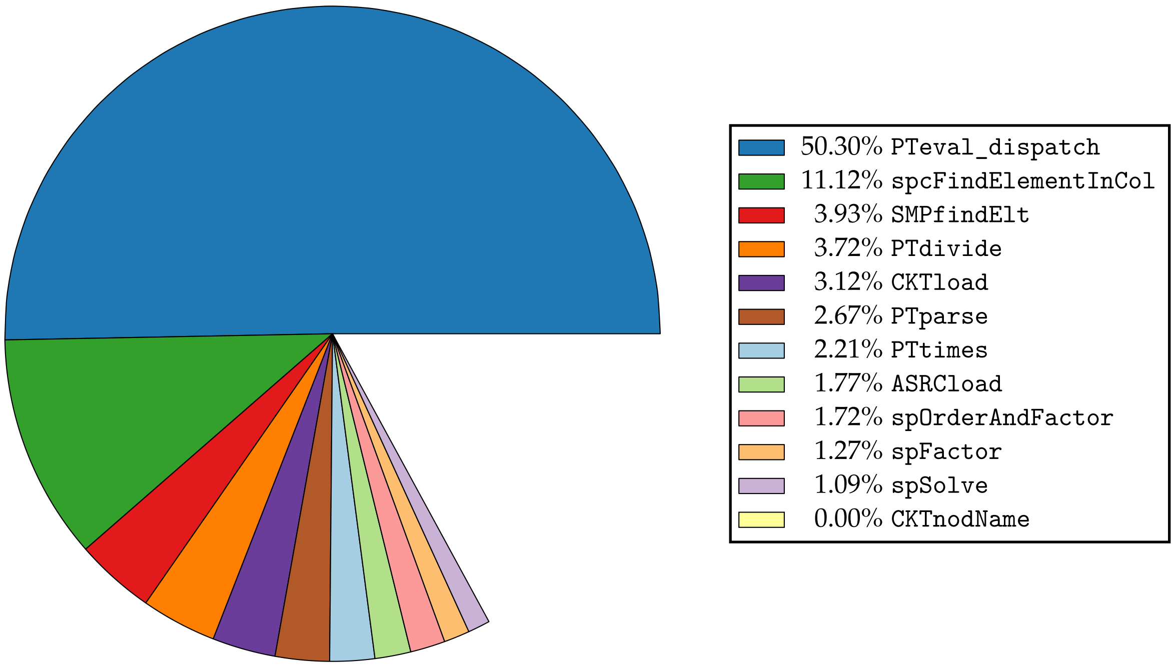


Figure 4.10: Breakdown of functions – code with shared expression reuse and caches

4.4.3 Sparse Matrix

As detailed before, SPICE constructs a nonlinear system of equations which is afterwards solved by Newton-Raphson iteration. The coefficients of this system of equations are stored in a sparse matrix. The matrix entries itself are stored as linked items as shown in Figure 4.11.

Two arrays contain a pointer to the first element of the corresponding row or column, respectively, and each entry has a pointer to the next entry in the same row and column, respectively, as well as its own row and column index. To load an entry $[n, m]$ from row n and column m, the first entry of row n is loaded.

This entry is then used as a starting point to traverse all entries within the same row until the currently processed entry has a column index equal to or greater than m. In the first case, the requested entry is found and can be returned, in the latter case the entry is not stored explicitly and can therefore be considered zero. In principle, the same could be done by traversing all nodes along column m and search for the correct row index.

To account for the various non-linear dependencies, the coefficients within the sparse matrix are updated in every step of the Newton-Raphson iteration. As part of this update, a set of certain matrix rows is processed to handle dependencies between connected nodes. To process an individual row of this set, every existing node is tested by trying to load the appropriate matrix entry within the current row. If it exists, the value is updated, otherwise no further action is necessary.

The update itself is performed by the methods `CKTload()`, `SMPfindElt()` and `spcFindElementInCol()` which together account for more than 18% of total runtime. In the original SPICE implementation, the search for an entry always begins from the first entry of a row and is repeated for every node, resulting in a very large number of traversals along the matrix rows.

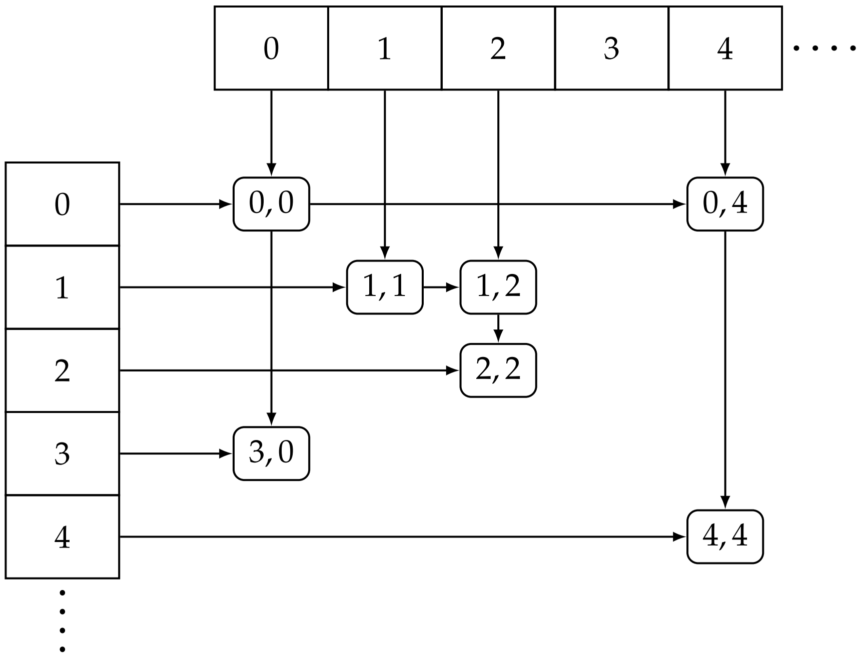


Figure 4.11: Sparse matrix storage in SPICE

The implementation was improved and restructured to extract a full row at a time from the sparse matrix and process it directly, avoiding most of the traversals. This resulted in another absolute runtime improvement of more than 14%. Figure 4.12 shows that after the optimization `CKTload()` only accounts for 0.7% of overall runtime and the time spent in `spcFindElementInCol()` and `SMPfindElt()` is no longer significant.

4.4.4 Improve Function Inlining & Branch Prediction

Due to the implemented optimizations and improvements in data structures and sparse matrix processing, the expression tree evaluation, i.e. `PTeval_dispatch()` and its helper functions, are now again responsible for almost 70% of runtime.

As mentioned before, `PTeval_dispatch()` consists of a single monolithic switch-statement that takes appropriate action depending on the type of the currently processed node. This makes correct branch prediction by the processor very hard, as a single point within the code branches differently for each node type, which can vary strongly when processing large expression trees.

Also, as helper functions like `PTdivide()` are indirectly called via a function pointer stored within the node itself, the compiler is unable to automatically inline the function, i.e. directly integrate the processing logic into the calling function. Because of this, a full function call including setting up the appropriate stack frame has to be done, even though the operation to be performed is something as simple as a summation of two floating point values which could otherwise be realized by a single processor instruction. This generates a big overhead, slowing down processing even further.

To reduce this overhead some of the processing logic within `PTeval_dispatch()` was extracted into their several independent methods. The optimized `PTeval_dispatch()` as well as `PTeval_Plus()`, as a sample of how the extracted methods look like, is shown in Listing 4.4.

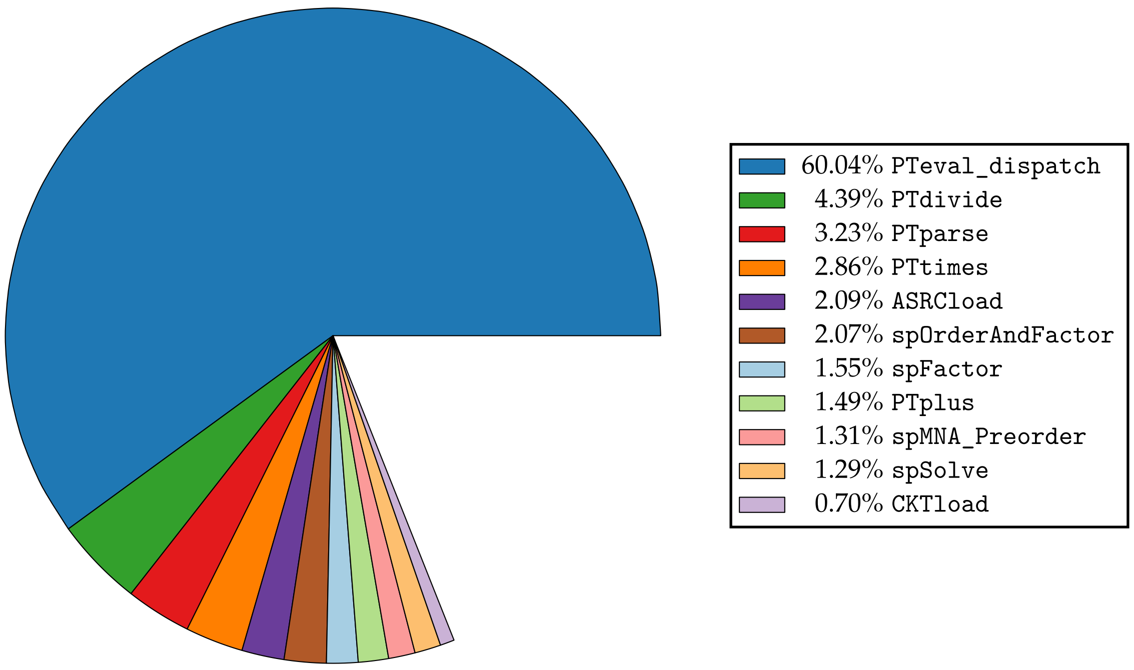


Figure 4.12: Breakdown of functions – code with shared expression reuse, caches and sparse matrix optimization

This restructuring has two advantages: First, it reduces the number of necessary function calls by encouraging function inlining. In theory it is possible to directly inline a recursively called function into itself up to a certain depth, but only few compilers support this by default and typically only within very strict constraints.

Splitting the function makes it easier for the compiler to inline functions into each other. In `PTeval_Plus()` the compiler can inline both calls to `PTeval_dispatch()`, reducing two function calls as well as creating more flexibility to perform automatic optimization, such as instruction reordering.

Additionally, mathematical operations such as $\times$ or $+$ are no longer handled by explicitly calling helper functions like `PTtimes()`. Instead, they are directly processed within the node handling function such as `PTeval_Times()` itself, which also allows further optimization by the compiler. A similar technique was used to directly handle the most commonly used arithmetic functions, i.e. log(), exp(), etc., resulting in further reduction of function calls and improved performance. However, for clarity, this optimization was omitted in Listing 4.4.

The second advantage is improved branch prediction by the processor. As the original implementation only has a single point where the type of the current node is taken into account, it is very hard for the processor to recognize patterns or recurring chains of certain node types.

With the split methods, the processor is able to keep record of the branch destination depending on the current node type, as each node type is handled by its own function. This makes it easier to identify patterns and subsequently correctly predict the taken branch. Such a pattern could be that a multiplication node is often followed by another multiplication node, which is caused by certain recurring expressions within the formulas describing the interaction of elements within the network as well as effects of the differential calculus used to generate the derivatives of the base formulas.

```
double PTeval_dispatch(INPparseNode* node, double* vals) {
    switch(node->type) {
    case PT_CONSTANT:   return node->constant;

    case PT_VARIABLE:   return vals[node->index];

    case PT_FUNCTION:   return PTeval_Func(node, vals);

    case PT_PLUS:       return PTeval_Plus(node, vals);

    case PT_MINUS:      return PTeval_Minus(node, vals);

    case PT_TIMES:      return PTeval_Times(node, vals);

    case PT_DIVIDE:     return PTeval_Divide(node, vals);

    case PT_POWER:      return PTeval_Power(node, vals);
    }
}

double PTeval_Plus(INPparseNode* node, double* vals) {
    return PTeval_dispatch(node->left, vals)
            + PTeval_dispatch(node->right, vals);
}
```

Listing 4.4: Split implementation of PTeval_dispatch()

All in all, for the representative simulation of the above-mentioned 441 transformer models, these optimizations decrease the overall number of branches from 52,932,848,830 to 47,623,837,286 respectively, which is a reduction of about 10%. This change is guaranteed to be a result of the optimization, since the variation for a complete simulation of all models is below 0.01% for both implementations – with and without the optimization.

Additionally, the relative number of branch-misses, i.e. the cases in which the processor predicted the wrong destination, was reduced from about 6.66% of all branches to about 4.10%. At the same time, the variation of the number of branch-misses decreased from ±0.45% to only ±0.05%, which is further indication, that the branch prediction is now more reliable.

On the whole, overall runtime was further reduced by over 20% due to the implementation of this change. Figure 4.13 shows the functions' share in runtime with all previously discussed optimizations applied and illustrates that the methods `PTeval_Times()`, `PTeval_Divide()`, `PTeval_Plus()`, `PTeval_Minus()` and `PTeval_Func()` now handle the work previously done by `PTeval_dispatch()` and its simple evaluation functions like `PTminus()`. Also, `PTeval_dispatch()` no longer shows up on the profile, indicating that it is typically inlined into the calling function as intended.

Figure 4.14 illustrates the absolute runtime required to fully simulate the 441 test models after successively applying the optimization techniques discussed above.

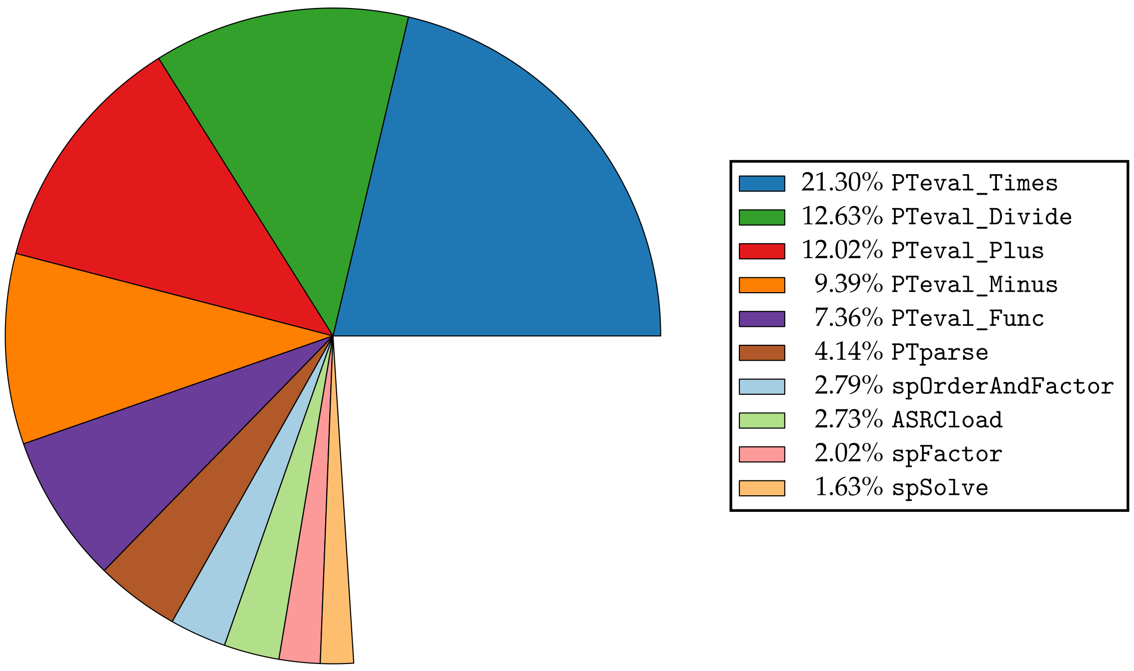


Figure 4.13: Breakdown of functions – code with shared expression reuse, caches, sparse matrix optimization and split evaluation function

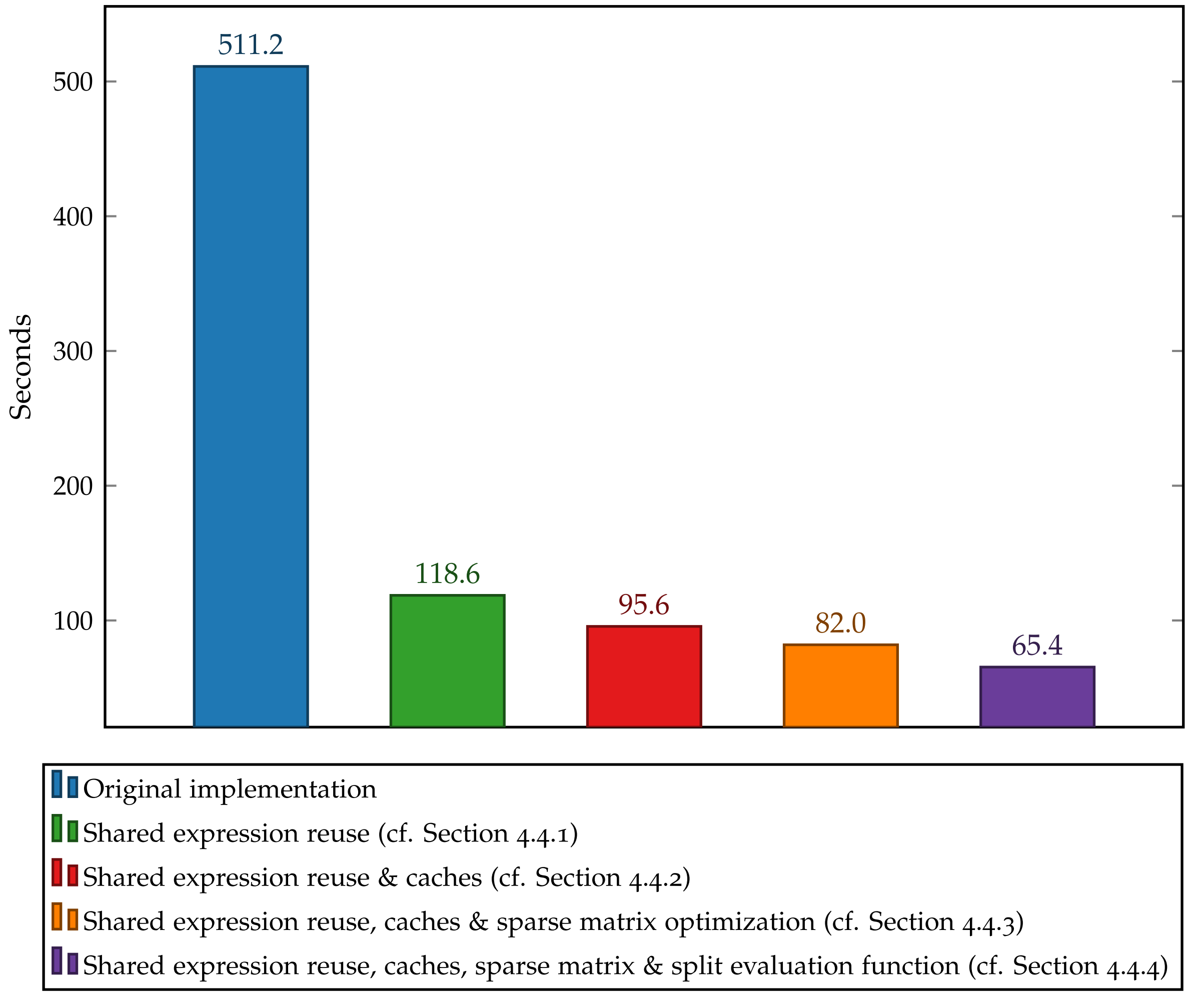


Figure 4.14: Runtime of solver with different optimizations applied

4.5 Dynamic Code Generation

4.5.1 Motivation

The original expression tree evaluation code consists of a single recursive function with a large switch statement to process the individual nodes and their children. Performance analysis shows that this switch statement is the cause for a significant number of branch mispredictions, as the processor's branch predictor is unable to reliably predict the correct branch due to the high variation of node types processed at this single point.

As shown in the previous section, the situation can be improved by splitting the large monolithic recursive function into several smaller ones which handle specific cases and node types. This improves the efficiency of the branch prediction and also makes it easier for the compiler to automatically inline function calls as the split functions are simpler, which additionally reduces some function call overhead.

Still, even with improved branch prediction, processing a node takes some time to determine its specific type and to decide how the node and its potential children should be handled. Also, every time the tree is traversed to one of the children, the relevant node must be loaded from memory.

In practice, there are too many trees and nodes stored in memory during an evaluation to fit all of them into the processor cache. Additionally, every node is only touched once during a single evaluation, which means that during processing the node is most likely not available in the processor cache but has to be loaded from main memory.

4.5.2 Basic Idea

To further speed up the evaluation of the expression trees, a more compact and efficient representation is necessary: One which reduces the scattered memory accesses and at the same time increases the performance of identifying and handling different types of nodes. One possibility to realize this is *dynamic code generation*.

Dynamic code generation and similar methods are an established strategy to obtain faster execution. If some characteristics of the input data is known, the code of an application can be specialized to improve performance, also known as *partial evaluation* [49, 30]. With C++, *templates* can be used to let the compiler generate specialized versions of functions. Modern Java Virtual Machine (JVM) implementations try to automatically detect if a function is executed within the same context and produce specialized code [53] and scripting languages such as Python or JavaScript allow to generate code at runtime.

More generally, generic compiler construction kits which include Just-In-Time (JIT) compiler components such as *LLVM* [56] can be used for specialization. However, code generation itself is typically not highly tuned so the generated code has to be run very frequently to show benefits. To reduce generation time, one can pre-compile skeletons from a given algorithm, and specialize by patching the skeletons with input data, as shown by Weidendorfer et al. [81].

Code generation techniques are also used to automatically tune code for specific architectures, e.g. for linear algebra kernels (ATLAS [82]), sparse matrices (OSKI [83]), or FFTs (FFTW [29]). As these libraries are typically reused many times after installation, they perform their autotuning phase during installation/building time, which means that generation speed is not critical. All in all, a good survey on Just-In-Time techniques can be found in Aycock [4] or Franz [27].

In the exemplary case considered in this chapter, code generation can be used to reduce the number of necessary tree traversals. Instead of fully traversing the tree and examine every node for every evaluation, it is traversed only once during code generation. Afterwards, the generated code can be executed without the need for additional tree traversals.

As presented in Müller et al. [63], a type specific code is generated for each node that produces the same result as an ordinary evaluation would. Since the type of the node is directly encoded, it is no longer necessary to explicitly evaluate its type and then conditionally evaluate the node and potentially its children.

The effect is a single instruction stream specifically created for a particular expression tree which does not need to include any conditional handling depending on node types and therefore contains only a very limited amount of branches. Also, as the node type and the function it represents are directly encoded into the instruction stream, it is no longer necessary to load the nodes from memory. Instead, only a single compact and continuous instruction stream needs to be streamed from memory and processed.

4.5.3 Implementation

Bytecode

The first step was to write a bytecode compiler and interpreter. This allows to serialize the expression tree evaluation in an architecture independent way without getting involved in hardware architecture and instructions and thus was an obvious way to start.

To avoid the potential overhead of a more general solution, it was decided to not use any existing bytecode specifications or implementations, such as the Java Virtual Machine (JVM). Instead, a newly designed bytecode, specifically tailored to the task at hand was created.

During bytecode compilation the expression tree is processed in post-order and depending on the type of the node a pointer to an appropriate evaluation function is added to the bytecode stream. For every type of node there exists a specific evaluation function, which takes care of the relevant operations, such as calling a unary or binary function or loading a constant value or variable. Constant floating point values and indices for variables are embedded directly into the bytecode stream. After processing the last node, a pointer to a special STOP function is appended to mark the end of the code. The bytecode generated to evaluate Equation (4.1) from above is depicted in Figure 4.15.

During execution, the bytecode interpreter works exactly like a stack machine [71]: The pointer to the next evaluation function is loaded from the bytecode stream and the function is executed. The evaluation function then loads optional arguments like constant values or indices of variables from the bytecode stream and mimics the processing of the appropriate node type it is responsible for. Constant values or

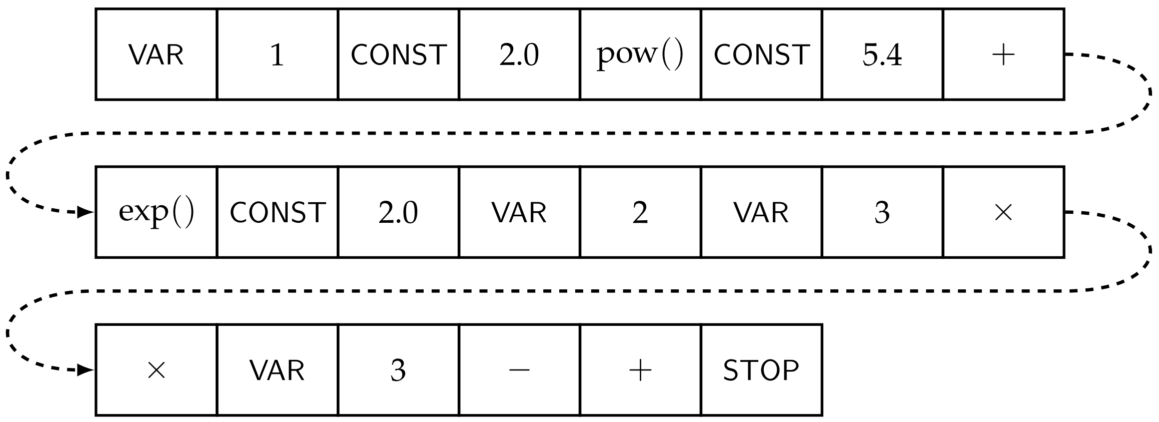


Figure 4.15: Bytecode representation of Equation (4.1)

variables are pushed onto the stack, while unary and binary functions first pop their arguments from the stack and then push the result of the operation back onto it.

Finally, at the end of the bytecode stream, the special STOP function is executed and the single remaining value on the stack is returned as result of the execution. The post-order processing during bytecode compilation guarantees that the necessary values are located at the top of the stack when needed, similar to reverse polish notation [17].

While the bytecode interpreter does not need to explicitly detect the type of the node, it still relies on a set of evaluation functions for handling the different types of nodes. If the evaluation function was called like any other function, it would again result in a recursion with many function calls.

To avoid this overhead, it is necessary to employ a technique called *tail call optimization* as detailed in Steele [74]. A tail call is a function call to `funB()` that happens as the last action at the end of another function `funA()` and if `funB()` returns a value, that value has to be returned by `funA()` unmodified.

Since `funA()` has reached its end, its stack frame is no longer needed and can be replaced by the stack frame of `funB()`. When tail call optimization is employed, the current stack frame of `funA()` is directly modified to match the stack frame required for `funB()`, and a direct jump into `funB()` is used to start executing it, instead of executing an actual call to `funB()`. When `funB()` returns, the calling function of `funA()` receives the return value of `funB()` as if `funA()` had forwarded it.

