TECHNISCHE
UNIVERSITÄT
WIEN

BACHELORARBEIT

Parallelization Strategies for Particle Monte Carlo Simulations

ausgeführt am Institut für Mikroelektronik
der Technischen Universität Wien

unter der Anleitung von
Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Erasmus Langer
Dipl.-Ing. Dr.techn. Josef Weinbub, BSc
Dr.techn. Paul Ellinghaus, MSc, B.Eng.

durch

Matthias Franz Glanz

MatNr. 1325620

November 21, 2016

Zusammenfassung

In den letzten Jahren sind Mehrzweck-Grafikberechnungs Beschleuniger (auf Englisch "General Purpose Graphics Processing Units" - GPGPU) und Coprozessoren auf Basis von vielen integrierten Prozessor Kernen (auf Englisch "Many Integrated Cores" - MIC) rapide entwickelt und verbreitet worden. Diese modernen Vielkern-Plattformen bringen ein enormes Potenzial an Parallelisierung mit sich, welches wiederum die Möglichkeit bietet, numerische Simulation um ein Vielfaches zu beschleunigen. Diese Arbeit gibt einen Überblick, wie diese neuen Ressourcen für Partikel basierte Monte Carlo Simulationen, wie den Open-Source Ensemble Wigner Monte Carlo Simulator in ViennaWD, verwendet werden können. Ein großer Teil der wissenschaftlichen Simulatoren verwenden heute das sogenannte Message Passing Interface (MPI) um die Rechenaufgaben auf eine große Anzahl von Berechnungs-Knoten zu verteilen. Aufgrund dessen soll in dieser Arbeit ein Überblick gegeben werden, wie Grafikkarten und Coprozessoren in den einzelnen Berechnungs-Knoten verwendet werden können, insbesondere auch innerhalb einer MPI Umgebung, welches das Potenzial für weitere Simulations-Beschleunigung erhöht. Die Parallelisierungssprachen OpenMP und CUDA werden verglichen, konkret im Aufwand, der benötigt wird um existierenden Code zu parallelisieren und in der Flexibilität verschiedene Plattformen zu unterstützten. Am Beispiel einer vereinfachten, aber doch aussagekräftigen, Monte Carlo Simulation auf MPI Basis wird der Vergleich präsentiert und es wird untersucht wie die Implementierung auf den unterschiedlichen parallelen Plattformen unterstützt wird. Die Ergebnisse zeigen klar, dass die Verwendung von hybrider Parallelisierung ein wichtiger und mit angemessenem Aufwand erreichbarer Schritt ist, um die modernen Plattformen von morgen zu unterstützen.

Abstract

In recent years General Purpose Graphics Processing Units (GPGPUs) and Co-processors based on Many Integrated Cores (MIC) were rapidly developed and widely distributed. These modern many-core computing platforms hold enormous potential for parallelism, which offers the opportunity to speed-up numerical simulations significantly. This thesis gives an overview how particle based Monte Carlo simulations can utilize these new resources for their typical large computational workloads, such as the free open-source ensemble Wigner Monte Carlo simulator shipped with ViennaWD. Usually, scientific simulators in general use the so-called Message Passing Interface (MPI) to distribute computational tasks to a number of distributed compute nodes. Therefore, this thesis additionally compares different approaches to utilize many-core processors on single compute nodes in combination with using the MPI for hybrid parallelization, to further increase the potential simulation speed-up. The parallel programming languages OpenMP and CUDA are compared with respect to programming effort to parallelize existing code and flexibility to support different platforms. To that end, a simplified yet representative example problem of a Monte Carlo simulation, which is distributed over MPI, is presented and different hybrid parallelization approaches are discussed. The results clearly show that utilizing hybrid parallelization techniques is an important and reasonably achievable effort for particle Monte Carlo simulators to efficiently utilize today's and tomorrow's computing platforms.