Using this technique, the bytecode interpreter can efficiently execute the generated bytecode stream and significantly reduce the number of costly memory references (because the nodes no longer need to be loaded) and necessary function calls.

Figure 4.16 at the end of this section shows a comparison between the original evaluation code and the bytecode compiler/interpreter implementation. The compilation process itself is extremely fast, requiring only about 600 milliseconds of runtime during processing of all 441 test models. At the same time, the evaluation of the expression trees is improved by more than 10s or about 30%, respectively.

Even though the bytecode implementation does indeed improve evaluation performance, analysis shows that there is still a considerable amount of time spent in moving data to and from stack and fetching function pointers and parameters from the bytecode stream. To reduce this even further, the next step was to generate real x86 machine code that can be directly executed by the processor.

LLVM

The LLVM project is a well-known "collection of modular and reusable compiler and toolchain technologies". Using its libraries one can construct functions and compile them into directly executable machine code, i.e. the LLVM libraries return a function pointer that can be called like any other compiler generated function.

The LLVM library provides interfaces for calling additional functions, loading constant values, referencing variables in memory, executing basic instructions on it and more. To construct a function, one specifies the return type, number and type of arguments and then adds instructions.

When processing an expression tree or its derivatives to generate executable code to evaluate it, simple operations like $+$, $-$, ... and functions like $sqrt()$ that are directly supported by the processor are directly added as a single instruction. More complex functions, such as $exp()$ or $pow()$, are processed by calling into compiler generated helper functions.

Even though the instructions are named like assembler instructions, everything takes place at a high and abstracted level. LLVM automatically takes care of data alignment, register allocation and calling conventions, which depend on the operating system and specify how parameters are passed to a called function and how the system stack is managed. LLVM also provides options to automatically optimize the generated function like an ordinary compiler would do when compiling source code.

LLVM is an open source project with many abstraction layers and dependencies and consists of many lines of code. On the one hand, this gets visible by the sheer size of the compiled libraries, which are several megabytes in size. On the other hand, the complexity and abstractions also result in a huge internal overhead during function construction and compilation at runtime. Even though function optimization was disabled, the fastest register allocator was selected and the structure of the generated code was designed to benefit the compilation process, the time needed for compilation is still enormous as illustrated by Figure 4.16 at the end of Section 4.5.

Generating executable code to process those trees using LLVM is more than 5 times slower than directly evaluating the trees using the original code. However, at the same time executing the generated code to evaluate the trees is more than 5, respectively 3.5 times faster than the original code or the bytecode interpreter.

AsmJit

As the execution speed of the code generated by the LLVM libraries is very promising, the compilation phase needs to be sped up. For this a different, slimmer code generation library called *AsmJit* [52] was used. AsmJit supports two different modes of operation – compiler and assembler.

When using the compiler mode, AsmJit behaves similarly to LLVM, but much more restricted and less abstracted. Register allocation is done using a linear-scan register allocator, which is especially suited for dynamic code generation as in practice it provides a good balance between processing time and allocation efficiency, as shown by Poletto and Sarkar [68].

Calling conventions are also handled automatically in this mode, but contrary to LLVM AsmJit does not offer any kind of automatic optimization. On the other hand, it does not have as many abstraction layers as LLVM and is therefore much faster when generating code, as illustrated in Figure 4.16.

Similar to the bytecode implementation constant values are embedded into the instruction stream. However, to avoid jumps during execution, the constant values are grouped together and appended as a single block after the return instruction. That way the code and constant values are close together and self-contained. Also, constant values that are used more than once need only be embedded a single time and can then be referenced multiple times at different points within the instruction stream. This compacts the code even further.

In assembler mode, the appropriate calling conventions for procedure calls have to be handled explicitly. To keep the code manageable, only the x86-64[3] calling conventions for Windows and Linux have been implemented [59, 57]. Using the 64-bit mode also enables the use of overall 16 vector registers as opposed to only 8 available in 32-bit mode.

Register allocation also needs to be handled explicitly in this mode. Most generic register allocation techniques require an additional processing phase at the end of a function to analyze how often values are used and where they have to be saved to and restored from memory.

Contrary to this, to keep code generation time low, a very straightforward and direct register allocation technique that directly assigns registers on-the-fly during code generation is used here. As always, the expression tree is processed in post-order and, as was the case with the bytecode implementation, the stack could be used to push and pop values – constants, variables, results of instructions – during execution as needed. However, to avoid unnecessary stack and therefore memory access, the processor's vector registers are used as a cache for the top of the stack.

As long as the number of values on the stack is smaller than that of the available vector registers no memory access is involved at all. Once the number of values on the stack grows beyond the number of registers, the lowermost values are moved from the registers to memory to free the registers. Moving a value that is still needed later on from a register to memory because the register is required for another value is called a register *spill*.

When a value is popped from stack, the respective registers are initially kept empty in case new values are again pushed onto the stack. When a value that was transferred to memory is popped, it is first moved back from memory into the appropriate register for processing. It should be noted though, that values that are spilled to memory are typically only moved into fast processor caches.

This register allocation technique tries to keep memory accesses at a minimum and retains the most current values within the registers to be able to process them as fast as possible. As it does not depend on code generated later, but only on the current size of the stack, it can be performed immediately during code generation, avoiding a separate register allocation phase.

The efficiency of a register allocation can be determined by the number of register spills occurring during execution of a function. When simulating all 441 test models, the *cached stack approach* detailed above generates almost 25% less register spills than the linear-scan register allocator implemented in the AsmJit compiler mode.

[3]Also called AMD64 or Intel64

However, as those memory transfers are mostly only to and from processor cache, it does not make a noticeable difference in this particular application. Nevertheless, it is a strong indication that this straightforward implementation is more than sufficient in this special case and that more complex algorithms are not likely to improve execution performance sufficiently to compensate for the increased generation time.

As depicted in Figure 4.16 the code generation phase is significantly faster than LLVM for both versions using AsmJit. Still, the additional abstraction layer at the compiler compared to the assembler mode slows down the compilation process noticeably, albeit not as much as with LLVM.

Execution time of the generated code is virtually identically for both modes of operation and even slightly faster than LLVM, as no compromises on the structure of the generated code was necessary to speed up the compilation process.

Finally, Listing 4.5 shows an annotated assembler representation of the actual generated code to evaluate Equation (4.1) including its derivatives. In this particular case, the expression trees are small enough to keep all intermediate results within the registers and completely avoid to spill registers to the stack.

```
; prolog (save registers, set up stack)
push   %r15                      # save r15 on stack
push   %rbx                      # save rbx on stack
push   %rbp                      # save rbp on stack
mov    %rsp,%rbp                 # set up frame pointer
sub    $0x320,%rsp               # allocate space on stack
; process function arguments
mov    %rdi,%r15                 # store pointer to variables
mov    %rsi,%rbx                 # store pointer to result of derivatives
; process formula
movsd  (%r15),%xmm2              # load variable x_1
movsd  0xef(%rip),%xmm3          # load embedded constant 2.0
movsd  %xmm2,%xmm0               # prepare first argument for helper function
movsd  %xmm3,%xmm1               # prepare second argument for helper function
movabs $0x406890,%rdx            # load pointer to helper function pow()
callq *%rdx                      # call helper function
movsd  %xmm0,%xmm2               # process result from helper function
movsd  0xd7(%rip),%xmm3          # load embedded constant 5.4
addsd  %xmm3,%xmm2
movsd  %xmm2,%xmm0               # load argument for helper function
movabs $0x406a30,%rdx            # load pointer to helper function exp()
callq *%rdx                      # call helper function
movsd  %xmm0,%xmm2               # process result from helper function
movabs $0x7f8e11ecc158,%rdx      # load pointer to cached shared result
movsd  %xmm2,(%rdx)              # cache result of shared expression
movsd  0xa1(%rip),%xmm3          # load embedded constant 2.0
movsd  0x8(%r15),%xmm4           # load variable x_2
movsd  0x10(%r15),%xmm5          # load variable x_3
mulsd  %xmm5,%xmm4
mulsd  %xmm4,%xmm3
movsd  0x10(%r15),%xmm4          # load variable x_3
```

```
subsd  %xmm4,%xmm3
addsd  %xmm3,%xmm2
; process first derivative
movabs $0x7f8e11ecc158,%rdx      # load pointer to cached shared result
movsd  (%rdx),%xmm3              # load result of shared expression
movsd  0x69(%rip),%xmm4          # load embedded constant 2.0
movsd  (%r15),%xmm5              # load variable x_1
mulsd  %xmm5,%xmm4
mulsd  %xmm4,%xmm3
movsd  %xmm3,(%rbx)              # store result of first derivative
; process second derivative
movsd  0x50(%rip),%xmm3          # load embedded constant 2.0
movsd  0x10(%r15),%xmm4          # load variable x_3
mulsd  %xmm4,%xmm3
movsd  %xmm3,0x8(%rbx)           # store result of second derivative
; process third derivative
movsd  0x39(%rip),%xmm3          # load embedded constant 2.0
movsd  0x8(%r15),%xmm4           # load variable x_2
mulsd  %xmm4,%xmm3
movsd  0x37(%rip),%xmm4          # load embedded constant 1.0
subsd  %xmm4,%xmm3
movsd  %xmm3,0x10(%rbx)          # store result of third derivative
; finalize formula
movsd  %xmm2,%xmm0               # prepare result of formula for return
; epilog (clean up stack, restore saved registers)
add    $0x320,%rsp               # free space on stack
pop    %rbp                      # restore rbp from stack
pop    %rbx                      # restore rbx from stack
pop    %r15                      # restore r15 from stack
retq                             # return
...
embedded constants
...
```

Listing 4.5: Annotated assembler representation of code generated to evaluate Equation (4.1) and its derivatives

codegen

To further reduce compilation time, the next step was to directly create executable machine code without using any generic libraries or abstraction layers. By optimizing and structuring the code generation to the exact needs at hand and only supporting and implementing instructions that are actually required, a great portion of special cases and especially addressing modes can be ignored. The resulting code generation framework is less generic but very tight and compact, and further reduces overhead.

Handling of register allocation and calling convention is done exactly as in the AsmJit assembler mode implementation, but instead of calling into a library to generated x86 machine instructions, the appropriate opcodes are constructed directly and added to the instruction stream according to the *Intel 64 and IA-32 Architectures Software Developer's Manual* [44].

Since it is not known beforehand how large the generated code will be, it is first generated into local buffers. Afterwards, the generated instruction stream is moved to a properly aligned and consecutive memory area which is then marked as executable. A pointer to the beginning of the instruction stream can then be called like an ordinary compiler generated function.

As intended, compilation time was further reduced to less than half the time needed by AsmJit in assembler mode for all 441 test models, while the execution time of the generated code is virtually identical to the one generated by the AsmJit implementations. All in all, using codegen the evaluation of the expression trees including compilation was reduced to 8.6s, which is a reduction of more than 77% of the 37.5s needed when using the original implementation.

Figure 4.16 shows a comparison of the individual runtimes of all different implementations covered so far. It should be noted that the depicted total runtime represents the time required to perform the full simulation of the 441 test models. This also includes the generation of the expression trees at the beginning, modification of various internal data structures like the sparse matrix to update the coefficients of the equation system, etc. Because of this, the total runtime is noticeably longer than the sum of the time required for the compilation and evaluation phases.

4.5.4 Intel Xeon Phi Porting

To be able to evaluate the performance and behavior of the simulation solver and the expression tree evaluation in particular on the Intel Xeon Phi, the application and code generation needs to be ported. As the original expression tree evaluation code as well as the bytecode compiler/interpreter does not generate architecture specific code, they can be compiled directly for execution on the Xeon Phi.

Of the three frameworks to dynamically generate executable machine code, LLVM, AsmJit and codegen, only the last one has been selected to be ported. Codegen is the most compact and manageable implementation among those three and the fastest one at the same time.

Because the architecture of the Xeon Phi is x86 based, porting the code generation was possible without much effort. All basic instructions regarding the general purpose registers R01-R15 are encoded in exactly the same way, which means that large parts of the code generation framework can be reused as-is. Moreover, because the Xeon Phi runs Linux, the calling convention is also identical and the corresponding code generation can be reused.

All in all, this leaves some alignment issues and the handling of floating point values within the vector registers. To accommodate the increased number and size of the vector registers, the Xeon Phi uses a newly designed instruction set for floating point data [40]. As it does not provide compatibility instructions, the code generation framework must be adapted to the new instruction set. Register allocation is still done the same way, except that the number of registers available to cache values is doubled. Consequently, initial porting of the code generation framework to Xeon Phi was rather straightforward.

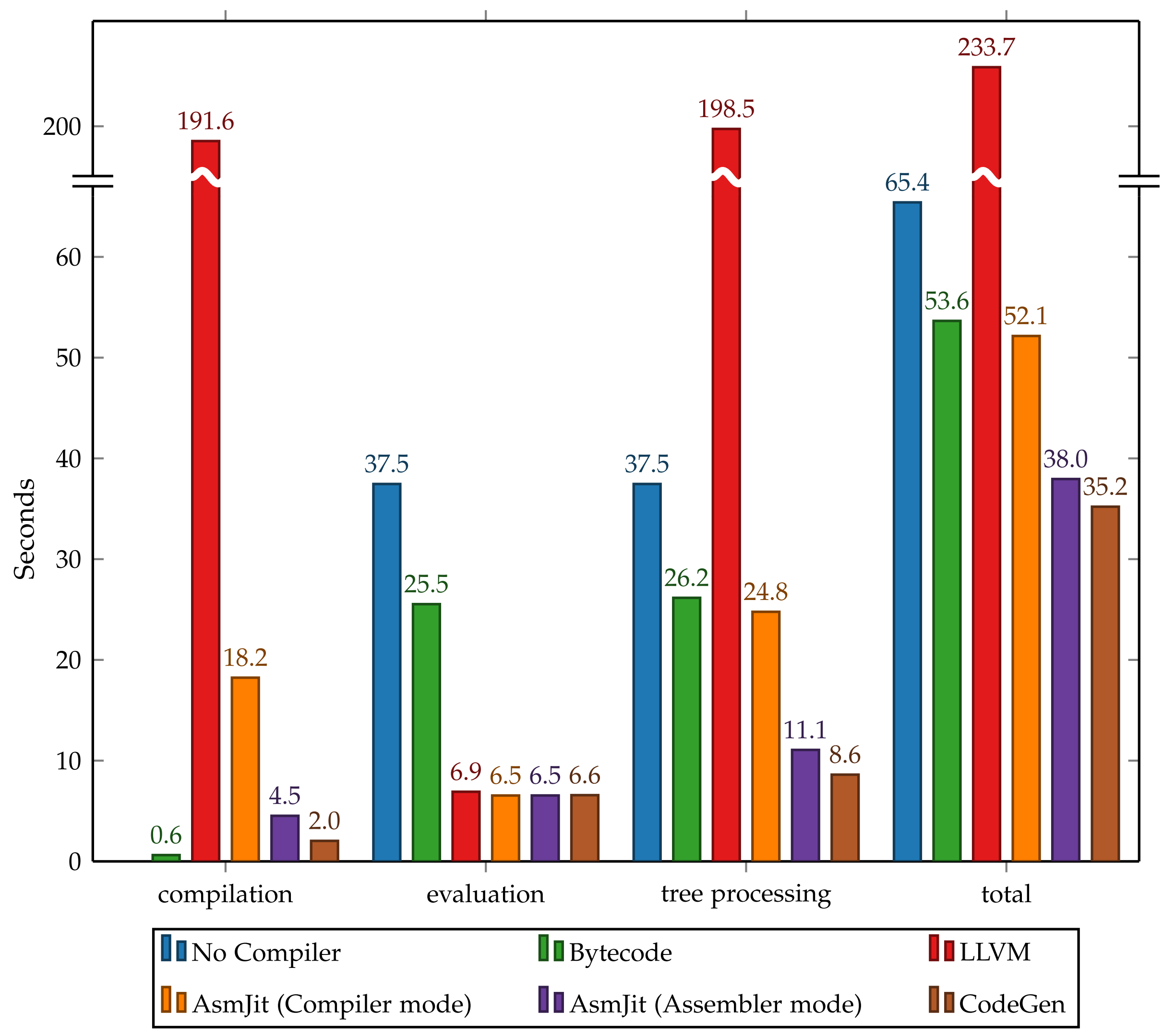


Figure 4.16: Comparison of sequential code generation implementations on Westmere-EP

As the new instruction set supports three operand instructions – two source and one destination operand – and write masks, the opcode encoding is more complex and voluminous resulting in larger generated code, as will be seen in Section 4.6.4.

Most floating point instructions have a correspondent version in both instruction sets, with the notable exception of floating point division and square root computation. Hence, these computations are forwarded to compiler generated helper functions similar to other functions without explicit floating point instruction support before.

4.6 Parallelization

The next step after improving the sequential performance of the thermal simulation solver is parallelization. Except for the LLVM implementation – which was ignored because of its extremely slow compilation process – both compilation phase as well as execution phase of all previously discussed code generation implementations were parallelized at thread level using OpenMP.

Since every formula and its derivatives are self-contained, parallelization was implemented by processing different expression trees with their respective derivatives at the same time. During compilation, the only necessary synchronization between different threads is for memory allocation. Everything else can be done completely independent from other threads.

The only interaction with shared memory during execution is to read variables and store the result of the evaluation in a memory area specific for every formula and its derivatives. Therefore no synchronization is necessary at this point. The same holds for the original code, which can also be parallelized in a straightforward way.

4.6.1 Simulation Solver

Figure 4.17 shows both runtime and speedup of the simulation solver, when executed on the Westmere-EP system using the expression tree evaluation implementations introduced above. All code generation implementations show virtually the same principal behavior, as they all evaluate the expression trees by doing a single traversal during computation and mostly process a continuous stream of executable code – bytecode or machine code respectively – afterwards.

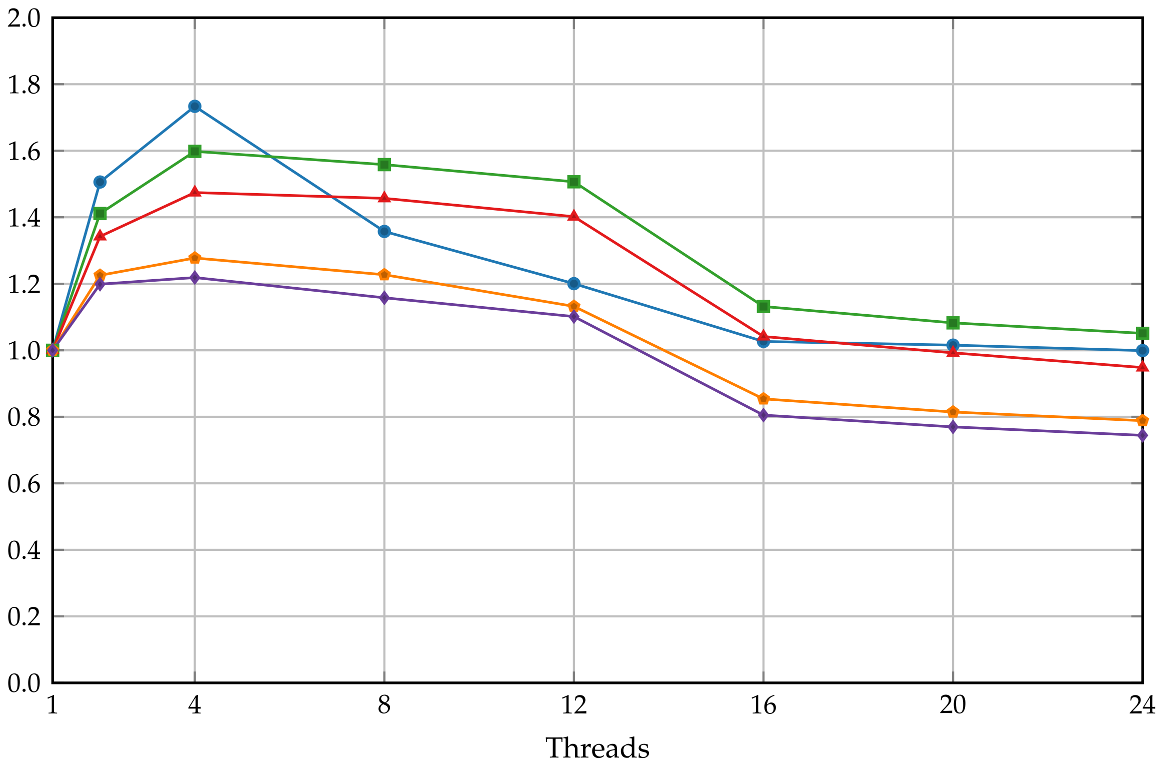
All in all, however, the application as a whole does not scale very well, especially when using more threads than physically available cores, i.e. when using Hyper-Threading. Additionally, comparing Figures 4.16 and 4.17 shows that the individual implementations achieve a lower speedup if their respective sequential performance is higher. The more efficient the evaluation of the expression trees, the more other parts of the application start to dominate, resulting in a reduction of the parallelized fraction of the application and subsequently reduced parallel efficiency. Figure 4.16 shows that evaluation using the original code consumes more than 55% of overall runtime. When using codegen, however, expression tree evaluation only accounts for less than 25%.

The sequential part of an application limits the maximum possible speedup, as it always requires the same amount of time, regardless of the number of threads that are used to process the parallelized portion. For a given percentage of sequential processing within an application, it is possible to estimate an upper bound for the achievable speedup when using multiple threads. This observation was formalized by Gene Amdahl [3] and is commonly known as *Amdahl's Law*.

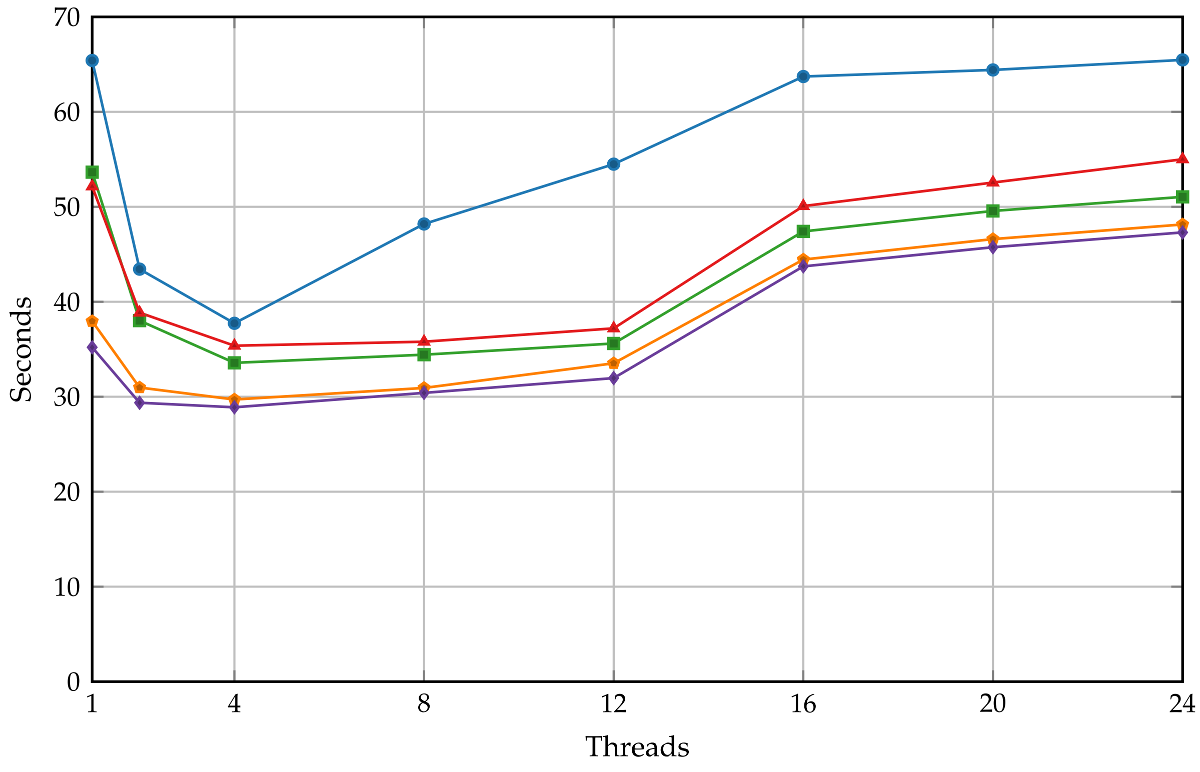
According to this, even when using four threads for parallelized tree processing the overall speedup of the application is limited to 1.75 and 1.22 for the original (0.57% sequential) respectively the codegen (0.24% sequential) implementation.

Furthermore, the maximum attainable speedup using an infinite number of threads is 2.32 for the original and 1.31 for the codegen implementation. In that case only an infinitely small amount of time is required to process the parallel part of the application, resulting in the same overall runtime for both implementations – the time required to execute the sequential part.

When comparing this to the results in Figure 4.17, the achieved speedup using four threads roughly matches the theoretical upper bound, indicating a good parallel efficiency when using only a limited number of threads. When further increasing the number of threads though, parallel efficiency decreases significantly and the application actually slows down again.



(a) Speedup



(b) Runtime

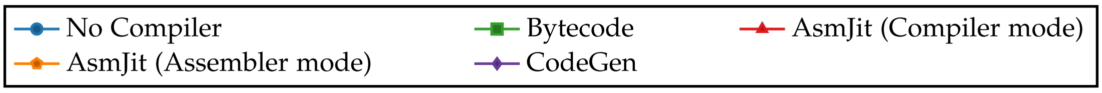


Figure 4.17: Runtime and speedup of the simulation solver on Westmere-EP

4.6.2 Benchmark Application

The fraction of runtime consumed by the evaluation of expression trees can change depending on the complexity and detailedness of the simulated transformer or the number of iterations necessary for convergence. Furthermore, evaluation of expression trees is generally applicable within other applications as well, so it is still interesting to known how it scales in itself.

A specifically developed benchmark application is used to investigate this more detailed. This application randomly generates expression trees and differentiates them as the simulation software would do. To ensure that the generated trees are comparable to those that are generated by the transformer simulation software the individual node types are selected with roughly the same relative frequency that was observed in real life. The random number generator used is part of the C++11 language standard [47] and is initialized to the same state at every start of the benchmark application to guarantee that every execution of the benchmark and every code generation implementation processes exactly the same expression trees.

The initially generated expression trees are targeted to have a uniform depth of about 14, i.e. every leaf node typically has a distance of 14 to the root node. A total of 1200 trees are generated and their first-order partial derivatives is computed. The default number of variables is 50, which means that for every generated formula 50 derivatives are generated, some of which will be 0, however, if the relevant variable did not occur in the formula. Finally, all generated expression trees and their derivatives are evaluated 250 times to simulate an iterative procedure.

Unless otherwise noted, all performance numbers and results from now on refer to this benchmark application instead of the full-fledged original simulation solver.

4.6.3 Parallel Performance

Westmere-EP

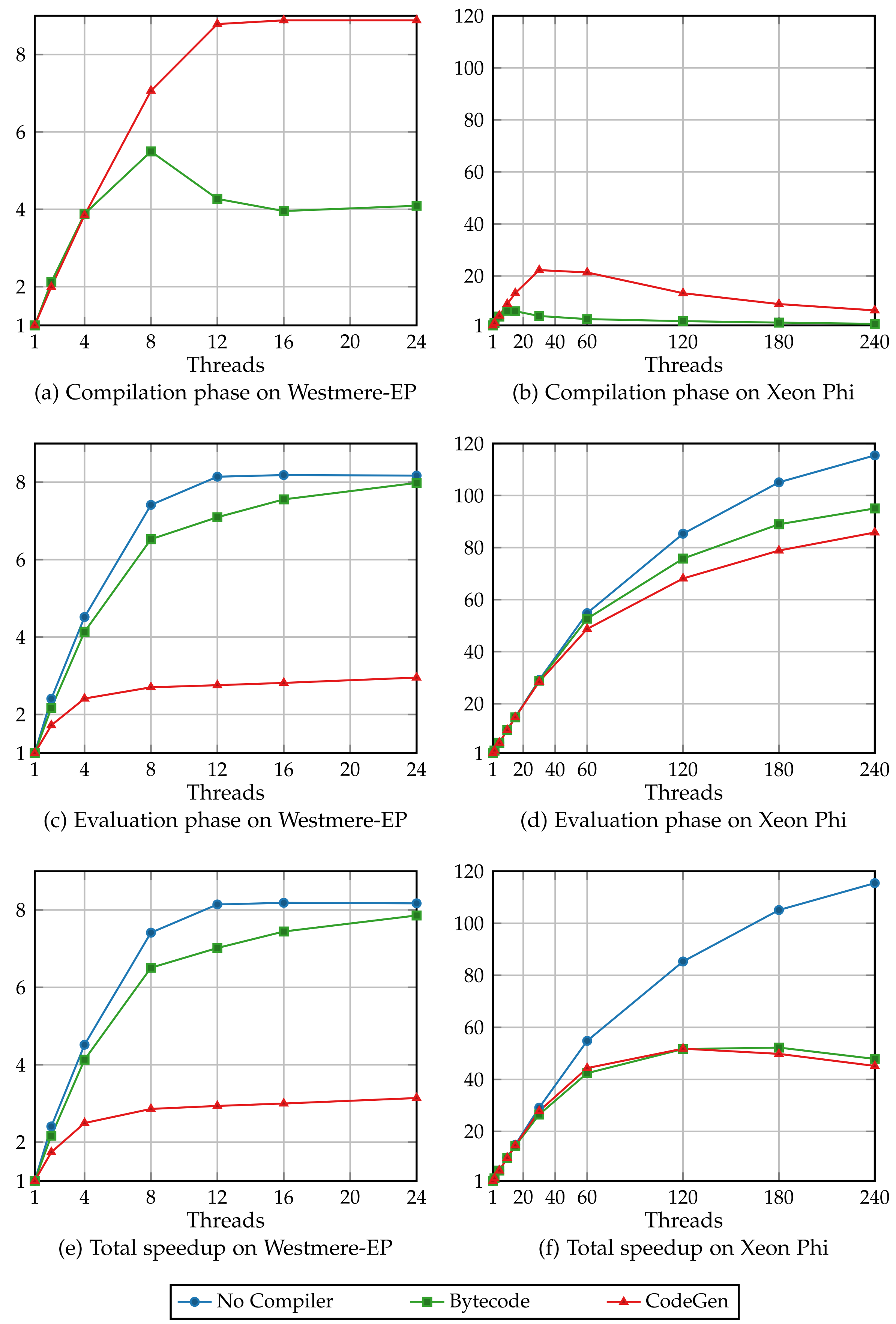
The results of the investigation of the benchmark application is depicted in Figure 4.18e. The original code and the bytecode implementation scale well up to about 8 threads and even show super linear speedup up to 4 threads due to cache effects.

The bytecode implementation continues to improve slowly up to 24 threads, while the original code no longer shows any improvement when using more than 12 threads. Both AsmJit versions and the codegen implementation improve up to about 8 threads, but stagnate afterwards.

In the end, as its sequential version is much more efficient, the codegen implementation is still more than three times faster than the original code.

Xeon Phi

Contrary to the observations on the Westmere-EP system, the performance of the original expression tree evaluation code increases noticeably up to the available 240 threads when executed on the Xeon Phi. At the same time, the bytecode and codegen implementations achieve their best results at 180 and 120 threads, respectively, and



(a) Compilation phase on Westmere-EP

(b) Compilation phase on Xeon Phi

(c) Evaluation phase on Westmere-EP

(d) Evaluation phase on Xeon Phi

(e) Total speedup on Westmere-EP

(f) Total speedup on Xeon Phi

Figure 4.18: Speedup of compilation and evaluation phases of the benchmark application

start to decline afterwards, as the threads now have to increasingly compete over resources within a physical core.

Comparing the speedup for the different phases – compilation and evaluation – on Xeon Phi in Figure 4.18 shows that the evaluation phase of the individual implementations continue to improve up to 240 threads, but are slowed down by declining performance of the compilation phase.

The much better speedup due to Hyper-Threading on Xeon Phi compared to Westmere-EP is due to the fact that a single core of Xeon Phi is much simpler structured and executes instructions in-order. When the code needs to load data from memory, a core of the Westmere system can often automatically rearrange instructions to continue processing while waiting for data to arrive from memory.

The in-order execution of a Xeon Phi core on the other hand waits and idles until the requested data is available. During this waiting time, the other logical Hyper-Threading cores can use the shared resources within the physical core.

All in all, every implementation – most notably the original evaluation code, which even surpasses the bytecode implementation – achieves better results on Xeon Phi than on Westmere-EP as shown in Table 4.2.

Again, even though the other implementations show better speedup, codegen is still the fastest implementation. It should be noted, that there is yet no optimization for the specific architecture and hardware characteristics, but only a straightforward port to the different instruction set. Further improvement should well be possible by incorporating specific hardware characteristics into code generation, such as strategically placed prefetch instructions.

4.6.4 Code Streaming Performance

Depending on the size of the relevant expression tree, the executable code generated to evaluate it can get quite large, possibly consuming several hundred megabytes.

Table 4.3 shows the amount of memory that is needed to store the generated code to evaluate the expression trees within the benchmark application. It also highlights that the code generated for the Xeon Phi is slightly larger. Because of this, it is important to verify if streaming the code to the processor cores is in itself a bottleneck.

To do so, the generated code was patched in two different variants. The first variant replaced the complete generated code with *nop* instructions, which is effectively a dummy or placeholder instruction that does nothing. This results in code that is the same size as the real code, but has no memory accesses besides the transfer of instructions to the processor cores and no floating point computation.

The second variant kept the basic structure of the code but replaced all instructions that access memory with a similar one that only operates on registers. This mimics the original code with respect to floating point computations, but still avoids explicit memory references.

Figure 4.19 demonstrates that both variants are virtually identical on Westmere-EP for all number of threads and runtime improvement stops at 12 threads and does not benefit from Hyper-Threading. On Xeon Phi both variants are very close up to 30 threads at which point the runtime improvement of the second variant starts to slow down while the first variant continues to improve at the same rate up to 120 threads.

	Westmere-EP	Xeon Phi
original code	34.63	10.64
bytecode	12.71	11.24
codegen	11.51	8.11

Table 4.2: Total best runtimes in seconds

	Westmere-EP	Xeon Phi
average	368 KiB	413 KiB
maximum	702 KiB	786 KiB
total	431 MiB	484 MiB

Table 4.3: Generated amount of executable code

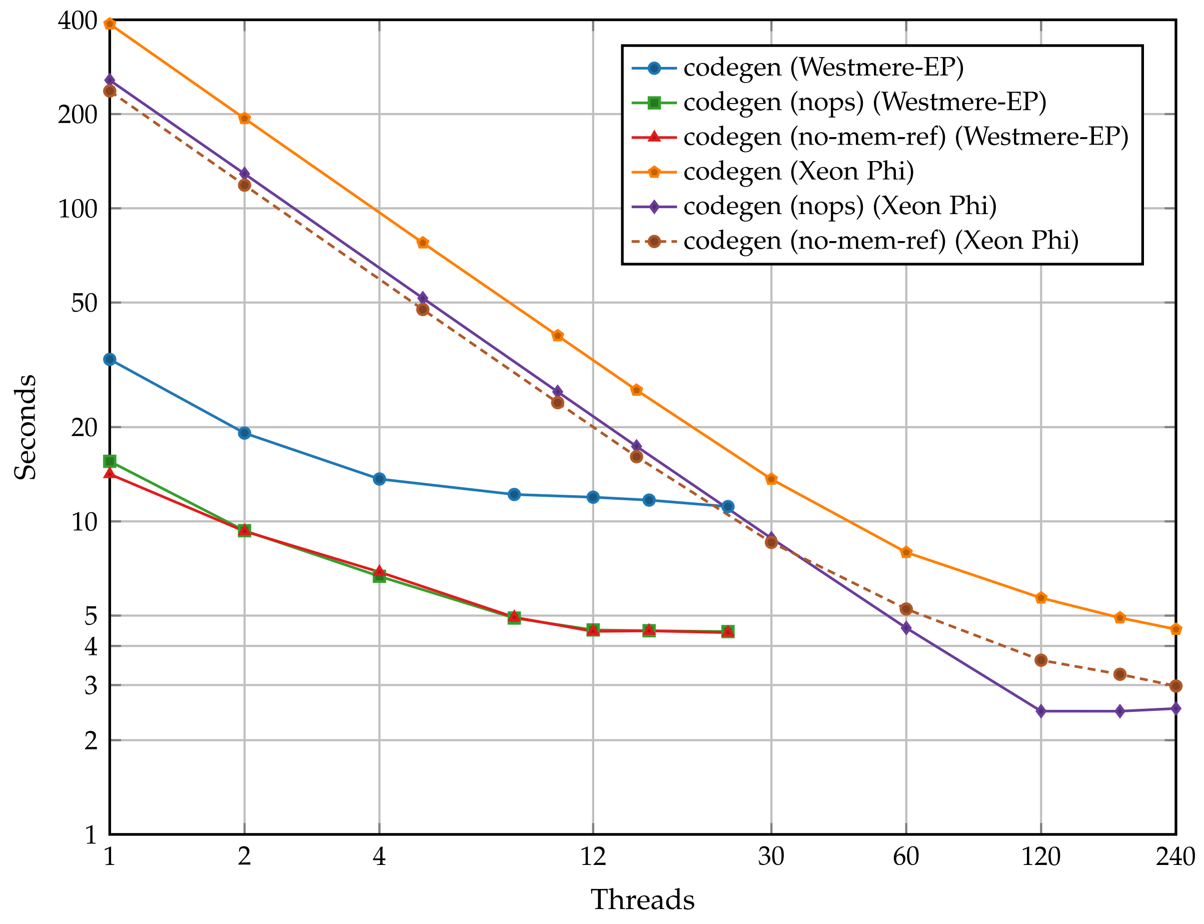


Figure 4.19: Runtime of evaluation using codegen and patched variants

	Westmere-EP		Xeon Phi	
	single thread	best result	single thread	best result
compilation	3.24	0.37	32.36	1.45
evaluation	32.92	11.15	387.98	4.52
total	36.16	11.51	420.34	8.11

Table 4.4: Runtimes of codegen phases in seconds

All in all, both variants are noticeably faster than the execution of the real generated code on both systems, indicating that the execution of the real code is not bound by instruction transfer, but slowed by the scattered memory accesses within the code.

4.6.5 Hybrid Computation

Figure 4.20 shows the portion of the compilation phase with respect to the total runtime. The portions of the respective compilation phases are almost identical on Westmere-EP and Xeon Phi as long as only a few threads are used.

However, while the impact of the compilation phase decreases or at least more or less stagnates on Westmere-EP, it grows much more dominant on the Xeon Phi, which shows that it scales worse than the evaluation on Xeon Phi. As a matter of fact, compilation on the Xeon Phi starts to slow down again when using more than 30 threads, as confirmed by Figure 4.18b.

All in all, compilation on Xeon Phi is slower than on Westmere-EP, with the best compilation time being 0.365s on Westmere-EP using all 24 threads compared to 1.454s using 30 threads on Xeon Phi. On the other hand, evaluation itself is almost three times faster on Xeon Phi with 4.522s using 240 threads compared to 11.148s using 24 threads on Westmere-EP as highlighted in Figure 4.19 and Table 4.4.

This fact could be leveraged by using the host system to cross-compile the trees to executable code suitable for Xeon Phi and subsequently execute it on the accelerator, which would result in the best overall runtime, as illustrated in Figure 4.21. It would also be possible to interleave both phases, as the Xeon Phi could start executing already compiled trees while the host continues to process the still remaining trees.

This hybrid computation can also be compared with the way OpenCL [76] operates, where the host compiles code at runtime for execution on the accelerator.

4.7 Conclusion

Sometimes changes that appear to be isolated at first glance can have significant implications on overall performance. In the case described in this chapter, the decision to split the original problem into two separate ones and use an iterative approach to compute a solution significantly shifts the overall workload within the application. To avoid creating new performance bottlenecks and restraints the now more heavily used

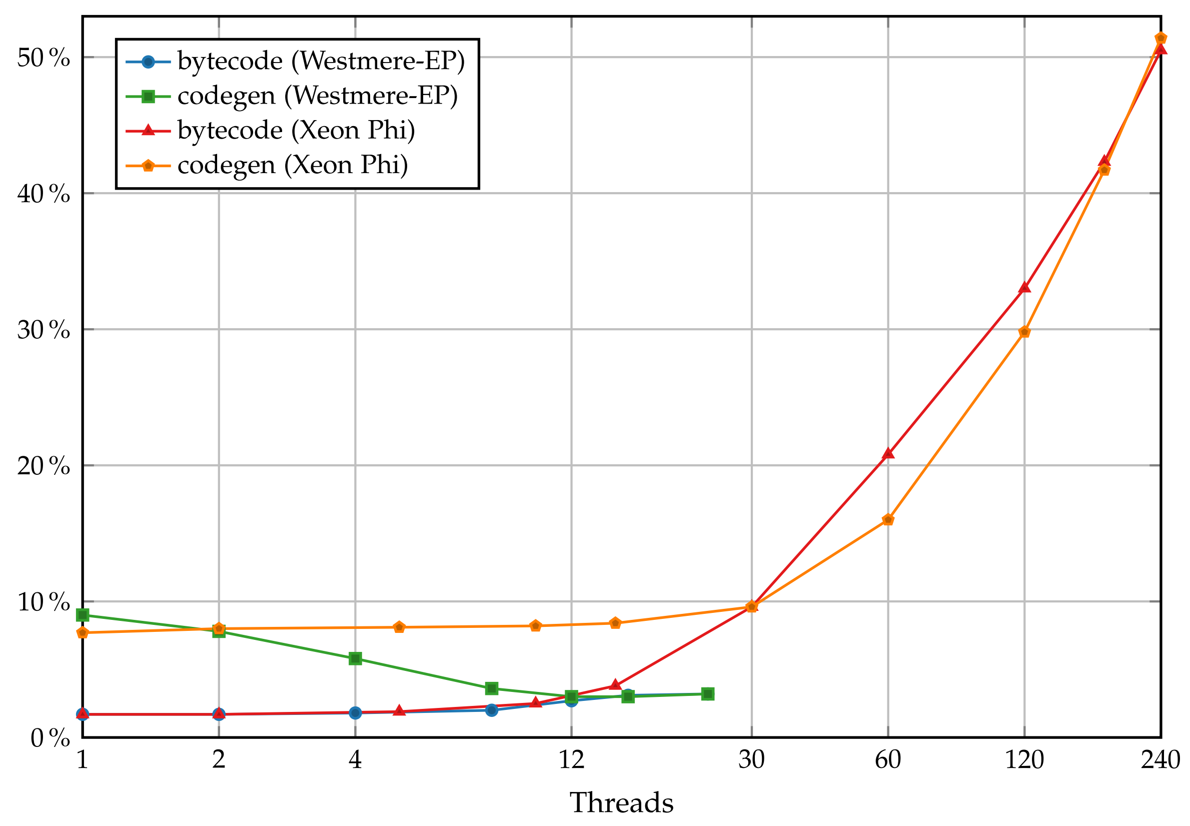


Figure 4.20: Portion of compilation phase of total runtime

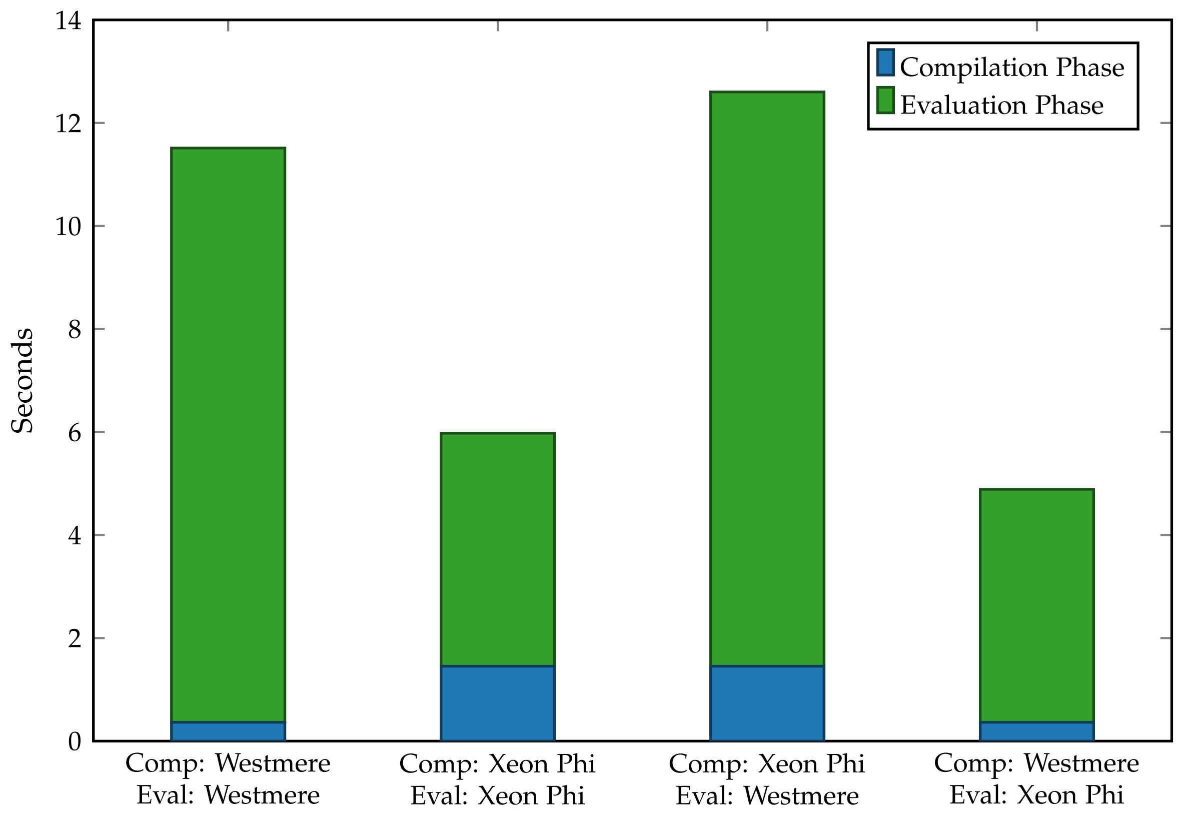


Figure 4.21: Possible hybrid combinations

parts of the application need to be under special scrutiny.

This is a common scenario originating from increasing computing capabilities. As the problems to be solved tend to get larger and more complex, they start to get too complex to be solved directly without adapting the algorithms within the application or splitting the problem into several smaller subproblems.

In Section 4.4 some algorithmic optimizations have been detailed, which can help to mitigate performance bottlenecks in general as well as bottlenecks originating from increased problem complexity.

One of the most effective optimizations is to be able to reuse already computed data instead of repeatedly recomputing it. By identifying and eliminating such duplicated computations, performance can be improved significantly, as it was done in this case study by reusing the result of shared subtrees.

In addition, it was shown how unoptimized data structures result in performance degradation when processing increasingly large data sets. The best approach is to replace such data structures with more suitable and efficient ones, but if the respective data structures are deeply interlaced within the application, this requires rewriting significant portions of the application which is not always feasible or worthwhile.

Instead, strategically placed local caches can be used to significantly speed up access to relevant data at performance critical parts of the application. Alternatively, the API to existing data structures can be extended to provide a more low-level and direct access. While this weakens the abstraction layer provided by the API, it can greatly improve performance as shown above for processing sparse matrix entries.

Moreover, it was demonstrated how performance can be improved by carefully restructuring confined parts of the source code and splitting up processing logic into smaller functions. This restructuring makes it easier for the processor to correctly predict branch targets and subsequently reduces performance degrading pipeline stalls.

Finally, in an effort to improve branch prediction even more, code generation techniques and implementations were presented, which allow to transform or compile expression trees into a representation that can be evaluated faster and more efficiently than the original tree-walking code – either by running a bytecode interpreter or by actually executing machine code.

As we can see in Figure 4.16, the bytecode implementation already improved performance noticeably. The bytecode can be generated extremely fast, is architecture independent and its evaluation already eliminates most scattered memory accesses necessary to load nodes and significantly reduces the number of branches during execution. At the same time it is also the most robust code generation implementation, as potential errors can be handled like in any other application.

However, best performance is accomplished by directly generating executable machine code, as there is no additional overhead by handling code involved. The execution speed is virtually identical across the different implementations, so the main difference in performance is compilation speed.

On the one hand, the results of the comparison of LLVM, AsmJit and codegen show that a high number of abstraction layers and a generic design come at a price and reduce compilation performance. On the other hand, using an existing library with an abstract API makes development easier and faster, as functionality such as register allocation or hardware specific opcode construction does not have to be implemented by oneself.

Also, directly generating executable machine code does make error handling and debugging harder. Subtle errors during code generation can result in wrong memory references or invalid instructions, causing the application to silently compute and return wrong simulation results or even to crash. Both incidents are very serious issues and need to be prevented reliably before this kind of code generation can be used in production.

These techniques are all powerful and cost efficient ways to increase the sequential performance and efficiency of data structures and individual functions of an application with a limited amount of work necessary.

The next step in further improving performance is parallelization. In this case study, compilation and evaluation of the expression trees was parallelized by processing multiple trees in parallel, which is very straightforward as there are virtually no dependencies between different expression trees.

However, even in the original code expression tree processing only accounts for about 60% of overall runtime, and this proportion gets even smaller when using a code generation implementation like bytecode (about 50%) or codegen (about 25%). Because of this, the application as a whole does not scale significantly with an increasing number of threads, as the runtime of the remaining sequential part remains unchanged (cf. Amdahl's Law [3]).

By using a benchmark application the behavior of the parallelized expression tree processing is investigated on both Westmere-EP and Xeon Phi, showing that a Xeon Phi can be used to efficiently evaluate expression trees. Additionally, for best performance a hybrid processing model can be employed in which the more complex code generation itself, i.e. the compilation phase, is performed using a general purpose processor as in the Westmere-EP system, and the generated code is then transferred to the Xeon Phi for execution.

Chapter 5

Case 2 – Electrostatic Field Simulation

5.1 Background

When constructing high-voltage devices such as a load breaker switch to be used in a substation, great care must be taken of dielectric design. It must be ensured that no spark-over between neighboring parts, such as a high voltage electrode and the device enclosure, can happen, as this might damage the device and subsequently cause outages. Similar to the thermal design of transformers discussed in Chapter 4, the objectives of a high-voltage device design typically include minimization of physical dimension and material costs, while at the same time guaranteeing all specifications required by the customer.

The most reliable method to prove the safety and conformance of a device is to perform real experiments with samples or prototypes in a high-voltage laboratory. There, extremal loads are applied to verify that no spark-over happens during the real life operation of the device.

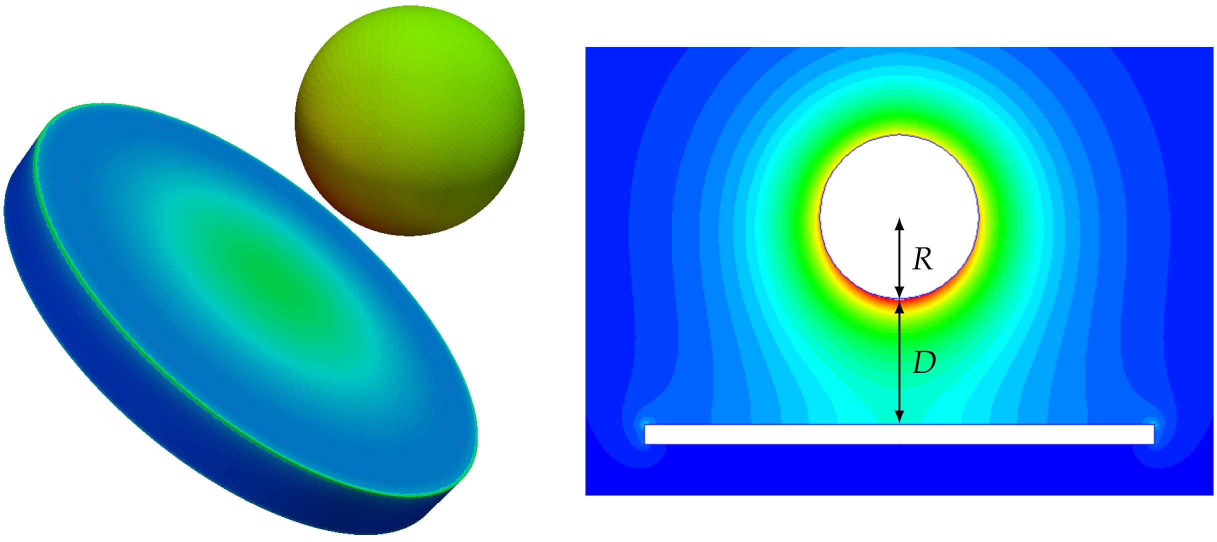
However, these experimental verifications are extremely cost intensive, as a prototype has to be constructed, which is possibly damaged or destroyed during the experiments. Also, if the verification fails, the device has to be adapted or redesigned and subsequently a new prototype has to be constructed for verification.

In addition to the money spent for constructing the prototypes itself, the process also consumes a lot of time, which can become problematic in case of an agreed delivery time or if a competitor finishes his product first. To avoid this, such devices and their behavior are usually simulated before building the first prototype.

The engineer develops a virtual model of the device using a *Computer Aided Design (CAD)* software, such as *AutoCAD*[1] or *PTC Creo*[2]. This design represents the exact physical layout, as well as used materials, their properties and so on. Using this design and various boundary conditions, such as the electric potential applied at different parts of the device itself or simulated surroundings, a simulation model can be generated.

This model is then used to compute the electric field inside the device. Afterwards, the result is analyzed to identify points or areas of high field strength that are potential candidates for a spark-over. In the next step the engineer can evaluate those candidates and adapt the design as necessary to further reduce the hazard of an electric breakdown.