Acronyms

ALU:	Arithmetic Logic Unit
API:	Application Programming Interface
AVX:	Advanced Vector Extensions
CPU:	Central Processing Unit
CUDA:	Compute Unified Device Architecture
FLOPS:	Floating Point Operations Per Second
GDDR:	Graphics Double Data Rate
GPC:	Graphics Processing Cluster
GPGPU:	General Purpose Graphics Processing Units
GPU:	Graphics Processing Unit
HBM:	High Bandwidth Memory
HPC:	High Performance Computing
MIC:	Many Integrated Cores
MPI:	Message Passing Interface
NUMA:	Non-Unified Memory Access
OpenMP:	Open Multi-Processing
PC:	Personal Computer
PCIe:	Peripheral Component Interconnect Express
RAM:	Random-Access Memory
SIMD:	Single Instruction Multiple Data

Contents

1 Introduction

Since the 1970s the increasing use of computer simulations in physics has started a disruption of the classical division of physics into theoretical and experimental physics. Computer simulations can be seen as a third branch next to the two classical ones [1]. Analytical problems with many degrees of freedom can often not be solved without a number of approximations. Information gathered by experiments sometimes does not lead to conclusive answers because some conditions of the experimental sample are not exactly known or unknown impurity effects occur [2]. The fundamental advantage of computer simulations is that it is easy to re-run a new simulation-based experiment with adapted parameters. Repeating regular laboratory experiments, often requires a defined surrounding controlled via, for instance, clean rooms - which are extremely expensive to operate and investigations might take a long time and man-power.

One of the major computer simulation techniques are so-called Monte Carlo simulations, which are widely used in statistical physics [1]. Monte Carlo methods are used in a variety of scientific fields from financial modelling, population biology, computer vision to interacting particle approximations. Particle based Monte Carlo simulations are used to, for instance, calculate electron, neutron and photon transport often within a certain geometrical configuration of cells [3]. These cells are used to have a discrete location grid on which the particles can move. Interacting mechanisms between particles and between particle and cell, e.g., scattering, absorption, local emission, and annihilation must be simulated on the cells.

Deterministic methods solve problems for the average particle behaviour. However, the Monte Carlo method simulates millions of particles and their individual history through the material [3]. By summation of particles at a given point and normalization the probability density of particles is calculated. The probability distribution resulting from a stimulation step is statistically sampled to describe the total phenomenon [4]. Probability distributions are randomly sampled using transport data to determine the quantities of interest for each simulation step.

This thesis gives an introduction into programming techniques necessary to perform Monte Carlo Simulations on today's available hardware and analyses the effort involved. The range of available computing resources reaches from standalone Personal Computers (PCs) to cluster systems with thousands of processors [5]. In recent years, GPGPUs and MIC coprocessor cards extended the variety of available computing platforms [6].

In Chapter 2, the available hardware is introduced and compared. Chapter 3 focuses on the different programming approaches to parallelize Monte Carlo algorithms. An example simulation problem, based on ViennaWD's ensemble Wigner Monte Carlo simulator [7], is presented in Chapter 4. The required changes to run this example on various hardware platforms and with different parallelization strategies are discussed. The conclusion presented in Chapter 5 compares the necessary changes which acts as a basis for estimating the required effort to utilize the analysed computing platforms.

2 Overview of Computing Platforms

Today, the number of transistors per chip still keeps doubling every 18 months as Moore's Law predicted [8]. In 2004 multi-core scaling began to change the processor landscape. As single-core processors reached their physical limits of chip level power and thermal implications the chip clock rate nearly saturated and could only be marginally increased [9]. The race for clock rate increases came to an end and other ways to improve processing power had to be found. Multi-core architectures enabled chip manufacturers to further increase the number of transistors on their processors. New technologies as dynamic voltage and frequency scaling were introduced to reduce the chip temperature. While the improvement of micro architectures has led to minor increases in processing power, the increasing number of cores was the reason for major performance increases [10]. The number of cores per die started rising with every new processor generation. Traditional parameters as per transistor speed or microcode efficiency rose far slower than the number of cores. Modern Central Processing Units (CPUs) with 18 cores are on the market now and the number of cores will keep increasing.