[1]`http://www.autodesk.de` (retrieved January 2014)
[2]`http://www.ptc.com` (retrieved January 2014)



(a) 3D model illustrating electric field on surface of sphere and plane

(b) 2D cross section through sphere and plane illustrating electric field between objects

Figure 5.1: Visualization of electrostatic simulation

Figure 5.1 shows a visualization of the simulation result of a small exemplary model, consisting of a copper sphere floating above an aluminum plane. A potential of 100 000 volts is applied to the sphere, while the plane below it is grounded, i.e. a potential of zero volts is applied to it.

The colors represent the strength of the electric field, increasing from blue to red. In this case, the electric field strength peaks at the points on the sphere and the plane that are closest to each other. Additionally, the rim of the plane also shows a high field strength, which is typical for sharp edges.

This simple sphere-plane arrangement can be considered as a typical optimization problem in high voltage engineering. For the fixed distance $R + D$ between the plane and the center of the sphere (cf. Figure 5.1b) one can find a radius R of the sphere for which the maximum field strength is minimum.

In real life the engineers have to find an optimum shape of very complex arrangements with many geometrical parameters, which requires, similar to the case of thermal simulations, a large number of computations. Ensuring high performance of such computations is essential for applicability of simulation in an engineering environment.

5.2 Mathematical Methods

The electromagnetic field is described by *Maxwell's Equations* [58], which are a set of coupled partial differential equations collated by James Maxwell. The foundation for electrostatics is provided by Gauss's law from which Laplace's equation can be derived as the basic differential equation to be solved numerically. For a detailed introduction into electrostatics and electrodynamics refer to e.g. Griffiths [34].

The mathematical methods considered here focus on the integral approach which assumes that the electric potential, as the basic solution of Laplace's equation, can be expressed in form of an integral of the electric charge or flux.

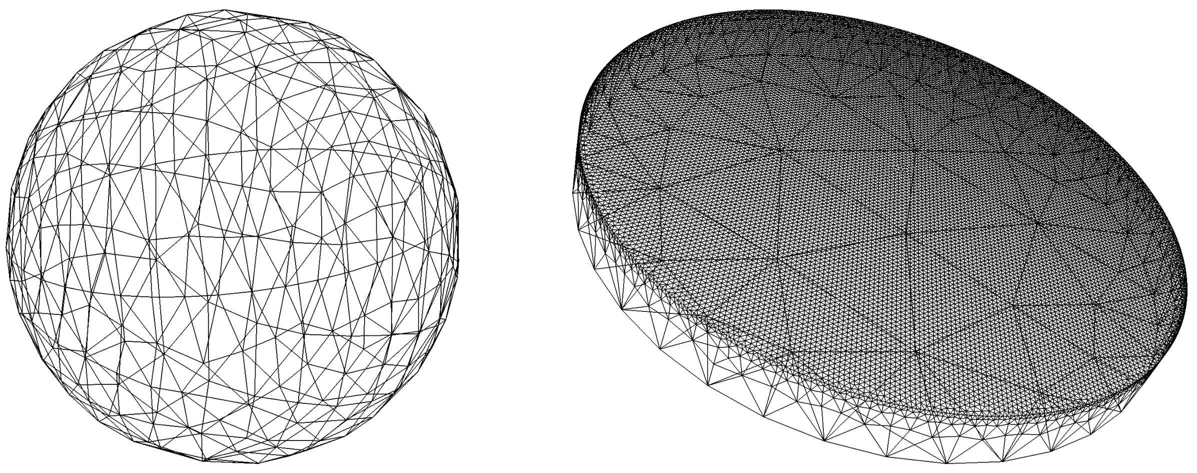


Figure 5.2: Boundary mesh of sphere and plane

5.2.1 Boundary Element Method

The *Boundary Element Method (BEM)* is a numerical method which can be used to solve certain types of differential equations, including Laplace's equation. A thorough introduction and analysis of the characteristics of the boundary element method can be found in Banerjee [6] or Wrobel and Aliabadi [85].

Similar to the more commonly known *Finite Element Method (FEM)* [88, 67] it uses a discretization of the involved objects. However, while FEM requires a discretization of the complete volume, a discretization of the boundary of involved objects is sufficient for BEM. This significant simplification of the model preprocessing is based on the fact that the electric permittivity as the relevant material property for the capacitive electrostatic analysis can be assumed linear.

The geometrical forms used to construct an appropriate discretization, or *mesh*, are called *elements*, the source of the name of both methods, while the vertices of those elements are usually called nodes. Commonly used elements are triangles for BEM and tetrahedrons for FEM, although more complex element shapes are possible and different ones may also be mixed.

Figure 5.2 shows a boundary mesh representing the two objects, sphere and plane, which was generated to simulate the model depicted in Figure 5.1. The mesh discretizing the plane is significantly more dense on the upper side of the plane, the one facing towards the sphere, increasing the accuracy of the computation at this region.

The discretization is then used to assemble a linear system of equations by integrating the electric charge or flux over all boundary elements for all discretization items (nodes or elements, each represented by one equation):

$$Ax = b, \qquad A \in \mathbb{R}^{n\times n},\ x, b \in \mathbb{R}^n,\ n \in \mathbb{N} \tag{5.1}$$

where A and b represent the individual coefficients respectively the right-hand side of the equations, n the number of unknowns and x the solution vector to be computed.

Often, this results in a fully populated matrix A and subsequently in high memory requirements to store and process it. Also, the type and size of the elements and the resulting number of nodes used to generate the mesh define the granularity of the

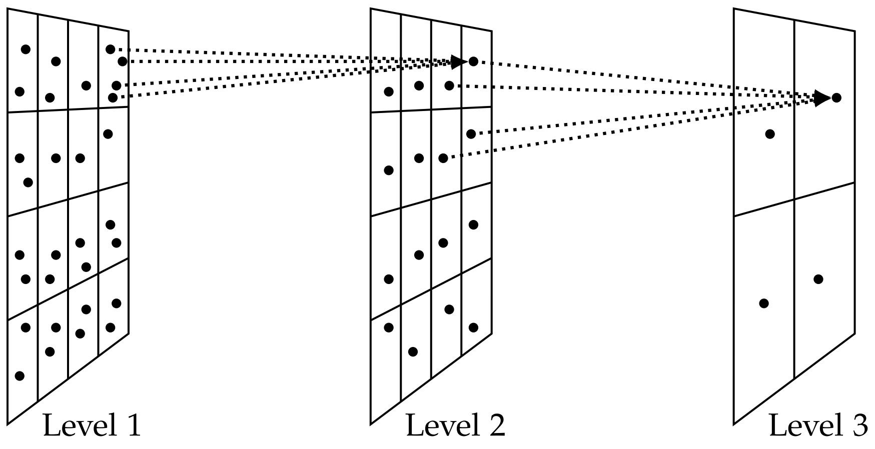


Figure 5.3: Illustration of hierarchical fast multipole method

discretization. The higher the number of nodes, the larger the system of equations and therefore the computational and memory requirements.

Finally, the solution of the equation system can be used to compute the actual electric field and, if required, generate an appropriate visualization.

5.2.2 Fast Multipole Method

To properly compute the electric field, the combined effect and interaction of every single charge on every other charge needs to be considered. When implemented straightforward this results in quadratic $O(k^2)$-complexity, with k being the number of charges.

In 1985 Greengard and Rokhlin [33] presented the *Fast Multipole Method* to improve this kind of computations and reduce runtime as well as space complexity. The basic idea is to avoid the need to explicitly consider the effect of every charge on every other charge.

Instead, the combined effect of a localized group of charges is computed and used as a representative "virtual" charge instead of the individual ones. When this combined effect is applied to other charges that are sufficiently far away, the deviation from the correct result is negligible.

This also works the other way around: When computing the interaction of faraway charges on a localized group of charges, the effect on the virtual representative charge is computed first. Afterwards, the combined effect of all far away charges on the virtual charge is distributed to the individual charges in the relevant group.

To further improve efficiency, the fast multipole method can be applied in a hierarchical manner to appropriately handle charges at different distances to each other. Figure 5.3 illustrates the unification of individual points into a single multipole at the respective next higher level.

Due to using representative charges instead of every individual charge, this method introduces an error into the calculated result. However, it is possible to establish an upper bound for this error, thereby guaranteeing a certain quality of the result. As detailed in the original publication [33], this scheme can significantly reduce the size of the required memory and speed up overall computational time.

5.2.3 Adaptive Cross Approximation Method

Another method to speed up a BEM computation is the *Adaptive Cross Approximation method* (ACA), presented by Bebendorf [7, 8], Bebendorf and Rjasanow [11]. It is based on hierarchical $\mathcal{H}$-matrices, which were introduced by Hackbusch [37] and are so-called *data-sparse* matrices.

Ordinary sparse matrices only contain a small percentage of non-zero values, which are stored in a compact way along with their position within the matrix. Every value not explicitly referenced in a sparse matrix is implicitly taken as being zero.

The data-sparse hierarchical matrices on the other hand may represent fully populated matrices. But instead of explicitly storing all values only enough data is kept to be able to recreate an approximation of the original matrix or to compute certain operations such as a matrix vector multiplication. Also, because of the reduction in stored data, these operations can be performed efficiently, as fewer data must be loaded from memory and subsequently processed.

The beneficial feature of the adaptive cross approximation method is, that, for some types of matrices, such as the ones occurring in certain BEM applications, it can be used to construct an $\mathcal{H}$-matrix from few of the original matrix entries. Hence, only those matrix entries, i.e. the coefficients of the linear equation system, that are required by the ACA method need to be calculated in the first place, reducing computational time and memory space.

Similar to the fast multipole method, application of the adaptive cross approximation method results in an approximation of the original result. Again, the error of the approximation can be constrained. A detailed introduction as well as an analysis of computational time, memory complexity and error bounds can be found in Bebendorf [9].

5.3 Test Models

Five test models were used for the evaluation and comparison of different simulation solvers. The models were chosen to be of increasing complexity, while being simple enough to be simulated on a single workstation. Four of these models, *EXK01*, *Dielektrik*, *GIS Isolator* and *GIS Arrester* are real-live models provided by courtesy of ABB.

The *Dielektrik* model simulates the flexible connection from a conductor located outside of the enclosure to the generator circuit-breaker that will be installed within the enclosure and which protects the generator in power plants. The simulation is used to ensure that the distance between the connector and the enclosure is large enough to reliably prevent spark-overs.

EXK01, *GIS Isolator* and *GIS Arrester* are all parts of gas insulated switchgear (GIS). There, for increased protection against sparks, the complete switchgear is built as a gas-tight entity filled with pressurized sulfur hexafluoride (SF6), which acts as an electrical insulator.

The smallest model, *EXK01*, represents a combined three-phase circuit breaker and earthing switch within a 145 kV switchgear. The simulation is used to optimize the geometry of the contact support to keep the maximum electric field strength below a predefined critical limit.

	Dimension	Elements	Nodes
EXK01	~ 39 000	~ 79 000	~ 157 000
Dielektrik	~ 62 000	~ 128 000	~ 274 000
Kugeln	~ 93 000	~ 185 000	~ 370 000
GIS Isolator	~ 144 000	~ 292 000	~ 625 000
GIS Arrester	~ 202 000	~ 425 000	~ 1 038 000

Table 5.1: Test model mesh characteristics

The *GIS Isolator* mechanically supports a one-phase high-voltage conductor in the middle and isolates it from the grounded casing within a 400 kV switchgear. The simulation is performed to optimize the device for minimal electric field strength on the shielding of the conductor.

GIS Arrester, the most complex of the test models, is used to protect against overvoltage within a 400 kV network. Within it, varistors are arranged in three columns and electrically connected in a spiral. The simulation is performed to compute the distribution of potential along the helical assembly of varistors during normal operation.

Finally, the fifth model, *Kugeln*, is an artificially generated toy model, constructed to have a complexity between that of *Dielektrik* and *GIS Isolator* and consists of a large sphere, which is surrounded by a circle of smaller spheres.

Figure 5.4 shows an exemplary visualization of all five models, while Table 5.1 lists some characteristic values of their discretization. The dimension column within the table refers to the dimension of the system of equations constructed and solved by reference solver *polopt0* which will be described in the following section. Also, the dimension corresponds approximately to the number of corner nodes of the triangular boundary elements.

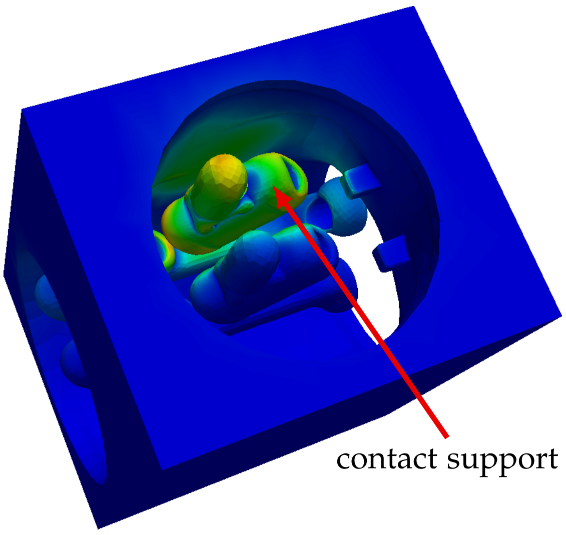
5.4 Established Reference Solver

Development of the *polopt0* BEM solver started in the 1980s. It is predominantly implemented in Fortran and has been constantly improved and extended since then. During the 1990s, it was parallelized using a message passing scheme based on PVM [13, 23] and was later ported to use MPI.

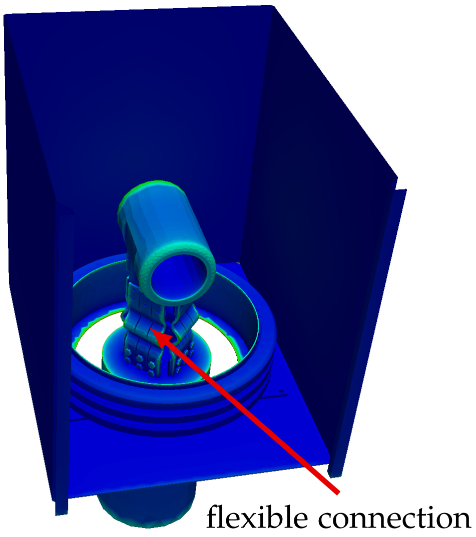
Over time, most bugs and critical cases where found and eliminated, resulting in a very robust and reliably solver that is still used for production computations within ABB.

5.4.1 Work Flow

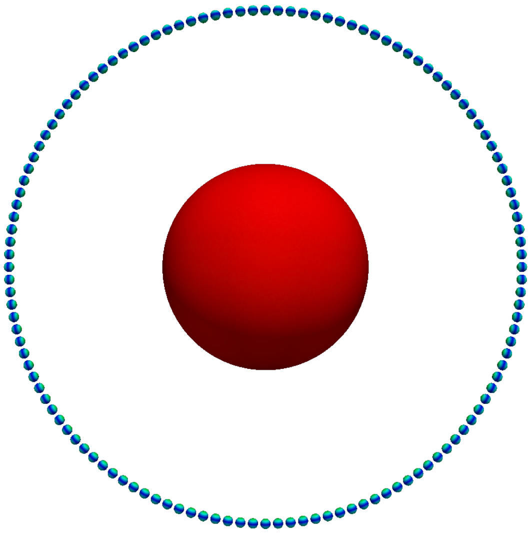
The work flow of polopt0 consists of a startup period during which the input data, i.e. the model to be simulated, is loaded and processed, followed by three distinct phases which dominate overall runtime. During the first phase, the coefficients of the large



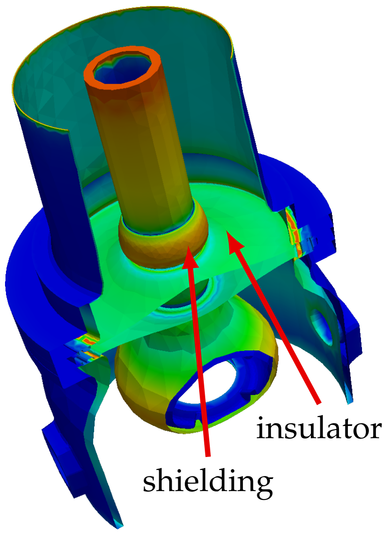
(a) EXK01



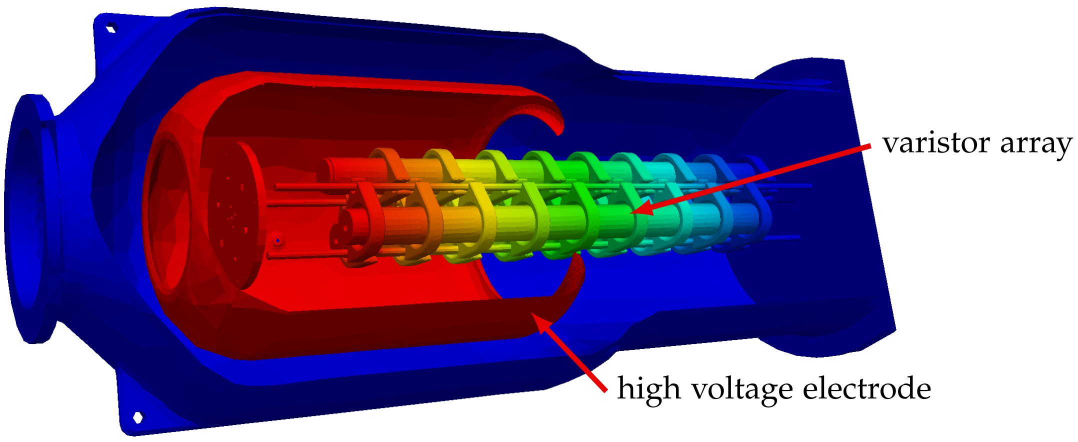
(b) Dielektrik



(c) Kugeln



(d) GIS Isolator



(e) GIS Arrester

Figure 5.4: Visualization of simulated test models

linear system of equations resulting from the BEM implementation are computed and stored as a dense matrix in memory, resulting in memory requirement of quadratic $O(n^2)$ complexity for n unknowns. This system of equations is then iteratively solved in the second phase. Finally, in the third phase, the computed solution is used to calculate the electric field at the nodes of the underlying mesh describing the simulated model.

The dense matrix created in phase one is not stored as a single big block, but instead split into chunks of several rows that are allocated as necessary. This repeated allocation of memory corresponds exactly with the stepwise increase in overall memory consumption visible in Figure 5.5a from the start until about minute 5.

Memory consumption then remains unchanged while the equation system is solved in phase two. When the solution is computed at around minute 6, the matrix is no longer used and the corresponding memory is released, resulting in a significant drop of overall memory consumption, which then again remains stable during phase three.

As of now, the solution of the linear equation system is computed using the *Generalized Minimal Residual Method (GMRES)* presented by Saad and Schultz [70], which iteratively finds a solution for Equation (5.1).

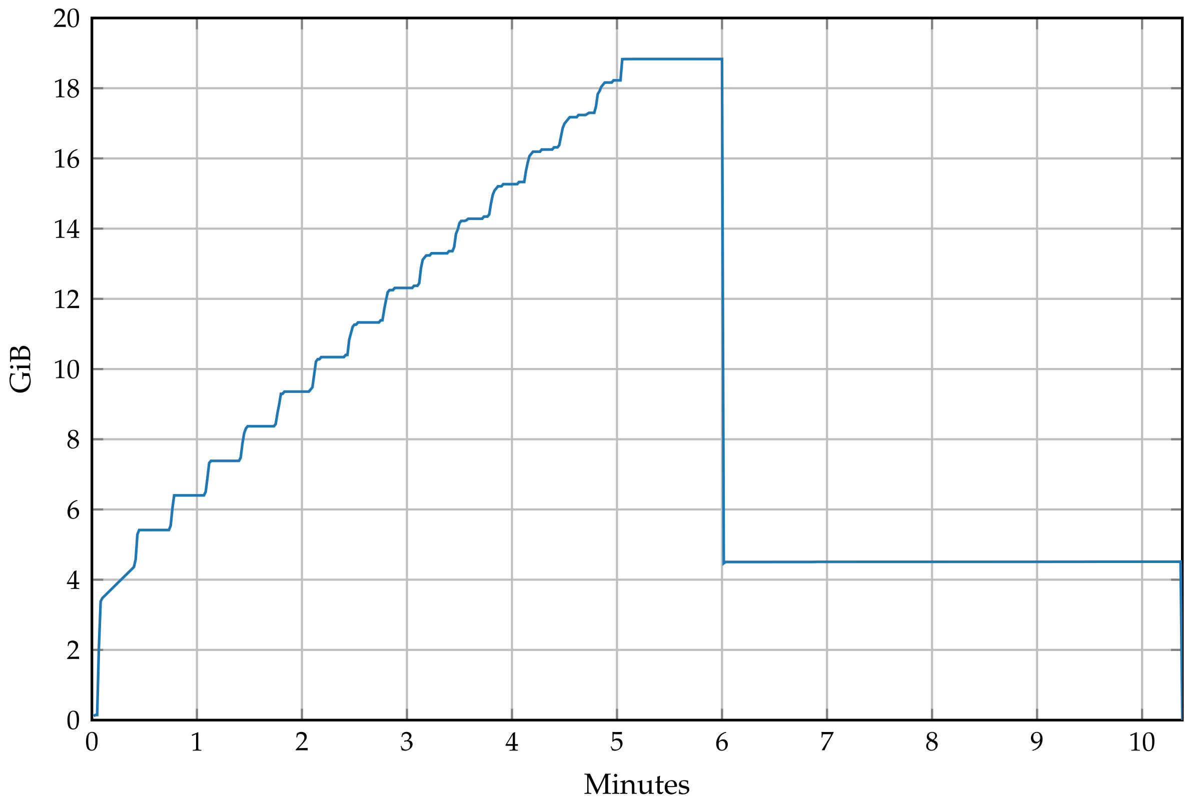
GMRES can be treated as a black-box solver, as it does not need to directly access the matrix A. Instead, it is only necessary to implement a matrix-vector product $A \cdot q_k$ with the coefficient matrix A and vectors $q_k \in \mathbb{R}^n$. In the course of the iteration process a vector q_k is generated for every iteration step $k = 1, 2, ...$ and the series of the generated vectors gradually approximates the required solution vector.

5.4.2 Parallelization

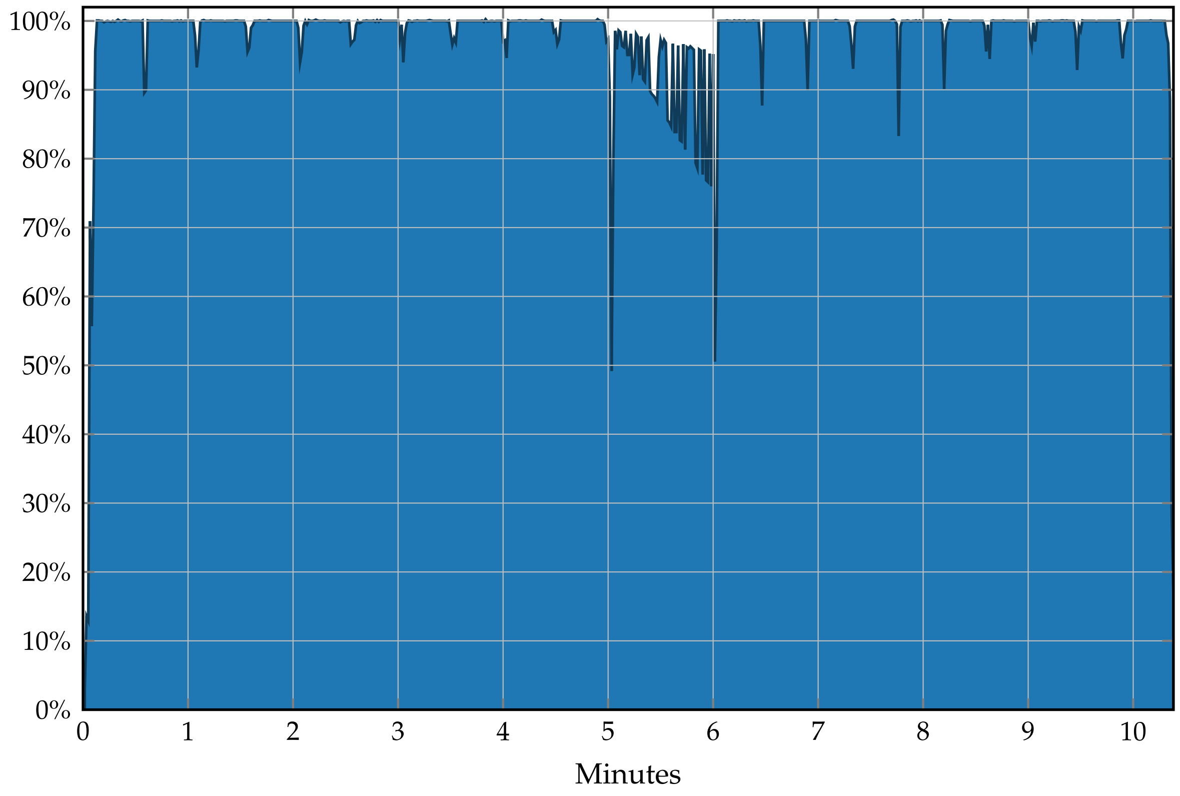
Using GMRES with a dense matrix makes the distributed memory parallelization of polopt0 straightforward. At the beginning, during the startup period, the input data, such as the discretization of the model or applied potentials are distributed to every MPI process. Then, the coefficient matrix A that is to be constructed is virtually split into as many chunks of rows as MPI processes are used, and every chunk is assigned to a specific process. After that, every process independently computes the values of its assigned matrix rows, which requires virtually no interaction between the different processes.

Because of this, parallel efficiency is high, as illustrated by Figure 5.5b which shows the overall processor utilization of the processes. After a short start-up time, utilization remains almost constantly at 100% until about minute 5 at which point all coefficients are computed and the dense matrix is fully set up and stored in chunks across all MPI processes. The short drop at the end of phase one is due to a small imbalance at distributing rows among processes. Because of this, some processes already finished their share of the computation, while others still have some coefficients to compute left.

Solving the equation system in parallel is a bit more complex. Basically, a single master process is responsible for executing GMRES iterations itself, while the necessary matrix-vector product $A \cdot q_k$ is computed in parallel for every iteration step k. To do this, the vector q_k is distributed from the MPI master to every slave process which then computes the relevant entries of the solution vector matching the locally computed



(a) Memory requirement



(b) System utilization

Figure 5.5: Polopt0 memory requirement and system utilization over time for model *Dielektrik* and 16 MPI processes

and stored rows of A.

A process assigned the rows $[s;t]$ of A, $s,t \in [0,n], s > t$ can use these rows and the distributed vector q_k to compute the components $[s;t]$ of the result vector of $A \cdot q_k$. These partial results are then sent back to the master process which reassembles them to obtain the full result vector and continues with the execution of GMRES.

The computational cost of a straightforward matrix-vector multiplication is $O(n^2)$. However, if the matrix rows are evenly distributed among p MPI processes the absolute time required to compute the result is reduced by a factor of p. In addition to the parallelized matrix-vector product, $O(kn)$ floating point operations have to be performed by the master process within GMRES at iteration step k, i.e. the computational cost steadily increases with every iteration step.

Again, Figure 5.5b shows this very clearly. During phase two – between minute 5 and minute 6 – processor utilization changes more frequently, as parallelized vector-matrix computation and sequential computations within GMRES alternate. Also, as the sequential part gets more dominant, the *average utilization* decreases steadily.

Finally, after GMRES has finished, the solution vector of the equation system is distributed to every process and phase three is started. Like phase one every process independently computes its predefined share and returns the result to the master process at the end of the computation. More details on the parallelization of the reference solver can be found in Blaszczyk et al. [15].

5.4.3 Localized Matrix Compression

To reduce the overall memory requirements of the traditional polopt0 solver, a minimally invasive localized matrix compression technique is evaluated. To preserve the straightforward and efficient parallelization and to avoid complex restructuring of existing code, the compression is performed at a very localized level, namely individual matrix rows.

Because of this, the parallelization and matrix construction itself need not be touched. Instead, every time one of the aforementioned chunks of memory used for matrix row storage is fully filled, it is compressed before the next set of rows is computed within a new memory chunk.

The compression scheme itself is very straightforward and is based on matrix row normalization using the diagonal element. As the magnitude of the values in a row varies heavily, a lot of them become quite small in magnitude ($< 10^{-3}$) due to this normalization process. This primarily happens for matrix entries related to small boundary elements located far away from the integration point assigned to the respective matrix line. During a vector matrix multiplication, the impact of small variations of these small values is limited.

Hence, a series of consecutive row entries with almost the same small magnitude and sign can be replaced with the average value of those entries without significantly influencing the final matrix vector product. This average value can be stored in a compressed way, as only the value itself and the number of entries replaced by it is needed.

All in all, the first two phases of polopt0 are influenced by this compression scheme. During construction of the matrix within the first phase, the relevant matrix rows are compressed at specific intervals, when a new memory chunk is about to be allocated.

Then, during the second phase, all matrix rows have to be decompressed repeatedly to perform the matrix vector multiplication required by the GMRES iteration. As the matrix is no longer needed for the third phase, the computation of the electric field at the mesh nodes, the matrix compression has no implication at all at this point.

This localized compression is a rather young implementation. Consequently, the experience with it, especially regarding overall error introduced into the simulation result as well as compression rate is limited, but preliminary results are encouraging. As will be seen in Section 5.6 both absolute runtime as well as parallel efficiency are not significantly impeded due to the implemented data compression. Also, for the models evaluated so far, accuracy of the computed results is within acceptable limits.

5.5 Alternative Solvers

From 2009 until 2013 the *CASOPT: Controlled Component- and Assembly-Level Optimization of Industrial Devices*[3] project took place under the umbrella of the *Seventh Marie Curie Framework Program* (FP7)[4] and was funded by the European Union. The CASOPT Project Consortium consisted of industry partner ABB and the academia partners University of Cambridge, Technische Universität Graz and Technische Universität München.

The project's aim was to research and develop or improve methods and tools to automatically optimize industrial devices. Instead of manually adapting the design according to the results and possible problems identified by a simulation, the idea was to start with an initial design and automatically adapt it to minimize a specified objective, such as production costs, while still guaranteeing all constraints.

During the project several simulation solvers where developed, enhanced or adapted to serve as research tools, potential supplement or even future replacement of the original *polopt0* solver: *polopt3*, *BETLdiel* and *gobem*.

polopt3 *Polopt3* is a direct advancement of polopt0, which is developed by ABB and partially even incorporates code from polopt0. Contrary to polopt0 though, it does not need to construct the dense matrix, but is instead based on the fast multipole method to speed up computation and increase memory efficiency. Specifically, large parts of the implementation are based on the works of Lage [54] and Schmidlin [72].

Similar to polopt0, polopt3 is parallelized exclusively using a message passing scheme, specifically MPI, making it possible to execute it on multiple nodes within a compute cluster.

BETLdiel Most of the *BETLdiel* solver was developed from scratch during the CASOPT project. Its foundation is the *Boundary Element Template Library (BETL)* [39], a "C++ template library for the discretization of boundary integral operators as they arise in various physical and engineering applications". This library is now further developed and studied at the seminar for applied mathematics of ETH Zürich. To

[3]`http://www.casopt.com` (retrieved December 2013)
[4]`http://cordis.europa.eu/fp7/home_en.html` (retrieved December 2013)

speed up computation and reduce memory consumption, BETLdiel relies on the AHMED [10] library which implements and applies the adaptive cross approximation method (ACA).

Contrary to polopt0 and polopt3, BETLdiel is parallelized using OpenMP instead of MPI. Because of this, it is currently not possible to use it on a cluster to simulate very complex models that are too memory intense for a single workstation.

gobem *Gobem* is developed at the department for numerical mathematics of TU Graz and, like polopt3, implements a fast multipole method.

The focus of gobem is on the accuracy of the solution rather than on high performance. In contrast to traditional formulations implemented by all other tested solvers, gobem implements a new Steklov-Poincare BEM formulation [2] which is capable to correctly solve models with very large differences of permittivity of materials. However, the new formulation does not have significant accuracy advantages for standard dielectric models, which are the main subject of this work.

It is parallelized using a hybrid model of MPI and OpenMP. At the beginning, a domain decomposition of the model that is to be simulated is performed. Subsequently, every independent domain is processed by a specific MPI process. The computation of each domain, i.e. the individual MPI process, is then parallelized using OpenMP. It should be noted that this scheme requires the use of exactly as many MPI processes as there are domains, limiting the number of usable nodes within a large cluster.

When processing the *GIS Arrester* test model, gobem requires a significant amount of time to solve the arising system of equations. Even when using 32 processor cores, a full simulation takes more than two and half days and more than four days on 16 cores. Observations indicate that runtime further slows down with lower number of threads and that a sequential simulation would take about a month. Because of this a full speedup evaluation is impracticable in the course of this thesis.

5.6 Solver Evaluation and Comparison

Apart from correctness of results, the duration of a simulation – and therefore runtime performance of the solver – is probably the most important factor. The longer a simulation takes, the longer an engineer has to wait for the result of his simulation and, depending on overall system utilization, other engineers have to wait even longer for their simulations to start.

As mentioned above, the test models used in this thesis were selected to be simulated on a single workstation. Specifically, the Westmere-EX system is used for all performance evaluations of these solvers and test models. This was necessary, as two of the evaluated solvers, BETLdiel and gobem, are (partly) parallelized using shared memory, making it impossible to fully distribute computation on a cluster. Also, this makes a sequential, non-parallelized computation possible, which provides a reference for speedup assessment.

However, real life models often are significantly more complex, making it impossible to properly simulate them on a single workstation, so a cluster with many compute nodes will be required to simulate those models within an acceptable time or even

		polopt0	**polopt0 (compression)**	**polopt3**	**BETLdiel**	**gobem**
EXK01	single	49	48	11	83	58
	best (cores)	2 (32)	2 (32)	5 (16)	3 (32)	6 (16)
Dielektrik	single	156	171	21	134	144
	best (cores)	6 (32)	6 (32)	8 (16)	6 (32)	13 (16)
Kugeln	single	326	358	30	317	263
	best (cores)	12 (32)	13 (32)	16 (8)	11 (32)	30 (16)
GIS Isolator	single	913	913	75	2063	1904
	best (cores)	36 (32)	38 (32)	33 (8)	182 (32)	184 (32)
GIS Arrester	single	2018	1968	112	–[5]	–[6]
	best (cores)	83 (32)	85 (32)	62 (4)	–	3703 (32)

Table 5.2: Solver runtime for each test model in minutes

simulate them at all. Because of this, the following evaluation concentrates on the attainable speedup across the different test models and solvers, as well as the required memory, to see if solving very complex models would be feasible at all.

Furthermore, polopt3 supports parallel execution only for certain numbers of MPI processes, namely a power of two. Due to this, only these numbers (1, 2, 4, 8, 16 and 32) are used for speedup evaluation, even though the Westmere-EX system features 40 physical cores.

It has to be stressed that great care has to be taken when trying to directly compare absolute runtime of these solvers. They do not represent different implementations of the same underlying algorithm, but instead use different basic approaches, such as different handling of the mesh elements, all of which have their advantages and disadvantages.

Also, the solvers partially are – to some extent – only prototype implementations to test and study specific approaches, meaning that they are not necessarily optimized for optimal performance. While keeping in mind the limited significance of those numbers, Table 5.2 presents selected absolute runtimes for the sake of completeness. The table lists the runtime in minutes each solver required for each test model when running sequentially on a single processor core. Additionally, the fastest achieved runtime using parallel execution on multiple cores and the respective number of cores is listed as well.

Besides runtime performance, memory requirements are also very important. The

[5]Due to a software bug BETLdiel was unable to simulate *GIS Arrester*

[6]Sequential runtime of gobem for *GIS Arrester* is too long to be practical

higher the memory requirements of a solver, the more hardware has to be provided or fewer simulations can be executed at the same time, resulting in longer waiting time. In the worst case, a simulation is not possible at all, because the available hardware does not provide enough memory to execute the solver and additional hardware is too costly.

Memory requirements will almost always increase with model complexity and precision of the discretization. This is confirmed by Figure 5.6a, which shows the maximum absolute memory consumption of each solver and corresponding test model.

Probably even more interesting though is the increase in memory demand in relation to the complexity and precision of the discretization of the simulated model. Figure 5.6b shows the amount of memory required by the individual solvers for each test model in relation to the number of elements used for the mesh representing the model.

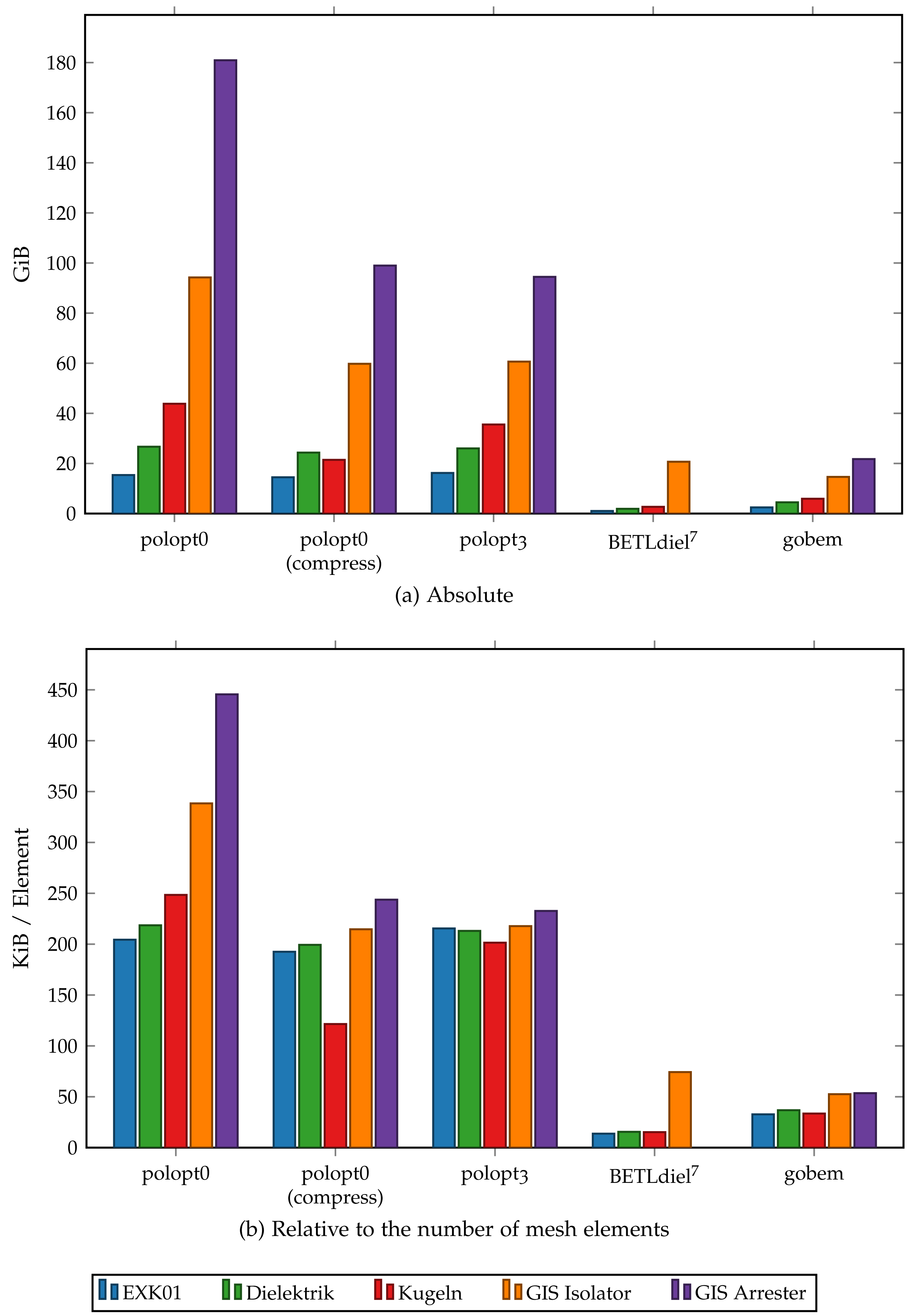
5.6.1 polopt0

Figure 5.7 shows clearly that polopt0 consistently improves with growing number of cores for all five test models, reaching speedup factors of about 22 to 27 when running on 32 cores. The attainable speedup is partly limited by the sequential startup period and potential work load imbalance, i.e. an uneven distribution of matrix rows among the MPI processes. Speedup is further reduced by the sequential part of the GMRES iteration, which includes the distribution of the generated vectors from the master process to all slaves. Due to historical reasons, the latter is currently implemented by transferring the vector to every individual slave one by one, increasing the required time with increasing number of used MPI processes.

As the compression implementation does not touch the underlying algorithms and parallelization, this polopt0 variant shows the same fundamental behavior as the unmodified version. The compression algorithm increases time spent for matrix construction during phase one up to only about 5%, as it has to be performed only once per matrix row and the computational cost to calculate the individual matrix row values still dominates this phase. Decompression on the other hand has to be performed during every step of the GMRES iterations, resulting in a more noticeable runtime increase of up to 40% for the second phase.

Furthermore, observation shows that the GMRES algorithm does need more iteration steps to compute a solution of the system of equations. Everything else being identical, this is a direct result of small errors and deviations from the original values created by the compression and decompression cycle, further reducing runtime performance and the attainable speedup. The actual effect on runtime, though, depends on the compression rate, i.e. the structure of the simulated model, as well as the configuration of the computer system used for simulation. Especially when executed on a very small number of processor cores on a NUMA system, the reduced memory requirements also result in reduced storage of data in non-local memory.

When considering the memory required by polopt0 during the simulation, Figure 5.6b demonstrates that polopt0's relative memory consumption consistently grows with increasing model complexity. This is due to the fully populated dense matrix that is constructed during simulation and which has a storage complexity of $O(n^2)$, leading to an overall quadratic increase of memory consumption.



(a) Absolute

(b) Relative to the number of mesh elements

Figure 5.6: Memory requirements

[7]Due to a software bug BETLdiel was unable to simulate *GIS Arrester*

The localized matrix compression variant on the other hand demonstrates more stable relative memory consumption, as a result of matrix compression. While absolute memory consumption is almost identical for the smaller models *EXK01* and *Dielektrik*, the other three models show a noticeable decrease. Especially the *Kugeln* model compresses very well, due to the extremely uniform layout of the model itself.

5.6.2 polopt3

Contrary to polopt0, polopt3 shows serious issues regarding parallel efficiency on all five test models, barely reaching a speedup of three.

Due to the adaptive hierarchical implementation of the fast multipole method, tree-like data structures, so-called panel cluster trees, are created within polopt3 to appropriately and efficiently handle interactions of charges at different distance layers. These trees can be very imbalanced, i.e. the depth of the subtree below different children of a node may vary considerably. During computation, these data structures have to be repeatedly traversed up and down.

This implementation is quite efficient in case of sequential runtime, as seen in Table 5.2. Moreover, even though absolute memory consumption for the two smaller test models is comparable to that of polopt0, the required memory does not increase as much for larger test models, i.e. the memory consumption in relation to model complexity is stable.

However, as is often the case, evenly distributing the construction and processing of tree-like data structures among multiple threads or processes and efficiently handling them in parallel is hard, especially when the structure of the trees highly depends on the input data.

In this case, trying to execute the solver on an increasing number of processors cores even results in a slowdown of overall runtime, as the additional parallelization overhead outweighs the benefits of the parallelization in the first place. This is visible in Figure 5.7 by observing the decline of the achieved speedup when simulating *Dielektrik* on more than 16 cores, or the *GIS Arrester* on more than as few as 4 cores.

5.6.3 BETLdiel

Similar to polopt0, BETLdiel shows consistent speedup for the three smaller test models, even surpassing polopt0 for *EXK01* and *Kugeln*. At the same time, the adaptive cross approximation method shows its advantage regarding memory requirements, as the relative memory consumption remains stable during the simulation of these three models.

For the *GIS Isolator* model, however, things are a bit different. In this case, the speedup of BETLdiel is significantly lower than that of polopt0, especially for a high number of cores. Starting at 8 cores, parallel efficiency drops considerably, resulting in a speedup of merely 11 at 32 cores, while at the same time the amount of memory required increases significantly. This indicates a strong correlation between speedup and memory requirements on the one hand and the simulated model on the other hand.

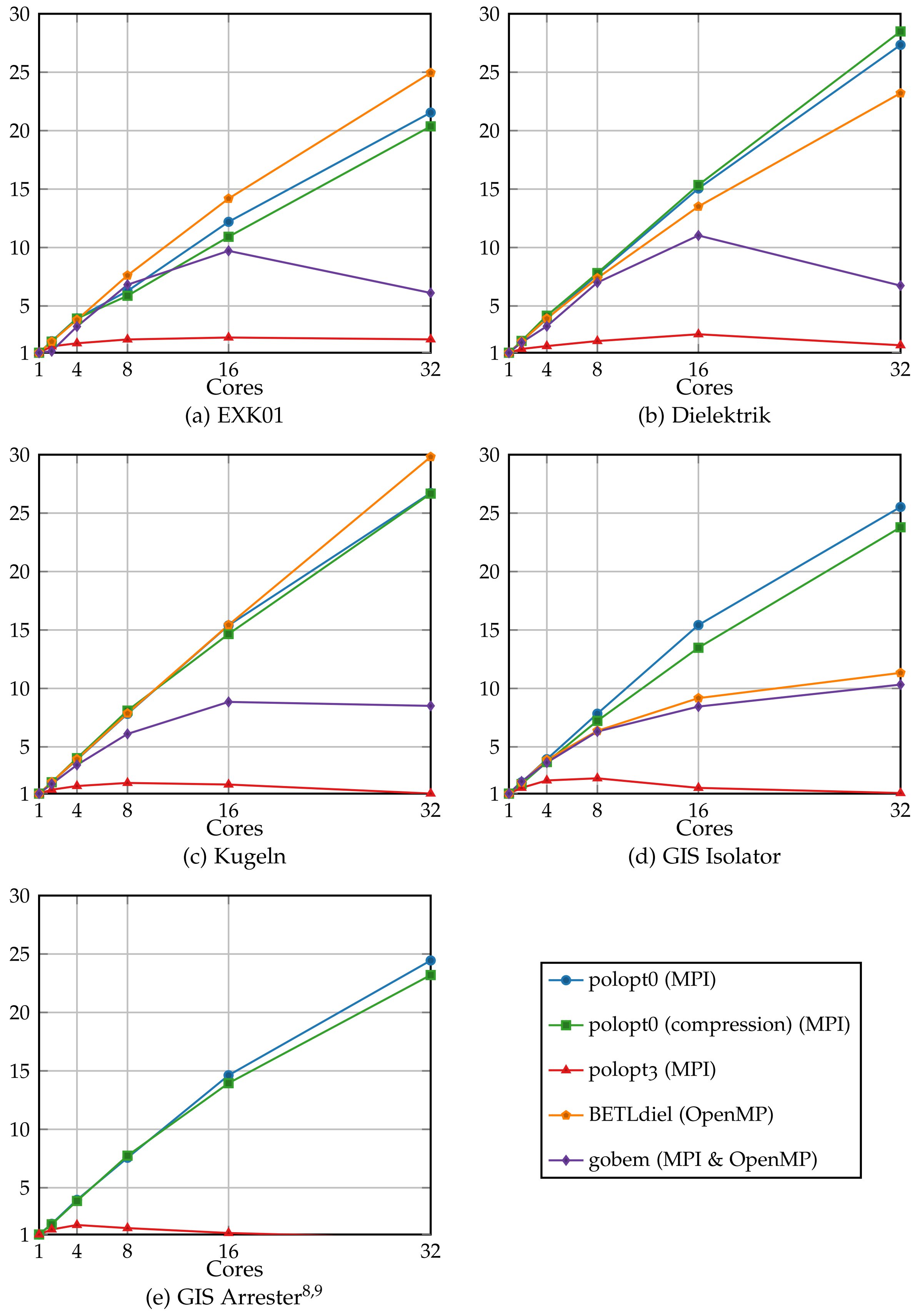


Figure 5.7: Speedup of solvers for each test model

[8]Due to a software bug BETLdiel was unable to simulate *GIS Arrester*
[9]Sequential runtime of gobem for *GIS Arrester* is too long to be practical for speedup evaluation

ACA accuracy	Iterations	Runtime	Memory
1.0×10^{-4}	13	~ 10 Minutes	2.6 GiB
5.0×10^{-5}	10	~ 11 Minutes	2.7 GiB
2.5×10^{-5}	9	~ 11 Minutes	2.9 GiB
1.0×10^{-5}	8	~ 63 Minutes	35.9 GiB
5.0×10^{-6}	8	~ 66 Minutes	58.3 GiB
1.0×10^{-6}	7	~ 69 Minutes	66.0 GiB

Table 5.3: Influence of *ACA accuracy* on BETLdiel for model *Kugeln* on 32 cores

This is confirmed by observing the effect of varying the precision or accuracy of the matrix approximation performed by the adaptive cross approximation method. This accuracy is configured by setting the admissible approximation error. Table 5.3 shows the behavior of BETLdiel when computing the *Kugeln* model on 32 cores, while varying the ACA accuracy parameter of the underlying AHMED library.

Increasing the accuracy of the approximation results in an increase of overall runtime required to fully simulate the model, while at the same time the number of iterations required to solve the system of equations is reduced. To guarantee an increased accuracy, the ACA method does need additional coefficients of the original system of equations to improve the constructed approximation. As required coefficients are only computed on demand, the additional ones are responsible for the increased runtime. On the other hand, the improved quality of the approximation is beneficial for the computation of a solution, resulting in a reduction of necessary iteration steps.

At some critical points, even a small increase in accuracy results in a significant difference in runtime and memory consumption. This is the case when switching the accuracy between 2.5×10^{-5} and 1.0×10^{-5}. While the number of iterations differs by only one, overall runtime changes by more than a factor of five, and the amount of required memory increases more than 12-fold.

This is comparable to the significant increase in relative memory consumption as well as absolute runtime when simulating the *GIS Isolator* model. This indicates that the model is unfavorably structured for the implemented adaptive cross approximation method, resulting in an increased number of coefficients necessary to guarantee the default approximation accuracy.

5.6.4 gobem

Finally, gobem also shows promising improvement when using up to 8 cores, but afterwards falls short of both polopt0 variants as well as BETLdiel for all test models. Admittedly, in case of *GIS Isolator* the difference to BETLdiel is quite small.

Due to the fast multipole method, memory requirements in relation to model size are stable. Nevertheless, similar to BETLdiel, gobem shows a tendency for increased relative memory consumption for the *GIS Isolator* model, albeit not as pronounced.

Notably, gobem also slows down again for three of the test models when moving from 16 to 32 cores. Closer inspection shows, that this increase in cores dramatically increases the amount of time spent within the OpenMP threading library.

For the *Dielektrik* test model the fraction of CPU time spent within the OpenMP library increases from about 14% at 16 cores to almost 60% at 32 cores, as shown by Figure 5.8. Time spent in the other significant library *MKL*[10], a library use by gobem during active processing for efficient math routines such as matrix vector multiplication, as well as within gobem itself is reduced.

This is an indication for an imbalance of work load distribution, resulting in wasted computing capabilities due to some threads waiting for others to complete their work. A more detailed analysis confirms this by revealing that the overwhelming time spent within the OpenMP library is due to the execution of two functions used for synchronization, `__kmp_wait_sleep()` and `__kmp_x86_pause()`, which are responsible for over 56% of overall CPU time.

Indeed, Figure 5.9 clearly shows long intervals in which only a fraction of overall processing capabilities is used for productive computations, while the remaining threads spend their time within the OpenMP library, waiting for OpenMP synchronization or not working at all. Because of this and the additional management overhead incurred by increasing the number of threads, overall runtime actually increases again.

5.7 Conclusion

With increasing processing power, the simulation of more complex and detailed models becomes feasible and desirable. However, the amount of available memory does not increase at the same pace as the available computing capacity. Therefore, the quadratic memory requirements of polopt0 are problematic and render the simulation of very complex models unreasonable.

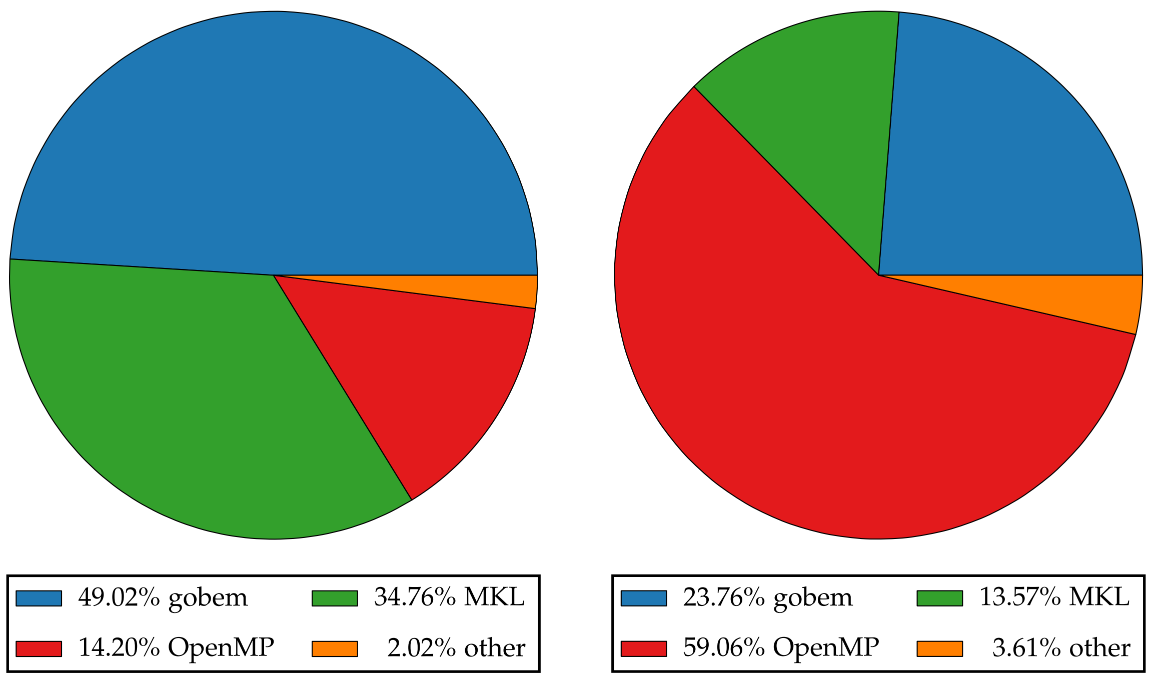
In that respect, the three alternative solvers polopt3, BETLdiel and gobem represent a significant enhancement over polopt0 as they implement methods to avoid constructing and storing the fully populated dense matrix that is the dominating factor for polopt0's memory requirements.