Graphics Processing Units (GPUs) have been originally designed to accelerate graphics related to professional graphics-focused applications as well as computer games [11]. With increasing realism the number of polygons per second processed on the cards kept rising continuously. Graphics calculations are computationally very extensive and in the early 2000s research software engineers started to utilize the massive computing potential of GPUs for their simulation problems. The term GPGPU was shaped. The raw throughput of primarily floating point-heavy operations exceeded the throughput of CPUs drastically. Extensions to high level programming languages founded the basis of Nvidia's Compute Unified Device Architecture (CUDA) platform (Chapter 2.3) which is widely used in scientific simulations today. Aside from Nvidia's CUDA platform, OpenCL is an open source alternative to allow unified and high level access to multi- and many-core computing platforms, such as CPUs and GPUs, but is not further considered in this work. However, effort of developing in OpenCL and CUDA are comparable and thus the CUDA findings presented in this work apply to a large extent to OpenCL as well.

Coprocessors, such as Intel's Xeon Phi, are the third branch of modern parallel computing platforms next to general purpose CPUs and GPGPU accelerator cards. These devices put over 40 in-order execution cores on a single Peripheral Component Interconnect Express (PCIe) card [12]. The aim, similar to GPGPU, is to equip workstations or compute nodes with significant additional parallel computing capabilities. To utilize the potential performance, however, large amounts of parallel operations have to be executed.

Figure 2.1 shows the different hardware topologies to illustrate the number of data processing units on each type of device.The multi-core CPU contains a number of general purpose out-of-order execution cores and extensions as vector math units [13]. The coprocessor is based on simpler in-order execution cores with extended vector processing capabilities. The GPU consists of a huge number of streaming multiprocessors, which are placed in groups on different hierarchy levels. As the number of cores in CPUs keeps rising with every generation the segregation of multi-core CPUs and coprocessors might end in the near future.

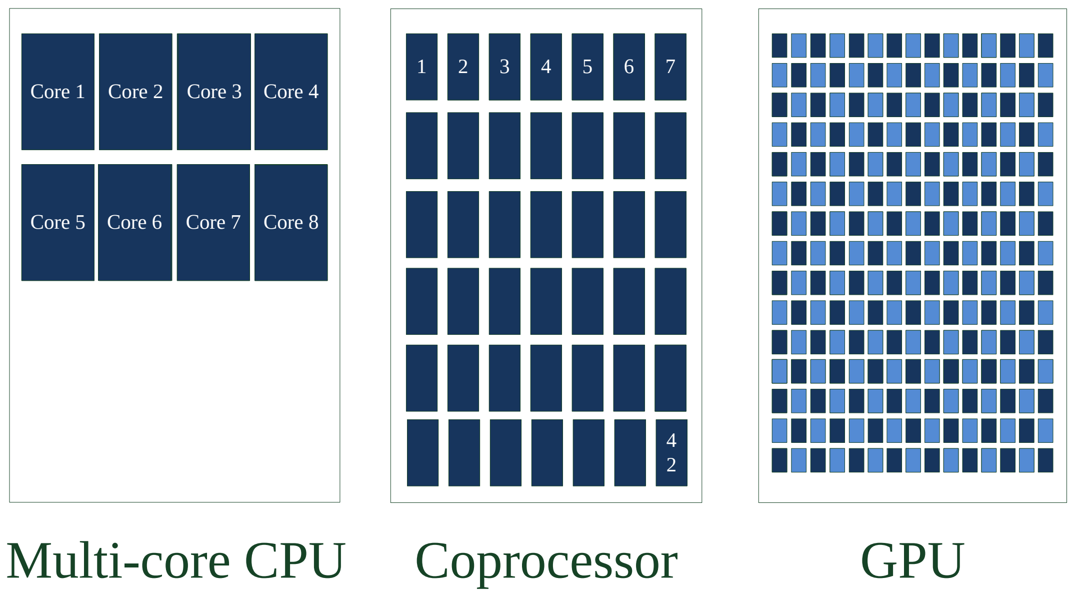

Figure 2.1: Compute core granularity in different compute resources [14]

Many scientific high performance cluster systems use GPGPU cards or coprocessors to boost the performance of the overall system [15]. Table 2.1 shows the number of systems in the Top 500 super computers list using coprocessors or GPGPU accelerators. 20,8% of the systems use hybrid parallelization. The Tianhe-2 system at the Chinese National Super Computer Center in Guangzhou is the world's fastest system in November 2015 and uses Intel Xeon Phi Knights Corners coprocessors [16]. The number 2 system Titan at DOE/SC/Oak Ridge National Laboratory in the United States uses Nvidia Kepler accelerators. These numbers show that hybrid systems are very popular with supercomputing centres, especially if economic costs / Floating Point Operations Per Second (FLOPS) and power consumption / FLOPS are the primary goals.

Table 2.1: Accelerators in TOP500 List from November 2015 [16]

Coprocessor / Accelerator Architecture	Number of Installations
Nvidia Kepler	52
Nvidia Fermi	14
Xeon Phi	29
ATI Radeon	3
Nvidia Kepler & Xeon Phi	4
PEZY_SC	2

2.1 Multi-Core Processors

The shrinking of transistor size from 130nm (e.g. Intel Pentium 4) to 14nm (e.g. Intel Skylake) shows the progress in chip manufacturing in recent years [17]. Despite all predictions the manufacturers were able to optimize their processes continuously till the current 14nm technology [18]. Physical limitations are increasingly manifesting themselves and for the first time Intel had to change its fast-pacing tick-tock scheme of micro architecture improvements and fabrication process improvements because the next step from 14nm to 10nm is delayed. After years of constant progress the shrinking process is slowing down.

Intel has continuously increased its dominance in the server CPU market in recent years with a peak market share of 99% in 2015 [19]. AMD is not offering competitive products to Intel Xeon E5 and Xeon E7 processors, however, AMD has scheduled a new product line in 2017 with the ZEN architecture.

Contrary to Intel's x86 CPU architecture, IBM's Power architecture is used in 23 clusters in the TOP500 supercomputing list (November 2015) [16]. The Power8 architecture was first presented in 2013 and allows up to 96 threads per socket [20]. IBM has launched the OpenPOWER initiative in 2013 with partners like Nvidia and Google to promote the use of Power processors in combination with Nvidia Tesla accelerators in High Performance Computing (HPC) environments to create an alternative to Intel based systems.

In recent years, clusters using ARM processors have been built to demonstrate the potential of modern mobile computing cores [21]. Power consumption is becoming a limiting resource in HPC. Mobile computing cores used in smart phones and tablet computers have drastically increased their computational power in recent years while being optimized for low power demand. These cores can be used for higher packaging density and lower cost per processor core chips. ARM added fully pipelined double precision floating point units in the Cortex A15 family to enable fast computation of complex problems. Furthermore, the NEON Single Instruction Multiple Data (SIMD) extension was designed to process multiple numerical operations in parallel to make ARM systems appealing to customers out of the mobile chip sector. While there might be potential for ARM based HPC clusters, they are mostly being used in storage appliances and database systems which are very I/O depending and do not require very high peak computing performance [22]. However, starting from this niche ARM based severs could gain market share in HPC in the next years.

Characteristics all these multi-core architectures have in common:

- Out-of-order execution of processor instructions to optimize the usage of available hardware resources on the chip
- Branch prediction to process branches while efficiently using pipelines
- Multiple Arithmetic Logical Units (ALUs) for floating point and integer arithmetic
- Different pipelines for different instructions

Figure 2.2 shows Intel's Silvermont Core architecture. This out-of-order execution core represents the lower end of Intel's current product range [23]. It has all the characteristics mentioned above and is used in tablet computers and low end laptops.