Both methods, the fast multipole method realized in polopt3 and gobem as well as the adaptive cross approximation method implemented in BETLdiel noticeably reduce the increase of memory consumption with increasing model complexity.

The polopt0 variant with localized matrix compression also helps reducing the growing relative memory consumption, but is not as effective as the other methods. On the other hand, due to the very localized changes necessary to implement the compression, the parallelization and favorably speedup behavior of polopt0 is maintained.

At the same time, the three other solvers currently show deficiencies regarding their parallelization. Polopt3 is the only alternative solver that is fully parallelized using MPI, making it possible to distribute it across multiple nodes on a compute cluster. However, parallel efficiency needs to be improved significantly to make its execution on a cluster reasonable.

[10]Intel Math Kernel Library



(a) 16 cores (b) 32 cores

Figure 5.8: Fraction of CPU time spent in libraries by gobem for model *Dielektrik*

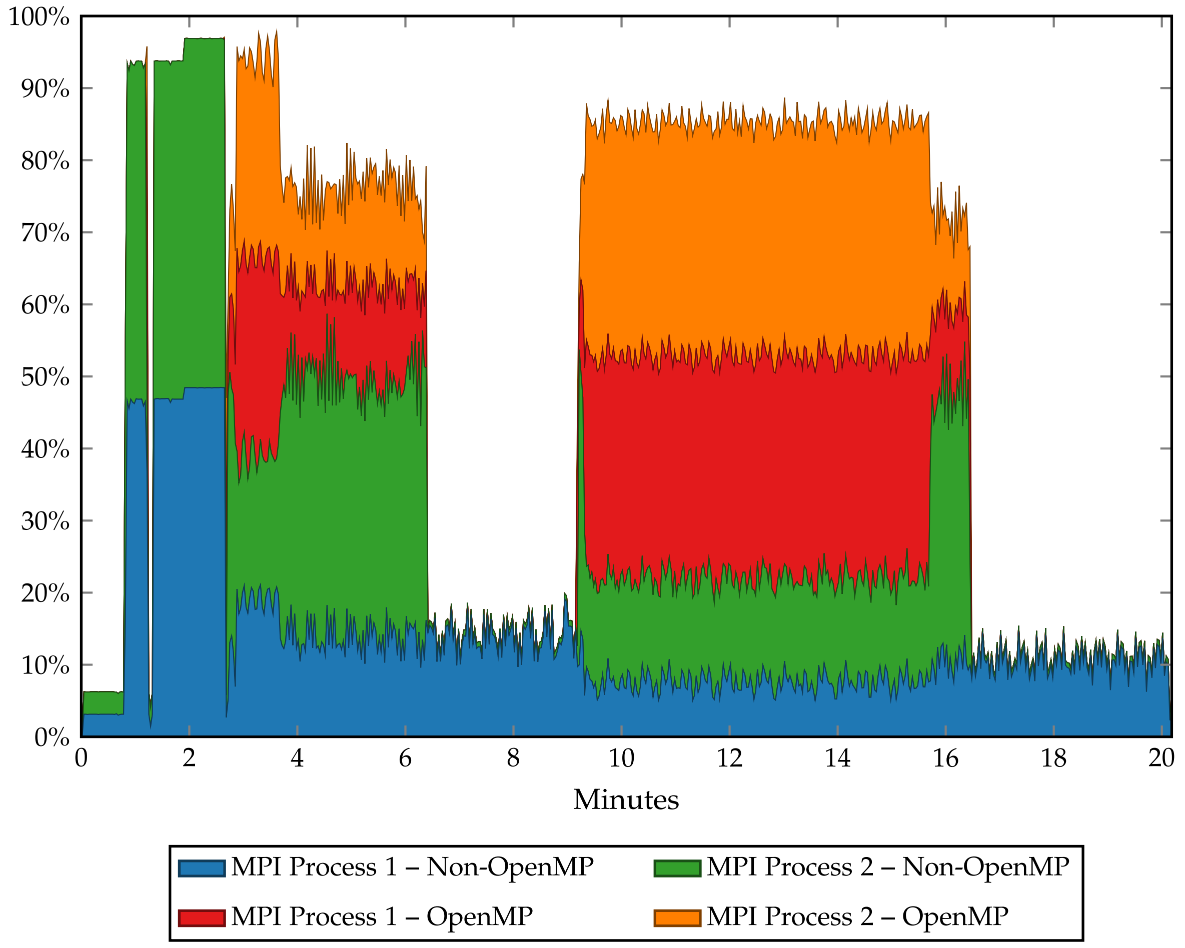


Figure 5.9: Gobem system utilization for model *Dielektrik* on 32 cores

Gobem employs a hybrid parallelization in which the number of MPI processes is determined by the domain decomposition during preprocessing. Again, this limits the possibility to distribute computational load among a large cluster to speed up computation, as the number of usable cluster nodes is limited by the number of domains, which in the current implementation typically is well below 10. Additionally, the shared-memory parallelization of the computation of a single domain has room for improvement.

Finally, BETLdiel shows speedup that – at least for some of the test models – is comparable to polopt0, but it is currently only usable on a single workstation, as its sole parallelization method is based on OpenMP.

All things considered, the difficulties to achieve consistently good parallel efficiency are a direct result of the increasingly complex data structures. By employing an extremely simple one – a fully populated dense matrix – polopt0 makes it very straightforward to parallelize the computation. By using more complex, often tree-like data structures to improve resource requirements and runtime performance, parallelization and especially efficient load-balancing is getting significantly more complex.

Additionally, to get best results, the specific parameters, such as the accuracy of the adaptive cross approximation method, have to be adapted for every model based on its specific characteristics. While it is possible to manually do so for a small number of selected models, it is infeasible for every engineer to properly understand all implications of those parameters and to know how to correctly set them by hand.

These implications include the accuracy of the result, overall runtime and memory consumption, which can vary significantly depending on the model and selected parameters, as demonstrated by BETLdiel. Without some kind of reliable prediction for runtime or memory requirements, it will be hard to efficiently allocate resources for computation.

Therefore, it will be necessary to develop an automatic parameter selection, based on fast analysis of the relevant model combined with know-how accumulated from similar models. To properly and reliably get this right though, a lot of time and resource consuming tests and experiments will be necessary.

All in all, this chapter illustrates the complexities that arise when replacing a well-established software package with a newly developed one. Even though the mathematical foundations that constitute the basis of the new applications are sound, implementing them in an efficient way takes time and effort.

If an existing application is known to be no longer sufficient in the foreseeable future and needs to be replaced as a whole, development has to start early. Otherwise one risks a situation in which no adequate and usable replacement exists and utilizing the existing application is no longer worthwhile due to excessive resource requirements.

Depending on the specific constraints, the usability of the existing application can be extended for some time by differently prioritizing resources, such as cutting back on runtime performance in favor of memory requirements. This strategy is demonstrated by means of the polopt0 variant featuring matrix compression. It tolerates an increased runtime due to additional processing capabilities used to perform data compression and decompression to achieve a reduction in the amount of memory required for simulation.

Chapter 6

Case 3 – Contingency Analysis

6.1 Background

The third and final type of application investigated in this thesis deals with power networks, specifically *contingency analysis*, i.e. the overall reliability and effects in case of disruptions of transmission lines for instance. If a transmission line is disrupted, e.g. due to a fallen tree, power is typically rerouted over other existing lines. However, great care has to be taken not to overload those other lines with the increased power, otherwise they will fail too.

Part of contingency analysis is to simulate outages of components of power networks, such as transmission lines or power generators, and verify that the resulting network and power transmission still adheres to all imposed limits. Performing analysis of this kind is essential to provide a stable power distribution and avoid outages.

It has to be noted that it is not sufficient to perform this kind of simulation only once. Every time the characteristics of the power network changes, it has to be performed again. These changes include shifts in power demand or generation due to changes in weather conditions or constructions at the power network itself.

The importance of this kind of analysis is probably best illustrated by a real-life incident demonstrating the risk of not performing the analysis thoroughly enough respectively not considering the results carefully enough. In November 2006, a planned manual disconnection of a heavily used transmission line in Northern Germany resulted in the overload and subsequent shutdown of another transmission line. Subsequently, in a cascading effect, the European power grid split into three separate areas with significant power imbalances, resulting in a power outage for more than 15 million European households. The detailed report of this outage was published by the Union for the Co-ordination of Transmission of Electricity (UCTE) [80].

For larger power networks the number of possible combinations of outages can be extremely high and it is subsequently infeasible to process all of them thoroughly. Because of this, a preprocessing called *contingency selection* or *contingency screening* is performed. Its purpose is to eliminate most of the trivial outages, which are guaranteed to pose no problems and only retain the critical ones. A detailed introduction into this topic can be found in *Power Generation Operation and Control* by Wood and Wollenberg [84].

As before, this thesis will concentrate on performance and scalability issues, not the underlying algorithms and power network modeling. Previous work done in this area includes Müller et al. [62] and Trinitis et al. [78, 79].

6.2 Performance Characteristics and Analysis

6.2.1 Parallelization

As discussed above, the purpose of the simulation basically is to analyze different combinations of possible outages to ensure safe operation of the modeled power network, even in the event of disruptions.

The most obvious way to parallelize this simulation is to analyze different outages in parallel, which is done using OpenMP. The following analysis is based on an implementation, that should be considered experimental and which is not used for production. Its main purpose is to study principal behavior and effects and potential tuning possibilities on different architectures.

Because of this, the parallelization is realized in a very straightforward way and data structures that are modified during processing are duplicated to provide a private copy for every thread. While this makes parallelization itself easier, it increases memory consumption and also affects performance, as will be shown below.

6.2.2 Memory Requirements

As often, memory requirements are of interest when analyzing software and especially when considering to port it to accelerators, as they typically feature only a limited amount of memory compared with generic workstations or servers.

As depicted in Figure 6.1, memory consumption of the application grows linearly with increasing number of threads. This is, at least partially, a direct result of the currently implemented parallelization scheme, which in doubt duplicates data structures to avoid conflicting write access by different threads. By using more specialized and suitable data structures the memory growth rate could be reduced significantly.

If one extrapolates to the 240 threads available on the Xeon Phi, memory consumption raises to over 8 GiB, making it impossible to execute the application on it – particularly with regard to additional memory required by the operating system running on the Xeon Phi.

Because of this, a modified variant of the application is used, in which the precision of the simulation was reduced to decrease memory consumption. Figure 6.1 also shows this variant and memory clearly still increases linearly with the number of threads, but overall rise is slower. This variant only consumes about 5.5 GiB for 240 threads, thus making it possible to run and evaluate it on the Xeon Phi.

This modification does not have a significant impact on the overall behavior of the application regarding runtime. Consequently, if not stated otherwise, the modified variant is used in the following to make it easier to compare the results between the Westmere-EX and the Xeon Phi system.

6.2.3 Runtime Performance

Table 6.1 shows the runtimes of the simulation on both architectures – Westmere-EX and Xeon Phi. The table shows both the runtime for a single thread, as well as the fastest overall runtime when using multiple threads and the number of threads with

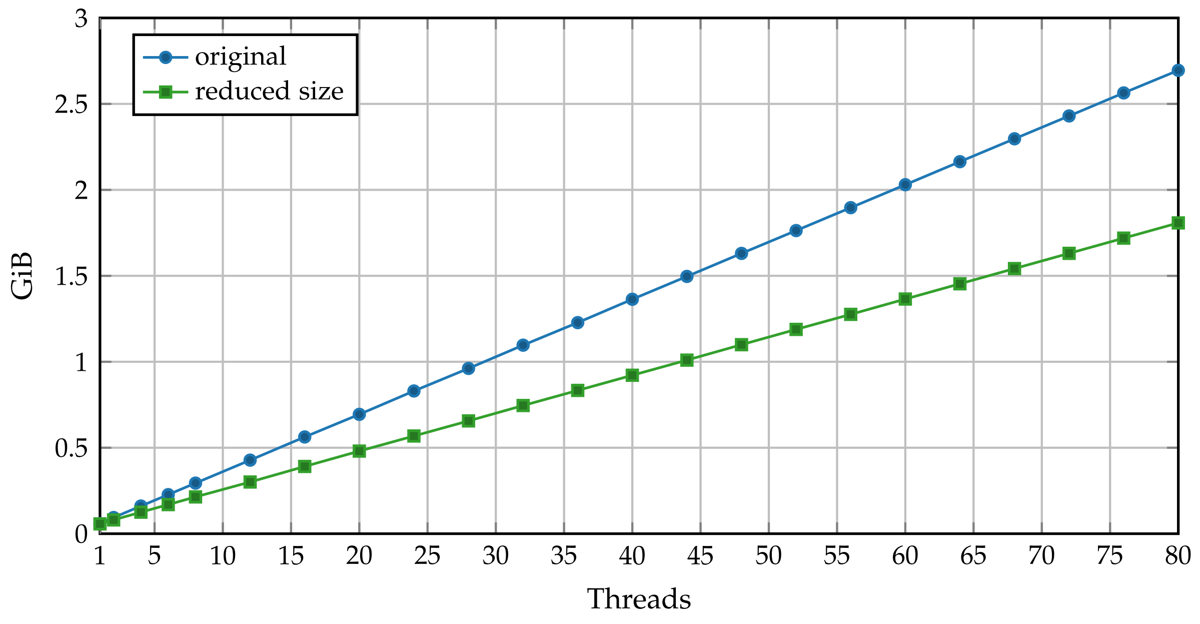


Figure 6.1: Memory requirements

	Westmere-EX		**Xeon Phi**	
	single thread	best result (32)	single thread	best result (60)
Screening	16.71	1.39	188.82	21.68
Analysis	34.34	2.02	412.57	12.80
Total	51.15	3.51	602.17	35.23

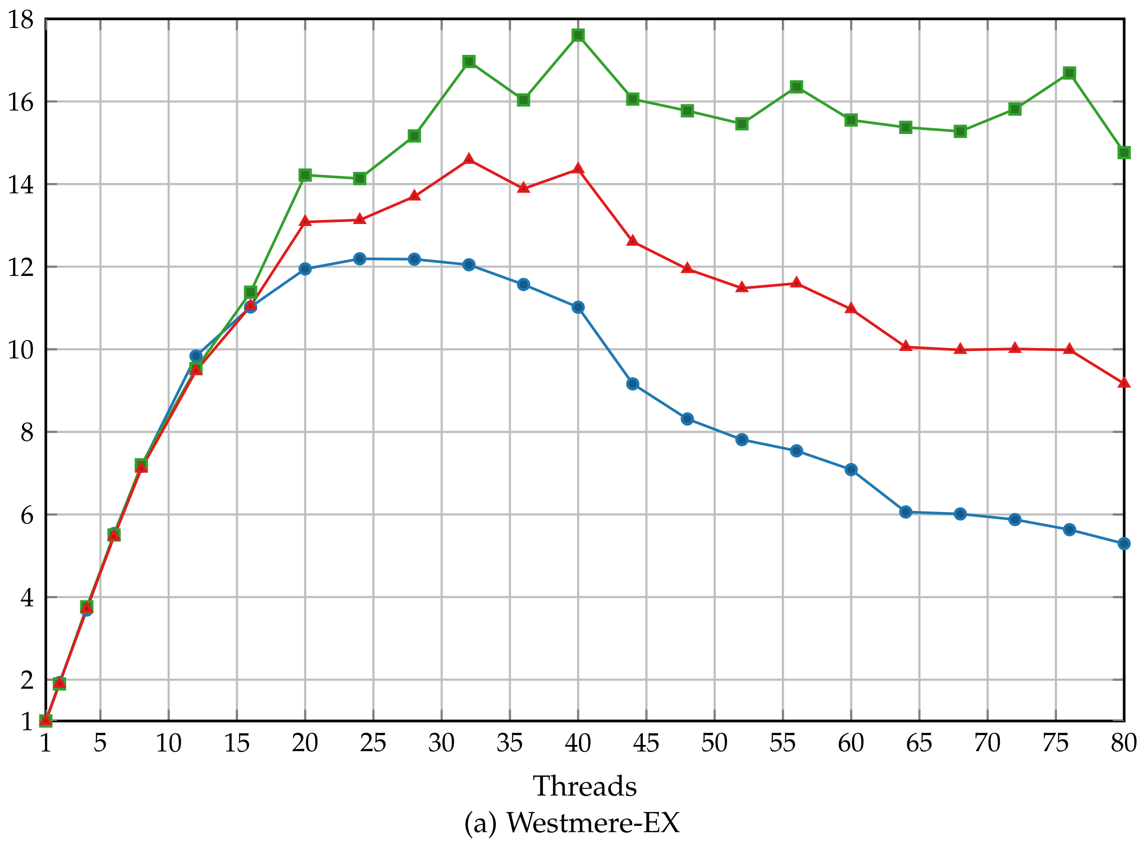
Table 6.1: Runtimes in seconds

which this was achieved. The total runtime is dominated by the two phases *screening* and *analysis* which are responsible for more than 99% of consumed time in case of sequential execution. Nevertheless, the simulation software also includes a small amount of additional handling and management, which is why the total runtime listed in the table is slightly more than the sum of the two phases.

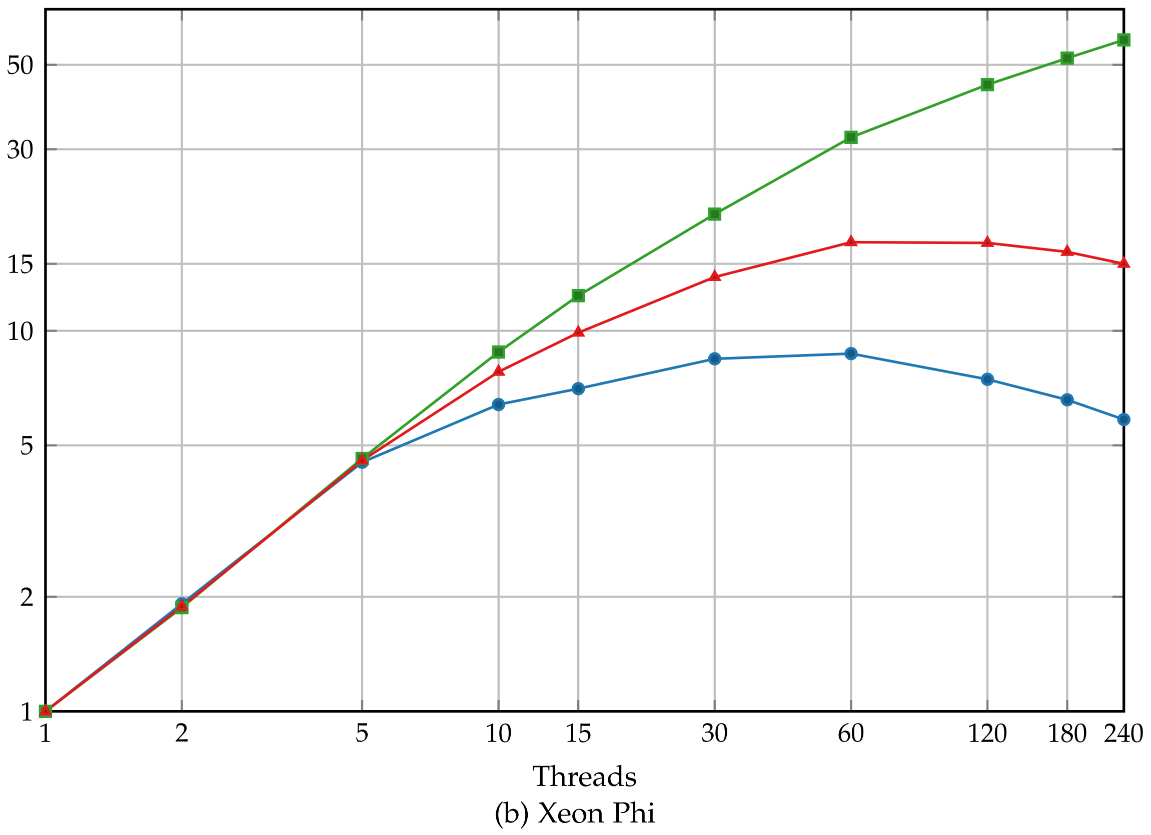
Both systems benefit from multiple threads, but the Xeon Phi system is not able to surpass the performance of the Westmere-EX system. The lower single core performance is a direct consequence of the much simpler individual processor core of the Xeon Phi and the simulation software does not scale well enough to compensate.

The single core performance of the Xeon Phi system is more than 170 times slower than the fastest runtime on the Westmere-EX system which is achieved when using 32 threads. To have a faster overall runtime on Xeon Phi, the speedup would have to be more than this factor of 170. But, as visible in Figure 6.2 showing the overall behavior of the simulation when using multiple threads in more detail, the maximum overall speedup on Xeon Phi is barely over 17.

Figure 6.2 also shows that the two phases scale quite differently. When using only a few threads – up to 12 on Westmere-EX or 5 on Xeon Phi, respectively, – both phases,



(a) Westmere-EX



(b) Xeon Phi

Figure 6.2: Speedup of different phases within simulation software

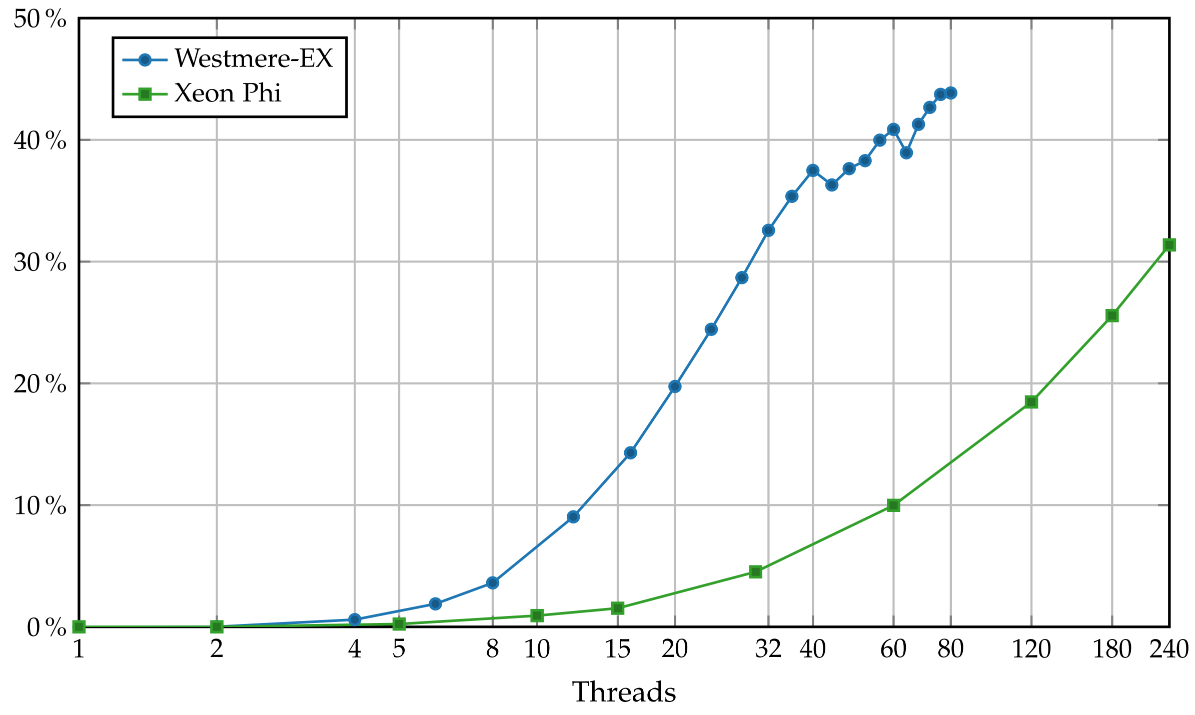


Figure 6.3: OpenMP overhead during contingency screening

and therefore overall runtime, scale quite well. With further increasing number of threads though, the analysis phase scales noticeably better.

The screening phase on the other hand shows a very different behavior. Its speedup on Westmere-EX peaks at 24 threads and drastically worsens with increasing number of threads – from a little over 12 to below 6. On Xeon Phi the peak performance with a speedup of a bit over 8 is reached at 60 threads – the number of available physical cores – and worsens with further increasing number of threads, similar to the Westmere-EX system. However, performance does not suffer as extreme on Xeon Phi as it does on Westmere-EX and the speedup merely drops to about 6.

One reason for the performance decrease of the screening phase is the dramatic increase of overhead caused by OpenMP. As visible in Figure 6.3, the overhead increases from virtually zero at the beginning to over 30%, respectively 40%. All in all this means that more than a third of the time spend within the screening phase is needed by OpenMP to organize and set up the different threads – mainly to duplicate large data structures to provide a private copy for every thread.

On Westmere-EX the speedup of the analysis phase peaks with almost 18 at 40 threads and mostly stagnates a little lower afterwards. On Xeon Phi, however, the analysis phase continues to improve up to 240 threads. Still, overall speedup also starts to decrease again when using a large number of threads on Xeon Phi, because of the decreasing performance of the screening phase.

This could very roughly be compared to Amdahl's Law [3], but instead of sequential and parallelized phases, we have phases with varying parallel efficiency. Even though the analysis phase shows a maximum speedup of over 50 on Xeon Phi when using all 240 logical cores, overall speedup merely reaches 15, as it is slowed down by phases with less parallel efficiency.

6.3 Optimization Possibilities

6.3.1 Pinning

As always when using multiple threads, there is the question about distributing them among physical resources. The easiest solution is to let the operating system decide how to place threads on physical cores.

However, Klug et al. [51] showed that this often results in unpredictable and sub-optimal performance behavior. Instead, optimal and predictable performance is achieved by strategically placing threads on appropriate cores. This optimal placement – or pinning strategy – depends on the application and architecture characteristics and can differ significantly between different applications.

Figures 6.4 and 6.5 sketch the physical layout of the Westmere-EX system and a Xeon Phi card, respectively. Both layouts list the available logical – i.e. Hyper-Threading – cores in the order as they are enumerated by the operating system.