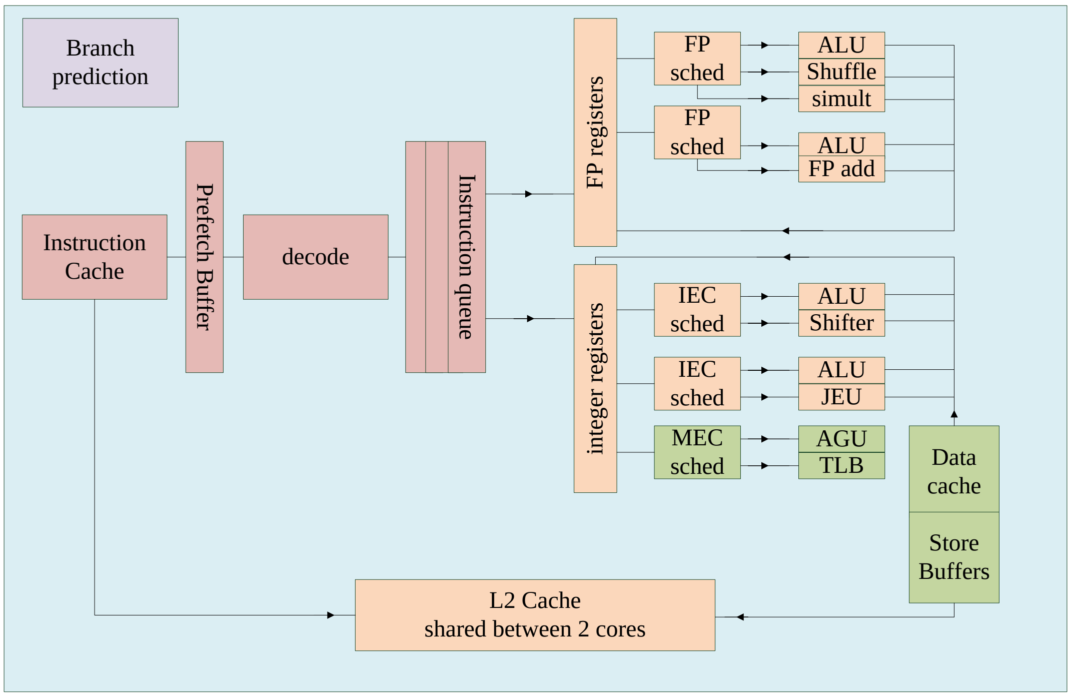

Figure 2.2: Core Block Diagram Intel Silvermont [23]

Current generations of high end multi-core processors have up to 22 cores [24]. Additionally, modern Xeon cores have additional functionality, like Intel Advanced Vector Extension (AVX) 2 which is especially of interest to accelerate SIMD problems. Therefore, these are very complex cores with out-of-order execution and scheduling on chip level.

2.2 Xeon Phi Many-Core Coprocessors

The first Xeon Phi (Knights Corner) coprocessors were introduced in 2012 as reaction to GPGPU accelerator cards by Nvidia and AMD [25]. Similar to GPGPU accelerators, the initial Xeon Phi coprocessor cards are connected to the host CPU via a PCIe interface and have their own separate memory on the card. The first generation of coprocessors cannot directly access the system's main memory and offered up to 64 cores.

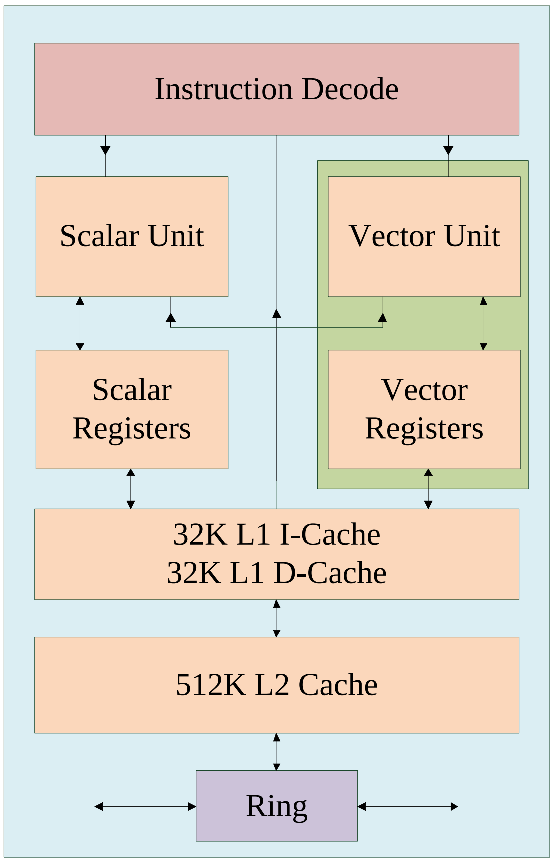

Figure 2.3: Block Diagram Intel Xeon Phi Core [12]

Figure 2.3 shows the block diagram of one single Xeon Phi Core. The core's complexity is reduced drastically compared to an standard CPU core, as shown in Figure 2.2. The main hardware characteristics of the Xeon Phi cores are [25]:

- Modified version of P54C design which was used in the first Pentium designs
- In-order execution
- 4-way simultaneous multi-threading per core
- 512 bit SIMD units
- 32 KB instruction cache
- Coherent Level 2 cache (512 KB per core)
- Ultra wide ring bus to connect memory and cores

2.3 Nvidia CUDA Accelerator Cards

Nvidia's Tesla architecture started changing the processing possibilities of GPUs drastically when it was introduced in November 2006 [26] [27]. The architecture unified vertex and pixel processors and extended the programmability. Previous GPUs consisted of graphics pipelines with separate stages. Vertex processors executed vertex shader programs and pixel fragment processors executed pixel shader programs. By unifying these elements into one functional unit non-graphics related parallel tasks are easier processed by the cards. Nvidia introduced the CUDA programming model and thus enabled utilizing the cards via an extended C programming language.

Nvidia's latest Tesla generation is the P100 GPU accelerator series with Nvidia's Pascal architecture [28]. These accelerators are optimized for HPC and deep learning applications. The Graphics Double Data Rate (GDDR) memory was replaced with second generation High Bandwidth Memory (HBM2). This memory is connected via Chip-on-Waver-on-Substrate to minimize the length of data paths. Furthermore, the accelerators are not only available with a PCIe interface but also with a NVLink interface which is promised to speed up the data link between two accelerators for up to five times. Calculations with half precision (16 bit floating point numbers) are now possible. This is expected to further boost applications like deep learning applications where throughput is more important than precision. A GP100 accelerator consists of an array of Graphics Processing Clusters (GPCs), Texture Processing Clusters (TPCs), and Streaming Multiprocessors (SMs) [29].

The hardware characteristics of one GP100 accelerator are:

- 6 GPCs per accelerator
- 10 SMs per GPC
- 64 single precision CUDA cores and 4 TPCs per SM
- Total 3850 single precision CUDA cores per accelerator
- Total 240 texture units per accelerator
- 4096 KB Level 2 Cache per accelerator

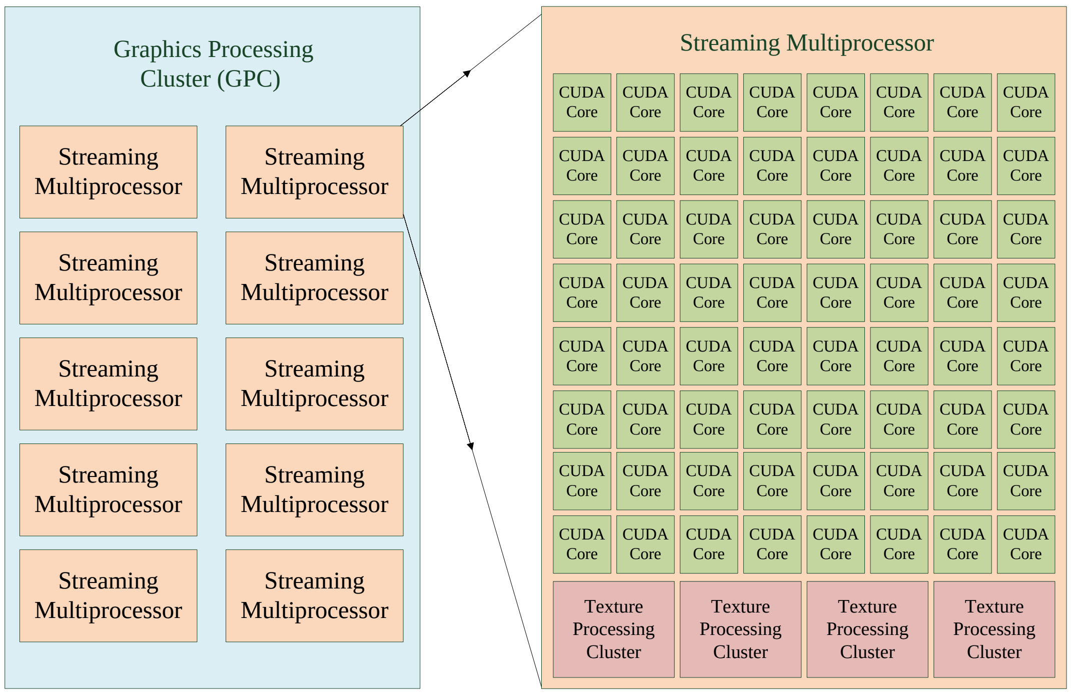

Figure 2.4: Block Diagram of a GPC [29]

Figure 2.4 shows the hierarchy of elements on the GP100 accelerator [29]. One GPC consists of ten SMs. Each of these consists of 64 CUDA cores and 4 TPCs.