The logical cores are then grouped to represent the underlying physical core 2 logical cores per physical core on Westmere-EX and 4 logical cores per physical core on Xeon Phi. The fact that logical core 0 is actually located on the last physical core on the Xeon Phi is a technical detail that is of no further concern at this point.

Finally, the layout also includes the available last level cache, which is 24 MiB per socket on Westmere-EX and 512 KiB per physical core on Xeon Phi, resulting in a total of 96 MiB respectively 30 MiB.

Here, three pinning strategies were evaluated: *scatter*, *compact-socket* and *compact-core*.

scatter This strategy distributes the threads as evenly as possible across physical resources, i.e. at first across sockets, then across physical cores and finally across logical cores. When using 8 threads on the Westmere-EX system, two threads will be placed on each of the 4 available sockets, i.e on processor cores 0, 1, 10, 11, 20, 21, 30 and 31. On the Xeon Phi cores 1, 5, 9, 13, 17, 21, 25 and 29 would be used, i.e. one physical core per thread.

compact-socket Here threads are placed on one physical core after the other. In case of the Westmere-EX system, that would result in all 8 threads being placed on a single socket, namely on cores 0-7. As the Xeon Phi only has a single socket, this strategy results in the same thread placement as *scatter* and is therefore skipped on evaluation on Xeon Phi.

compact-core This is a more extreme variant of the *compact-socket* strategy. Instead of placing threads on one physical core after the other, logical cores are used. When again using 8 threads, cores 0, 40, 1, 41, 2, 42, 3 and 43 would be used on the Westmere-EX and cores 1-9 on Xeon Phi.

Figure 6.6 shows the speedup of the main simulation subroutine for different pinning strategies on both Westmere-EX and Xeon Phi. To highlight the relevant differences, only the speedup up to 60 threads is shown. The lowest absolute runtime for all pinning strategies on both systems as well as the respective number of threads is listed in Table 6.2.

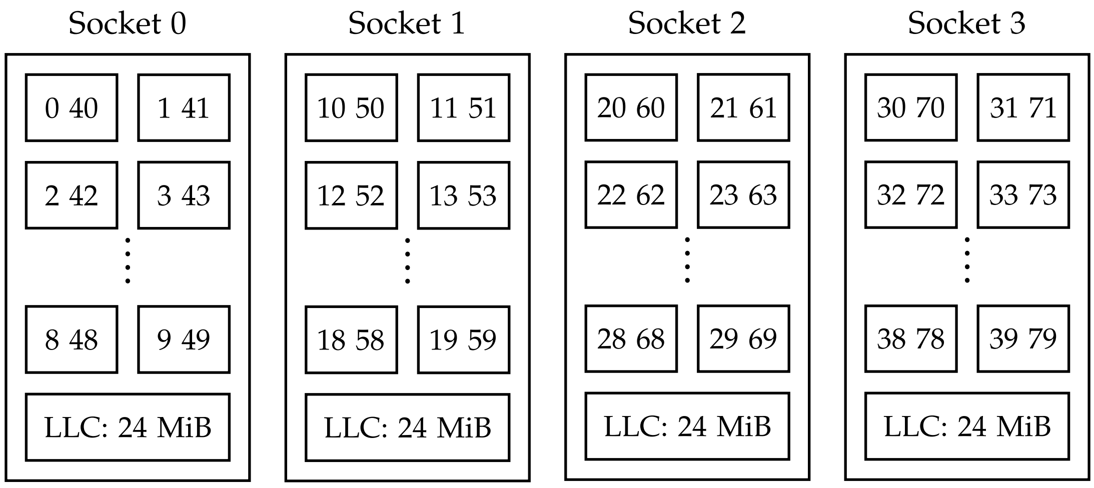


Figure 6.4: Processor sockets and cores layout on Westmere-EX

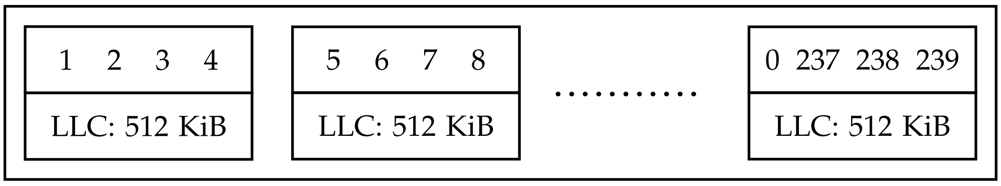


Figure 6.5: Processor core layout on Xeon Phi

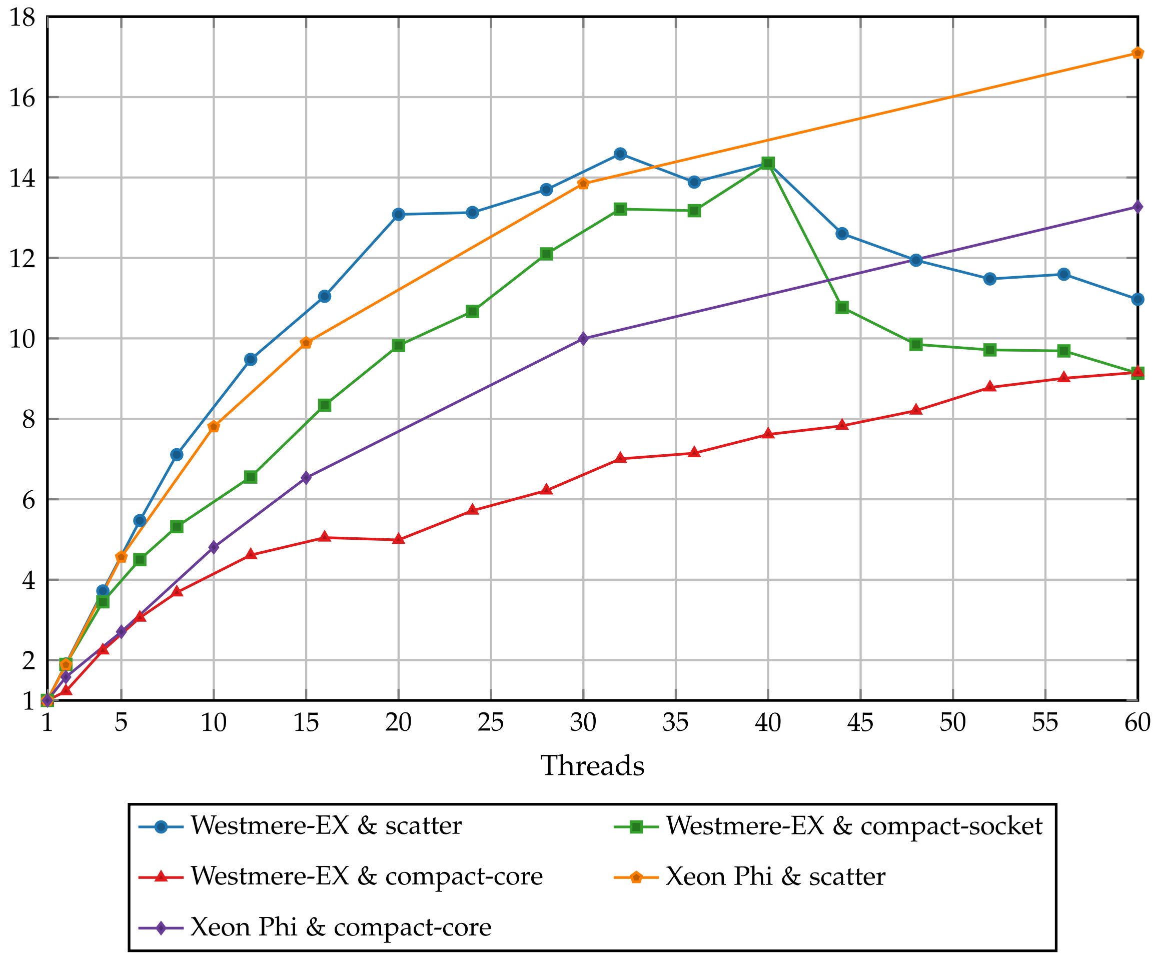


Figure 6.6: Speedup with different pinning strategies

	Westmere-EX		Xeon Phi	
	best runtime	# threads	best runtime	# threads
scatter	3.51	32	35.23	60
compact-socket	3.56	40	–	–
compact-core	5.28	76	38.53	120

Table 6.2: Runtimes with different pinning strategies in seconds

The *scatter* strategy results in best performance on both Westmere-EX and Xeon Phi and, on Westmere-EX, when using 40 threads performance is identical for both *scatter* and *compact-socket*, as in this case every available physical core is handling one thread. When using a lower or higher number of threads though, performance differs. Finally, *compact-core* is significantly slower on both architectures.

It should be noted, that overall best runtime on the Westmere-EX system is actually achieved by employing 32 threads and pinning them using the *scatter* strategy instead of using all available physical cores with 40 threads.

When multiple threads load large amounts of data from memory for processing they inevitably cause the eviction of other threads' data. By deliberately using fewer threads and evenly distributing them across physical resources each thread is able to use a larger share of the last level cache for its own data, thereby increasing cache efficiency. This interaction is also discussed in the following section covering cache optimization.

6.3.2 Cache Optimization

As detailed before, parallelization of this contingency analysis simulation is done by simulating different outages in parallel, while each outage is computed sequentially. This indicates that the threads mostly run independently from each other with almost no interaction or communication between them.

Figure 6.7 illustrates the rising rate of cache misses when increasing the number of threads on the Westmere-EX system. This rise can be observed for all three pinning strategies discussed before, but the rate of increase is very different.

When using one of the *compact* pinning strategies, the cache misses rate raises very fast to over 30% when using 8 threads. While the miss rate stays at this level for *compact-socket* for increasing threads, it even increases to over 50% for *compact-core*.

The *scatter* strategy on the other hand shows a significantly less rapid increase of the cache misses rate. Instead, it is steadily increasing and reaching its maximum with 40 threads. All physical cores have their private L1 and L2 caches, while all cores on a socket share a single L3 cache or LLC. The *scatter* pinning strategy maximizes the share of the last level processor cache available for each individual thread, which is why it results in better performance.

To further investigate this, a cache optimized version of the application was evaluated. The cache optimization consisted in rearranging some internal data structures to increase cache line reuse, as discussed in Section 2.4.3. Figure 6.7 also shows the results

	Westmere-EX		Xeon Phi	
	single thread	best runtime	single thread	best runtime
original	51.15	3.51 (32)	602.17	35.23 (60)
cache-opt	50.73	3.34 (40)	595.85	34.93 (60)

Table 6.3: Runtimes with and without cache optimization in seconds

of the evaluation of cache misses for the cache optimized variant. The cache misses rate was noticeably reduced by the cache optimization and is effective for all three pinning strategies.

An effect on runtime performance is also measurable and shown by Table 6.3 as well as by Figure 6.8, which also illustrates the correlation of the cache optimization effect and the pinning strategy. For a small number of threads, the cache optimization does not make a significant difference – even slowing the application down slightly – as cache misses are not an issue at this point. With increasing number of threads, however, a clear trend emerges, especially when comparing Figures 6.7 and 6.8. As more cache misses occur, the effect of cache optimization on runtime performance becomes more pronounced.

With the *compact* strategies, the cache misses rate increases very rapidly, which is why the improvement due to cache optimization already peaks at 12 threads and then slowly decreases a little until up to 40 threads. On the other hand, the cache misses rate increases much slower with the *scatter* strategy – due to more available processor cache per thread –, which is why the cache optimization's gain gets visible only gradually and keeps improving until up to 40 threads.

All in all, the effect of the cache optimization is definitely measurable on both architectures, proving the influence of using different data structures and access patterns on runtime performance.

It has to be noted, however, that according to the technical specification [45], the hardware prefetcher built into the processor may also influence the cache misses event count. Data the hardware prefetcher anticipates to be required in the future and which is subsequently automatically prefetched from memory into the processor cache may count as a cache miss, even when it is not used afterwards and therefore performance was not impeded by waiting for a data transfer from memory.

Also, even if the data is actually used, performance is unhindered if the prefetch is already completed by the time the data is processed. The exact interaction is highly hardware implementation specific, which is why event counts from different architectures can not be directly compared with each other.

6.3.3 Prefetching

As already discussed in Section 2.3.4, the Xeon Phi does not feature memory prefetching hardware and consequently does not automatically try to preload data from memory that is anticipated to be needed in the future. Instead, the application itself

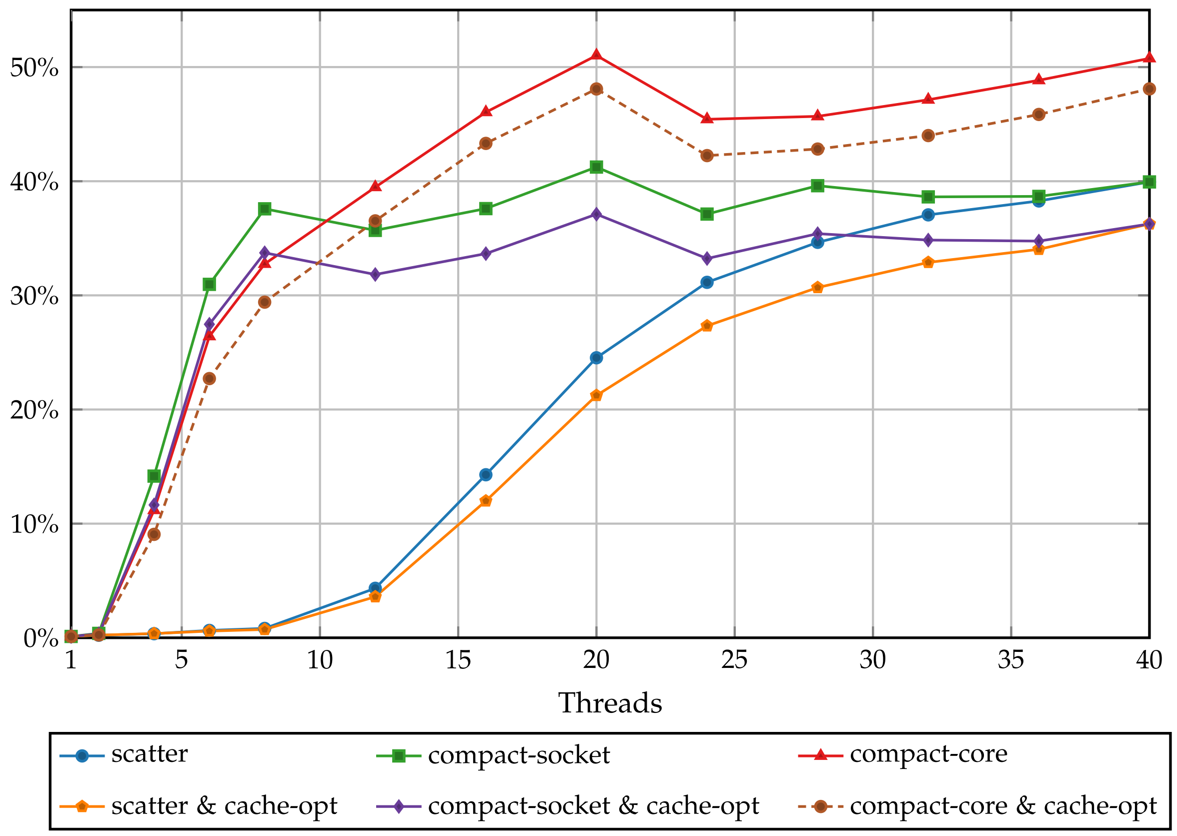


Figure 6.7: Cache miss rate for the original and the cache optimized variant on Westmere-EX

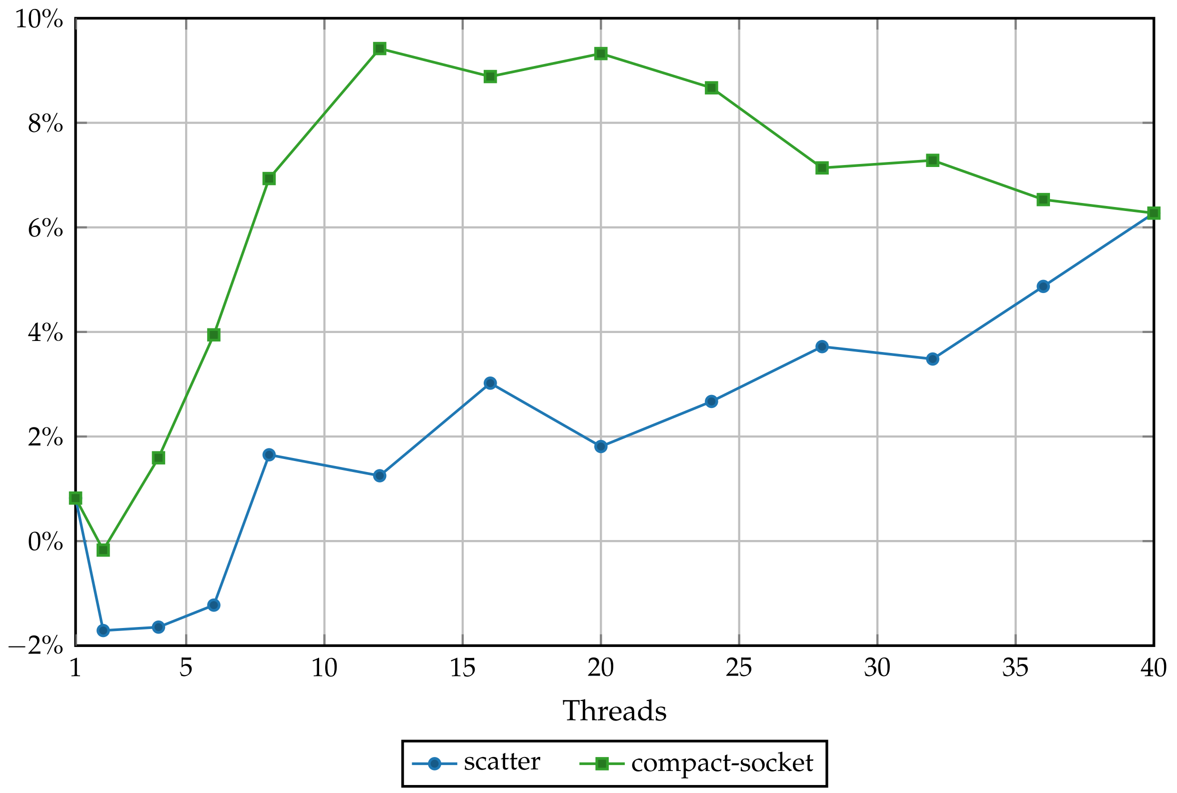


Figure 6.8: Relative improvement with cache optimization for different pinning strategies on Westmere-EX

	single thread	best runtime
default	602.17	35.23 (60)
no-prefetch	567.71	33.94 (60)

Table 6.4: Runtimes in seconds with and without prefetching instructions on Xeon Phi

is responsible for executing prefetching instruction as needed to avoid unnecessary waiting time while data is loaded from memory.

By default, the Intel compiler for Xeon Phi automatically inserts those explicit prefetching instructions where it sees fit and expects it to have a beneficial effect. A compiler flag is provided to disable this automatic instruction insertion.

Table 6.4 lists the overall simulation runtime for a single thread as well as the best runtime using multiple threads for both options. The lower runtime achieved without automatically inserted prefetching instructions demonstrates that they are actually hurting performance and shows that the heuristics used by the compiler to decide where to initiate prefetching are not suitable for every type of application.

Instead, to obtain optimal performance, manual analysis and optimization would be necessary. This is a time consuming process and would presumably have to be repeated again every time the architecture changes, as it is highly sensitive to processor performance and memory bandwidth and latency. Also, even subtle changes to the application may invalidate this manual optimization, making it necessary to start from the beginning.

On the other hand, the hardware prefetcher featured in the general purpose processors in the Westmere-EX system, can automatically detect memory access patterns at runtime, making manual optimization to the precise processor architecture and system hardware configuration less important.

6.4 Conclusion

The parallelization implemented in this case is not very efficient as illustrated by the maximum achieved speedup. Duplicating large data structures in memory will always be problematic, as it increases the workload with increasing number of used threads, constricting the positive effects of using multiple threads in the first place. Instead, one should implement more appropriate data structures which support shared access by multiple threads, thereby reducing the need to duplicate data for every thread.

Nevertheless, the implementation in its current form is useful for demonstrating common effects observed when handling parallelized applications. By means of a contingency analysis software, this chapter showed how performance is influenced and tunable by various parameters – from software design choices to runtime environment.

Of those, cache optimization is the most intrusive one, as it frequently touches the core data structures of an application, which means that existing software typically has to be adapted and restructured to properly apply cache optimization.

For data structures that are used at many different parts of a large application this is a time consuming process and requires an in-depth understanding of the application, the underlying algorithms and the hardware architecture, i.e. how processors access memory and load data from it, as well as specialized profiling tools to identify and highlight bottlenecks.

Determination of the optimal number of threads and their pinning to available processor cores on the other hand is done at runtime after code development itself and depends on the actual hardware that is used to execute an application. As detailed in this chapter, using as many threads as available processor cores does not result in best performance for all applications.

Instead, for specific applications deliberately using fewer threads and strategically pinning them to appropriate cores results in increased overall performance, as is the case for the simulation software covered in this chapter. Here, performance is limited by the amount of data that can be transferred from memory in a given time. By using fewer threads, the individual threads are able to keep more of their private data within the limited processor caches. When using a higher number of threads, data has to be transferred from memory more often, undoing the positive effect of processing more data sets in parallel.

Additionally, different phases of an application have to be accounted for and the respective number of threads used for processing needs to be adapted for best performance. When executed on the Westmere-EX system using a fixed number of threads for the complete simulation fastest runtime for the exemplary contingency analysis simulation was achieved with 32 threads.

On the other hand, Figure 6.2a indicates that using different number of threads per phase results in better overall performance, namely 24 threads for screening and 40 threads to process the analysis phase. The same is true for the Xeon Phi system, where using 60 threads for screening and 240 threads for contingency analysis results in best performance for the respective phase.

Closely related to the number of threads is the question of how to assign them to the available physical resources. In this case study, the *scatter* pinning strategy resulted in the best performance, as the individual threads do not have to communicate and synchronize a lot, but instead independently process their share of the overall workload.

The situation is different though, if the threads require more fine grained synchronization or communication. An application implementing a producer-consumer technique, in which some threads prepare or create data items which are in turn processed by other threads, will benefit from being able to directly exchange data between different threads using the processor cache instead of accessing main memory. This is facilitated by placing corresponding producer and consumer threads on the same processor socket by using the *compact-socket* pinning strategy.

Performing an exhaustive search for the best combination of number of threads and pinning strategy for each phase of an application is very cumbersome to do manually. Instead, automated tools, such as *autopin* [51] can be used to perform this search in an automated and systematic way.

This chapter illustrates how software development decisions and runtime environment parameters influence each other. For best performance, the optimal choice of runtime parameters, such as number of threads, pinning or even suitable architecture, depends on the algorithms and patterns implemented in the application. At the same time, performance can be improved by incorporating known hardware characteristics during software development. Understanding the mutual interactions helps in writing and adapting high performance software as well as in identifying appropriate runtime parameters.

Chapter 7

Conclusion

In the course of the previous three chapters of this thesis, several different performance optimization and improvement strategies, techniques and approaches were presented, applied and evaluated. This was done using a number of exemplary real world simulation software packages covering three areas of high voltage engineering, provided by industry partner ABB.

The discussed strategies and findings can roughly be classified into localized and global modifications, as well as parallelization and runtime optimization and, finally, specifics related to Xeon Phi.

7.1 Strategic Localized Modifications

When optimizing the performance of an application, one starts by creating a profile of the application to get a general idea about potential bottlenecks and the distribution of work amongst the various different parts of the application.

Using this profile, one can then start to investigate the application more extensively and identify individual functions that require an excessive amount of time to execute. If possible, these functions subsequently are accelerated by adapting data structures or algorithms or reusing results whenever possible.

By systematically identifying and reducing performance bottlenecks, this optimization technique allows for significant overall performance improvement while being very cost-efficient at the same time. This process can be carried out step by step, until the application meets the requirements. Additionally, as the fundamental behavior of the application does not change extensively, the time required to properly conduct quality assurance is reduced.

In Chapter 4 this strategy is applied to a thermal simulation software. At first, the application is adapted to be able to reliably handle the increasingly complex models arising in industry. Similar to the divide and conquer approach, this is done by splitting the complex problem into two smaller, coupled problems, which are then solved iteratively.

While this partitioning makes it possible to simulate more complex models in the first place, it also significantly shifts the emphasis of work within the application, as the separated problems themselves have to be simulated repeatedly. To compensate for the increased work load, the simulation of the individual subproblems has to be speeded up.

In Section 4.4 various techniques to do so are presented. Independent of the characteristics of the processor used, the fastest computation is the one that does not have to be performed at all. Consequently, previously computed results should be reused whenever possible, or at least as long as recomputing them is more costly than storing and retrieving them. Section 4.4.1 exemplarily shows how this can be realized by taking advantage of specific characteristics of the underlying algorithm and how a small and localized alteration during computation can increase performance by more than a factor of four.

Another peculiarity often encountered in established applications are data structures that do not scale well with increasing number of data items and are not very cache efficient at the same time, such as linked lists. While their use is very straightforward, they are notoriously slow when using them to repeatedly look up specific items in a large data set.

Also, successively loading data from memory only to learn the position of the next data item increases pressure on the memory subsystem. Data items which are scattered in memory also decrease the efficiency of the processor caches, as often only a few of them are closely grouped together, resulting in an unnecessary high number of cache line transfers and evictions. Additionally, computational performance is increasing faster than memory access speed, leading to increasing memory latency and therefore reduced performance when using such data structures.

Section 4.4.2 demonstrates how performance can be improved by using more efficient data structures as caches to speed up data item retrieval at performance critical points without having to restructure the complete application. Similarly, in Section 4.4.3 the algorithm to access such a linked data structure is adapted at a very localized and confined level, without tampering with the data structure itself. The effects of reorganizing data structures for increased cache efficiency are presented in Section 6.3.2.

However, data structures do not only affect performance with respect to time. Instead, they also strongly determine the amount of memory that is required to perform a certain operation and are therefore crucial to the question if an application can be executed at all on a given computer system. Depending on the characteristics of an application, it can be beneficial to store data in a compressed form and decompress it on-the-fly during access. While this sacrifices processing power due to increased computations, it enables the application to run on smaller systems or to process larger input data sets.

All in all, using appropriate data structures is a critical point for every application. Especially when being repeatedly used during performance critical computations, the utilized data structures should be cache efficient and exhibit both low time and space complexity to avoid performance bottlenecks with increasing input data sets in the future or the growing discrepancy between computational performance and memory access latency, i.e. the memory wall.

At the same time, code with highly varying branches should be avoided. Modern general purpose processors highly depend on efficiently exploiting their instruction pipeline to achieve best performance. However, highly varying branches impede the automatic branch prediction and lead to reduced performance due to pipeline stalls. Section 4.4.4 illustrates the adverse effect of having a single branching point that depends on a highly varying condition. Splitting it up allowed the branch predictor to more reliably predict the destination of a branch and subsequently increased overall performance noticeably.

To further improve overall efficiency dynamic code generation can be employed to reduce scattered memory accesses and branch mispredictions. Section 4.5 demonstrates how the repeated traversal of an expression tree can be avoided by generating and executing code to obtain the same result within much shorter time.

Generally, code generation can be realized in very different ways, all of which have advantages and disadvantages. Consequently, the correct choice for the task at hand highly depends on the specific requirements and characteristics of the application, such as the number of times the generated code is executed. Generally, the higher the execution count, the more additional effort during code generation to produce the best and fastest code possible is worthwhile.

7.2 Global Modifications

Depending on the characteristics of an application small and localized changes are not enough to sufficiently improve performance or reduce resource requirements. Instead, more extensive and fundamental changes have to be performed, eventually rewriting the application as a whole. This scenario is presented in Chapter 5.

The original and established electrostatic simulation solver exhibits a quadratic dependency between the size of the model to be simulated and the memory required to perform the simulation, that is, if the model size doubles, the memory requirements increase fourfold. Even though the application is parallelized using MPI and the data structures can be distributed over multiple nodes within a cluster, the increasingly high memory consumption is getting critical.

On the one hand, the steadily increasing computing capabilities make it feasible to simulate very large and detailed models within an acceptable time frame. One the other hand, the amount of memory that can be cost-effectively installed into a single node is not increasing fast enough to compensate for the quadratic growth, creating an incentive to develop a more memory efficient alternative.

By now, several mathematical methods for improving the simplistic implementation are known and published. As illustrated in Section 5.2, these methods feature better theoretical properties regarding both memory consumption and runtime. While this raises hope for a cost-effective and fast implementation of an alternative simulation solver, the observations in Chapter 5 prove otherwise.

Three different simulation solvers, each implementing one of the two improved methods mentioned there, were evaluated. Both methods are based on using approximations to speed up computation and reduce memory consumption, but at the same time add a dependency on the characteristics of the simulated model, instead of only on model size. Because of this, the exact amount of memory required to simulate a specific model is no longer known beforehand and fluctuates significantly.

Overall, all solvers show an improved memory requirement as memory no longer grows quadratically with model size. But the uncertainty regarding the exact required amount of memory complicates resource allocation.

Generally, the exemplary study highlights difficulties arising when applying large scale changes or reimplementing an application based on newer methods. Even if the theoretical foundations are sound, implementing them in an efficient way takes time and effort and needs to be started early enough to be completed and tested before the existing application can no longer be reasonably utilized.

7.3 Parallelization

Parallelization is a powerful and established technique to speed up an application. At the same time, it is a necessary technique to efficiently utilize the growing number of processor cores and to benefit from the current trend in processor design.

In principle, parallelization can be implemented in both a localized as well as a global manner. But, as pointed out by Amdahl's Law, when parallelizing only a part of an application, the achievable speedup is limited by the remaining sequential portion independently of the number of processor cores available.

Gustafson [36] does relativize Amdahl's Law to some extend by considering input data sets whose size increases with parallel computing performance. Still, if the objective of the parallelization is to reduce the overall runtime of an application, the sequential portion should be kept as small as possible and should also not increase with increasing input data size.

Even if parallelization is applied almost globally, its efficiency depends on various influencing factors. A significant one of them is the efficiency of work load balancing. If the computation is not evenly distributed across physical resources, parallel efficiency is reduced as processing power is wasted by idle threads or processes waiting for the other ones to finish.

Increasingly complex methods and data structures complicate efficient work load balancing as investigated in Chapter 5 by means of various electrostatic simulation solvers based on different theoretical methods. The polopt0 solver as well as its variant featuring matrix compression both utilize a very simple and straightforward data structure: a fully populated dense matrix. While this is problematic with respect to memory consumption, it also allows for a very straightforward and efficient work load distribution. On the other hand, both polopt3 and gobem, implementing the fast multipole method, have work load balancing issues.

The parallel efficiency of the contingency analysis application considered in Chapter 6 is also impeded by the utilized data structures and their handling. While load balancing is not a big issue there, the data structures themselves are not well suited and need to be duplicated for every parallel thread to allow for private modifications during processing. Here, a more efficient data structure supporting shared access to data would reduce the overall amount of memory required and increase performance due to a reduced number of cache misses.

All in all, parallel efficiency heavily depends on proper work load balancing and adequate data structures. This is a common area of conflict as new theoretical methods and theories typically get more complex and therefore harder to efficiently implement. This leads to conflicting optimization objectives when potential changes improve memory and sequential efficiency on the one hand, but significantly handicap parallelization on the other hand, as was the case in Chapter 5. In such situations careful consideration is necessary to prioritize the objectives.

7.4 Runtime Optimization

The last step after tuning an application on an algorithmic and data structure level is to optimize the runtime performance itself. The naive expectation would be that a parallelized application runs fastest, when all available resources are used, be it processor cores in a shared memory system or nodes in a cluster.

However, depending on the characteristic properties of the application, a lower number of parallel threads or processes will actually be more efficient and subsequently faster. This fact is illustrated by the gobem solver evaluated in Chapter 5, as well as the contingency analysis application from Chapter 6.

This counterintuitive behavior can be the result of many different effects. If work load balancing is inefficient, additional threads or processes do not noticeably contribute as they only get a marginal amount of work but need to be considered in all synchronization operations. For instance, in Section 5.6.4 gobem was shown to exhibit a noticeable work load imbalance and to spent considerable time handling synchronization of running threads.

Even if the work load is balanced, the running threads can negatively influence and slow each other down. As previously discussed, the last level cache of a processor is typically shared among all physical cores. When processing data from memory, all threads basically compete over this shared resource and may cause the eviction of cache lines used by another thread, potentially decreasing cache efficiency and reducing overall performance. This effect is demonstrated very well in Section 6.3.2 by the significant increase in cache misses when executing the contingency analysis application with an increasing number of threads.

The open question when using fewer threads than processor cores available on a shared memory system is how to allocate those threads to physical cores. The answer to this depends on the specific characteristics of the application as well as the computer system it is executed on and can therefore not be given generically.

If the individual threads slow each other down due to competing over the shared processor cache, they should generally be scattered as evenly as possible across all available sockets, to maximize the available processor cache per thread. This scenario was acted out in Section 6.3.1 using the contingency analysis application.

Nevertheless, other settings are possible as well. A group of threads frequently communicating with each other benefit from being able to exchange data directly via the shared processor cache, instead of having to transfer it between different sockets.

To complicate things further, the characteristic behavior of an application may change repeatedly during execution, requiring a change in the number of threads employed or a reallocation of the involved threads to get optimal performance. Conducting an exhaustive evaluation of all possible options can be very time consuming, especially when done manually, so using an automated tool such as *autopin* is recommended.

7.5 Xeon Phi

The Xeon Phi is based on Intel's many integrated core architecture. As such, it is not fully comparable to general purpose processors. Instead, it is designed as an extension card, which is installed in addition to the host system's processors to serve as an accelerator as appropriate.

Initially porting an application to run natively on the Xeon Phi is straightforward, as it is also based on the x86 architecture and most of the commonly used libraries are available. Fully exploiting the computing capabilities of this architecture on the other hand is more difficult, as several hardware traits differ significantly from those of general purpose processors.

To be able to fit such a large number of physical cores into a single processor package, the individual cores were deliberately kept simple and trimmed down and are missing an automatic hardware prefetcher or branch prediction, for instance. This dramatically increases the number of cache misses and pipeline stalls produced by unoptimized applications, resulting in decreased performance.

Even though the clock frequency of the Xeon Phi is only about two or three times lower than of the Westmere-EX respectively Westmere-EP processors, sequential runtime of the contingency analysis application (Table 6.1) as well as the expression tree evaluation using dynamically generated code (Table 4.4) is more than ten times slower.

This architectural handicap can be compensated by utilizing the 4-way simultaneous multiprocessing provided by the Xeon Phi's cores. When the execution stream of one logical core has to be halted to wait for a data item from memory, the execution streams of the three other logical cores may use the shared physical resources, improving overall utilization of the processors resources.

Due to this reason the applications of both test cases evaluated on the Xeon Phi partly continue to scale when using more than 60 threads – the number of physical cores on a Xeon Phi. The general purpose processors on the other hand often automatically avoid the need to halt an execution stream, thereby reducing the possibility to execute multiple streams without slowing each other down.

Although the architectural compatibility of the Xeon Phi allows to compile almost every application for execution on it with no or very limited changes to the source code, established industrial applications are rarely structured to allow efficient massive parallelism which would be necessary to exploit a Xeon Phi to its full potential. Moreover, even if the application efficiently supports a high number of threads the default memory consumption often is too high for the currently available memory on the accelerator card.

To sum up, the immediate use of a Xeon Phi to process significant parts of industrial simulation software, speeding it up in the process, is often not practical without considerable changes and optimizations. As illustrated in Section 4.6.5, however, it can be used to efficiently execute specific, suitable parts of an application.

All in all, Xeon Phi's theoretical computing capabilities are impressing and should be utilized where possible and worthwhile. Also, even if not used in production, it can be used to adapt established software for increased parallel efficiency. The current trend in development of general purpose processors is pointing in a similar direction, so the experience gained by tuning an application for the Xeon Phi will also help in improving the performance on general purpose processors.

Generally, other accelerator architectures exist as well. One notable example are graphics cards and their utilization is commonly called *General Purpose Computation on Graphics Processing Units (GPGPU)*. However, as these architectures differ significantly from the commonly used x86 architecture, it is difficult to cost-effectively port generic industrial simulation software to them and achieve good performance. Therefore, these architectures are not covered in this thesis.

Chapter 8

Synopsis and Outlook

8.1 Synopsis

The possibilities to improve performance of existing applications are almost endless and often, especially within corporate environment, it is necessary to balance the time and effort that is invested into performance optimization against the achieved performance improvement.

The thesis presents techniques to improve performance of typical established applications by using a cost-effective step by step approach. By systematically locating performance bottlenecks and performing localized optimizations of methods and data structures, the overall performance of an application can be improved significantly without the need to rewrite considerable parts of it.

In some situations, the characteristics of the underlying algorithms or methods cannot be changed or compensated by only conducting localized and confined changes. Instead, more fundamental modifications are necessary, often coming at the price of additional complexity and turning out to be cumbersome with respect to other properties such as parallel efficiency.

Also covered are possibilities to tune the performance of an application by adapting runtime parameters such as the number of threads used or the distribution of these threads among the physical resources.

Last but not least, speeding up an application by using a (Xeon Phi) accelerator is also considered. While the Xeon Phi is perfectly capable of doing so, the effort required to properly use its potential is rather high. This especially applies for applications with a high demand for memory, as the amount available on the accelerator is very limited.

8.2 Outlook

Current trends in processor research and development clearly point at further increasing number of cores and larger SIMD units. Intel's many integrated core architecture, the basis for the Xeon Phi accelerator, provides a glimpse into that future, albeit currently with slimmed down cores.

It may well be that both architectures, the general purpose and the many integrated core one, reunite sometime in the future. In the meantime Xeon Phi is both a powerful accelerator as well as an opportunity to study and improve the behavior of massively parallel applications.

In the long term, software from all disciplines, be it academic or industry, will need to adapt to the increasing number of processor cores and the further growing disparity between computational capability and memory latency. The techniques devised in this thesis illustrated how this can be put into practice.

Also, the code generation implementation presented here can be used as a generic and powerful tool to efficiently evaluate expression trees on both general purpose processors as well as the many integrated core architecture. By adapting the generated code to simultaneously handle multiple sets of variables using SIMD units, efficiency and utilization of modern processor hardware can be increased even further.

Bibliography

[1] *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors*. Advanced Micro Devices, Inc., January 2013. 42301 Rev 3.14.

[2] D. Amann, Andreas Blaszczyk, Günther Of, and Olaf Steinbach. Simulation of floating potentials in industrial applications by boundary element methods. Berichte aus dem Institut für Numerische Mathematik 2013/3, Technische Universität Graz, 2013.

[3] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967. doi: 10.1145/1465482.1465560.

[4] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857077.

[5] Paul Bachmann. *Die Analytische Zahlentheorie*. B. G. Teubner, 1894. Zahlentheorie, Band 2. in German.

[6] Prasanta Kumar Banerjee. *The Boundary Element Methods in Engineering*. McGraw-Hill College, 2nd rev. edition, 1994. ISBN 978-0-07-707769-3.

[7] Mario Bebendorf. Approximation of boundary element matrices. *Numerische Mathematik*, 86(4):565–589, 2000. ISSN 0029-599X. doi: 10.1007/PL00005410.

[8] Mario Bebendorf. *Effiziente numerische Lösung von Randintegralgleichungen unter Verwendung von Niedrigrang-Matrizen*. PhD thesis, Universität Saarbrücken, 2000. dissertation.de, Verlag im Internet, 2001. ISBN 978-3-89825-183-9. in German.

[9] Mario Bebendorf. *Hierarchical Matrices*, volume 63 of *Lecture Notes in Computational Science and Engineering*. Springer, 2008. ISBN 978-3-540-77146-3. doi: 10.1007/978-3-540-77147-0.

[10] Mario Bebendorf. Another software library on hierarchical matrices for elliptic differential equations (AHMED), 2013. URL `http://bebendorf.ins.uni-bonn.de/AHMED.html`.

[11] Mario Bebendorf and Sergej Rjasanow. Adaptive low-rank approximation of collocation matrices. *Computing*, 70(1):1–24, 2003. ISSN 0010-485X. doi: 10.1007/s00607-002-1469-6.

[12] Vasile Berinde. *Iterative Approximation of Fixed Points*, volume 1912 of *Lecture Notes in Mathematics*. Springer, 2nd rev. and enlarged edition, 2007. ISBN 978-3-540-72233-5.

[13] Andreas Blaszczyk and Carsten Trinitis. Experience with PVM in an industrial environment. In Arndt Bode, Jack Dongarra, Thomas Ludwig, and Vaidy Sunderam, editors, *Parallel Virtual Machine — EuroPVM '96*, volume 1156 of *Lecture Notes in Computer Science*, pages 174–179. Springer, 1996. ISBN 978-3-540-61779-2. doi: 10.1007/3540617795_22.

[14] Andreas Blaszczyk, Reto Flückiger, Thomas Müller, and Carl-Olof Olsson. Convergence behavior of coupled pressure and thermal networks. *COMPEL: The International Journal for Computation and Mathematics in Electrical and Electronic Engineering*. ISSN 0332-1649. accepted for publication.

[15] Andreas Blaszczyk, Zoran Andjelic, P. Levin, and A. Ustundag. Parallel computation of electric fields in a heterogeneous workstation cluster. In Bob Hertzberger and Giuseppe Serazzi, editors, *High-Performance Computing and Networking*, volume 919 of *Lecture Notes in Computer Science*, pages 606–611. Springer, 1995. ISBN 978-3-540-59393-5. doi: 10.1007/BFb0046688.

[16] Andreas Blaszczyk, Harsh Karandikar, and Giovanni Palli. Net value! Low-cost, high-performance computing via the intranet. ABB Review 1/2002 (M708-200201), ABB, 2002. pages 35–42.

[17] Arthur W. Burks, Don W. Warren, and Jesse B. Wright. An analysis of a logical machine using parenthesis-free notation. *Mathematical Tables and Other Aids to Computation*, 8(46):53–57, 1954. ISSN 0891-6837. doi: 10.2307/2001990.

[18] David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997. ISBN 978-0-2016-3392-4.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, 3rd edition, 2009. ISBN 978-0-262-53305-8.

[20] Peter J. Denning. The locality principle. *Communications of the ACM*, 48(7):19–24, July 2005. ISSN 0001-0782. doi: 10.1145/1070838.1070856.

[21] Peter Deuflhard. *Newton Methods for Nonlinear Problems: Affine Invariance and Adaptive Algorithms*, volume 35 of *Springer Series in Computational Mathematics*. Springer, 2011. ISBN 978-3-642-23898-7. doi: 10.1007/978-3-642-23899-4.

[22] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994. doi: 10.1137/S0097539791194094.

[23] Michael Eberl, Wolfgang Karl, Carsten Trinitis, and Andreas Blaszczyk. Parallel computing on PC clusters — an alternative to supercomputers for industrial applications. In Jack Dongarra, Emilio Luque, and Tomàs Margalef, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1697 of *Lecture Notes in Computer Science*, pages 493–498. Springer, 1999. ISBN 978-3-540-66549-6. doi: 10.1007/3-540-48158-3_61.

[24] Joseph Edwards. *An elementary treatise on the differential calculus: with applications and numerous examples*. Macmillan, 1892.

[25] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54 (12):1901–1909, December 1966. ISSN 0018-9219. doi: 10.1109/PROC.1966.5273.

[26] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, September 1972. ISSN 0018-9340. doi: 10.1109/TC.1972.5009071.

[27] Michael Franz. *Code Generation On the Fly: A Key to Portable Software*. PhD thesis, ETH Zürich, 1994.

[28] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, July 1984. ISSN 0004-5411. doi: 10.1145/828.1884.

[29] Matteo Frigo. A fast fourier transform compiler. *ACM SIGPLAN Notices*, 34(5): 169–180, May 1999. ISSN 0362-1340. doi: 10.1145/301631.301661.

[30] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. ISSN 1388-3690. doi: 10.1023/A:1010095604496.

[31] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, September 2004.

[32] Christoph Gramsch, Andreas Blaszczyk, Helmut Löbl, and Steffen Großmann. Thermal network method in the design of power equipment. In Gabriela Ciuprina and Daniel Ioan, editors, *Scientific Computing in Electrical Engineering*, volume 11 of *Mathematics in Industry*, pages 213–219. Springer, 2007. ISBN 978-3-540-71979-3. doi: 10.1007/978-3-540-71980-9_22.

[33] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987. ISSN 0021-9991. doi: 10.1016/0021-9991(87)90140-9.

[34] David J. Griffiths. *Introduction to Electrodynamics*. Addison Wesley, 3rd edition, 1999. ISBN 978-0-13-805326-0.

[35] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996. ISSN 0167-8191. doi: 10.1016/0167-8191(96)00024-5.

[36] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5): 532–533, May 1988. ISSN 0001-0782. doi: 10.1145/42411.42415.

[37] Wolfgang Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. part i: Introduction to $\mathcal{H}$-matrices. *Computing*, 62(2):89–108, 1999. ISSN 0010-485X. doi: 10.1007/s006070050015.

[38] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Elsevier, 5th edition, 2012. ISBN 978-0-123-83872-8.

[39] Ralf Hiptmair and Lars Kielhorn. BETL – a generic boundary element template library. Research Report 2012-36, SAM - Seminar for Applied Mathematics; ETH Zürich, November 2012.

[40] *Intel® Xeon Phi™ Coprocessor Instruction Set Architecture Reference Manual*. Intel Corporation, September 2012. Reference Number: 327364-001.

[41] *Intel® Xeon Phi™ Coprocessor System Software Developers Guide*. Intel Corporation, June 2013. SKU# 328207-002EN.

[42] *Intel®64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, July 2013. Order Number: 248966-028.

[43] *Intel®64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture*. Intel Corporation, September 2013. Order Number: 253665-048US.

[44] *Intel®64 and IA-32 Architectures Software Developer's Manual – Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*. Intel Corporation, September 2013. Order Number: 325383-048US.

[45] *Intel®64 and IA-32 Architectures Software Developer's Manual – Volume 3 (3A, 3B & 3C): System Programming Guide*. Intel Corporation, September 2013. Order Number: 325384-048US.

[46] ISO/IEC/IEEE 9945:2009. *Information technology – Portable Operating System Interface (POSIX®) Base Specifications, Issue 7*. International Organization for Standardization, September 2009.

[47] ISO/IEC 14882:2011. *Information technology – Programming languages – C++*. International Organization for Standardization, September 2011.

[48] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH Computer Architecture News*, 38(3):60–71, June 2010. ISSN 0163-5964. doi: 10.1145/1816038.1815971.

[49] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, June 1993. ISBN 978-0-13-020249-9. With chapters by L.O. Andersen and T. Mogensen.

[50] Robert M. Keller. Look-ahead processors. *ACM Computing Surveys (CSUR)*, 7(4): 177–195, December 1975. ISSN 0360-0300. doi: 10.1145/356654.356657.

[51] Tobias Klug, Michael Ott, Josef Weidendorfer, and Carsten Trinitis. autopin – automated optimization of thread-to-core pinning on multicore systems. In Per Stenström, editor, *Transactions on High-Performance Embedded Architectures and Compilers III*, volume 6590 of *Lecture Notes in Computer Science*, pages 219–235. Springer, 2011. ISBN 978-3-642-19447-4. doi: 10.1007/978-3-642-19448-1_12.

[52] Petr Kobalicek. AsmJit – Complete x86/x64 JIT Assembler for C++ Language., 2013. URL `http://code.google.com/p/asmjit/`.

[53] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the java hotspot™ client compiler for java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):7:1–7:32, May 2008. ISSN 1544-3566. doi: 10.1145/1369396.1370017.

[54] Christian Lage. *Softwareentwicklung zur Randelementmethode: Analyse und Entwurf effizienter Techniken*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 1995. in German.

[55] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*, volume 2. B. G. Teubner, 1909. in German.

[56] LLVM Team; University of Illinois at Urbana-Champaign. The LLVM compiler infrastructure, 2013. URL `http://www.llvm.org`.

[57] Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*, October 2013. URL `http://x86-64.org/documentation/abi.pdf`. ABI Draft 0.99.6.

[58] James Clerk Maxwell. *A treatise on electricity and magnetism*. Clarendon Press, 1873.

[59] *x64 Software Conventions*. Microsoft Corporation. URL `http://msdn.microsoft.com/en-us/library/7kcdt6fy.aspx`. Microsoft Developer Network (MSDN).

[60] Gordon Earle Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[61] Gordon Earle Moore. Progress in digital integrated electronics. In *Electron Devices Meeting, 1975 International*, volume 21, pages 11–13, 1975.

[62] Thomas Müller, Carsten Trinitis, and Jasmin Smajic. Cache efficiency and scalability on multi-core architectures. In Victor Malyshkin, editor, *Parallel Computing Technologies*, volume 6873 of *Lecture Notes in Computer Science*, pages 88–97. Springer, 2011. ISBN 978-3-642-23177-3. doi: 10.1007/978-3-642-23178-0_8.

[63] Thomas Müller, Josef Weidendorfer, and Andreas Blaszczyk. Expression tree evaluation by dynamic code generation - are accelerators up for the task? In *Parallel Processing (ICPP), 2013 42nd International Conference on*, pages 230–239, October 2013. doi: 10.1109/ICPP.2013.32.

[64] Laurence W. Nagel and Donald Oscar Pederson. SPICE (Simulation Program with Integrated Circuit Emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, April 1973.

[65] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.1*, July 2011.

[66] J.M. Ortega and Werner C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, 1970. ISBN 978-0-89871-461-6.

[67] Darrel W. Pepper and Juan C. Heinrich. *The Finite Element Method: Basic Concepts and Applications*. Series in Computational and Physical Processes in Mechanics and Thermal Sciences. Taylor & Francis, 2nd edition, 2006. ISBN 978-1-59169-027-6.

[68] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(5):895–913, September 1999. ISSN 0164-0925. doi: 10.1145/330249.330250.

[69] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. Adaptive insertion policies for high performance caching. *ACM SIGARCH Computer Architecture News*, 35(2):381–391, June 2007. ISSN 0163-5964. doi: 10.1145/1273440.1250709.

[70] Youcef Saad and Martin H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986. doi: 10.1137/0907058.

[71] Klaus Samelson and Friedrich L. Bauer. Sequential formula translation. *Communications of the ACM*, 3(2):76–83, February 1960. ISSN 0001-0782. doi: 10.1145/366959.366968.

[72] Gregor Schmidlin. *Fast solution algorithms for integral equations in* $\mathbb{R}^3$. PhD thesis, ETH Zürich, 2003. Nr. 15016.

[73] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press Cambridge, 1995. ISBN 978-0-262-69215-1.

[74] Guy Lewis Steele, Jr. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, lambda: The ultimate goto. In *Proceedings of the 1977 Annual Conference*, ACM '77, pages 153–162. ACM, 1977. ISBN 978-1-4503-2308-6. doi: 10.1145/800179.810196.

[75] Thomas L. Sterling, Daniel Savarese, Donald J. Becker, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages 11–14. CRC Press, 1995.

[76] John E. Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in Science and Engineering*, 12(3):66–73, 2010. ISSN 1521-9615. doi: 10.1109/MCSE.2010.69.

[77] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[78] Carsten Trinitis, Tilman Küstner, Josef Weidendorfer, and Jasmin Smajic. Sparse matrix operations on multi-core architectures. In Victor Malyshkin, editor, *Parallel Computing Technologies*, volume 5698 of *Lecture Notes in Computer Science*, pages 41–48. Springer, 2009. ISBN 978-3-642-03274-5. doi: 10.1007/978-3-642-03275-2_5.

[79] Carsten Trinitis, Tilman Küstner, Josef Weidendorfer, and Jasmin Smajic. Sparse matrix operations on several multi-core architectures. *The Journal of Supercomputing*, 57(2):132–140, 2011. ISSN 0920-8542. doi: 10.1007/s11227-010-0428-9.

[80] Union for the Co-ordination of Transmission of Electricity (UCTE). Final Report – System Disturbance on 4 November 2006, January 2007. URL `https://www.entsoe.eu/fileadmin/user_upload/_library/publications/ce/otherreports/Final-Report-20070130.pdf`.

[81] Josef Weidendorfer, Tilman Küstner, and Sally A. McKee. Performance optimization by dynamic code transformation. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 7:1–7:2. ACM, 2011. ISBN 978-1-4503-0698-0. doi: 10.1145/2016604.2016614.

[82] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2): 3–35, 2001. ISSN 0167-8191. doi: 10.1016/S0167-8191(00)00087-9. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000.

[83] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 38:1–38:12. ACM, 2007. ISBN 978-1-59593-764-3. doi: 10.1145/1362622.1362674.

[84] Allen J. Wood and Bruce F. Wollenberg. *Power Generation, Operation, and Control*. John Wiley & Sons, 2nd edition, 1996. ISBN 978-0-471-58699-9.

[85] L. C. Wrobel and M. H. Aliabadi. *The Boundary Element Method*. Wiley-Blackwell, 2002. ISBN 978-0-470-84139-6.

[86] William Allan Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995. ISSN 0163-5964. doi: 10.1145/216585.216588.

[87] Tjalling J. Ypma. Historical development of the newton-raphson method. *SIAM Review*, 37(4):531–551, 1995. doi: 10.1137/1037125.

[88] Olek C. Zienkiewicz, Robert L. Taylor, and J. Z. Zhu. *Finite Element Method - Its Basis and Fundamentals*. Elsevier Butterworth-Heinemann, 6th edition, 2005. ISBN 978-0-7506-6320-5.