3 Parallel Programming Approaches

Soon after scientists started using computer simulations to verify theories the simulation problems started to exceed the memory and/or computing capabilities of a single workstation. The next step was to use multiple workstations to distribute the workload. ARCNET was the first commercially available cluster solution launched in 1977 [30]. Approaches to run software on different computing nodes had to be found. Distribution of software, synchronisation between nodes, and collection of results had to be defined. Different approaches were in use when the standardization of MPI started in 1992 [31]. The MPI enables to conveniently exchange data between compute nodes which work together to solve a computational problem. Due to the distributed-memory nature of the MPI, the developer is forced to carefully consider the structuring, decomposition, movement, and placement of data in the development phase to achieve efficient parallel implementations.

Multi-threading led to new concepts of parallelism on a single computer using a shared-memory approach, where all the threads of the thread-group can access the same memory address space (in contrast to MPI, where the MPI processes do not have access to each others memories). Utilizing multiple physical cores to work on a single task required new approaches designed for multi-core computers: POSIX threads (Pthreads) became a standard in 1995 whereas in 1997 the OpenMP standard was introduced [32]. OpenMP allows programmers to use parallel sections in software at a high level of abstraction, making it easier to develop parallel programs. OpenMP reduces the complexity of developing parallel programs drastically and is therefore widely used in modern simulations.

Hybrid parallelization typically uses MPI to realise communication between different compute nodes but on individual nodes the tasks are assigned to the available cores using, for instance, a shared-memory OpenMP approach or offloaded to a coprocessors or GPGPU accelerator (e.g. Nvidia Tesla accelerator) [33]. OpenMP offers a range of scheduling mechanisms with the aim to keep the utilization of the single cores as high as possible.

If a local compute node has a coprocessor or an accelerator card the computation tasks have to be realised using, for example, offload-OpenMP or CUDA programming [34]. Furthermore, coprocessors and accelerators have their own memory spaces. Therefore, data locality, reducing communication, and synchronisation is critical for achieving efficiently parallelized applications.

In Chapter 3.1, the basic concepts, topologies, and usage of MPI are presented. Chapter 3.2 gives on overview over shared memory multi-thread programming using OpenMP. In Chapter 3.2.2, the traditional multi-core OpenMP model is extended and approaches to utilize coprocessors using offload and hybrid OpenMP are presented. Finally, Chapter 3.3 focuses on the usage of GPGPU accelerator cards using the CUDA programming language.

3.1 MPI

The MPI was designed with the goal to unify syntax and precise semantics of message passing libraries [31]. The standardization began in 1992 and the initial version 1.0 of the standard was released in 1994. MPI is a message-passing Application Programming Interface (API). There are many implementations of the MPI standard of which several are free open-source projects and several commercial software packages.

The main reasons for using the MPI are [35]:

- Standardization: The MPI is the only message passing library that can be considered a standard that is supported on virtually all HPC platforms.
- Portability: When using a different platform that supports the MPI standard there is little to no need to modify source code.
- Performance Opportunities: Vendor implementations can use native hardware features to increase the MPI transmission speed.
- Functionality: There are over 430 MPI routines in MPI3 which cover nearly every use case. Most simple MPI programs can be written with about 10 MPI routines.

3.1.1 Structure of MPI Programs

The MPI uses objects called communicators and groups to define which collections of processes can communicate with each other. For larger and more complex problems it can be very useful to group and name processes to increase the readability of source code. Furthermore, every MPI process has a unique integer identifier called rank. The rank is an integer value within a communicator and can be used to assign certain tasks to specific MPI processes (e.g. `if(rank==0),{...}`). This is useful when, for instance, rank 0 is used as master process which reads data and sends computing tasks to the remaining processes following a master/slave approach. Then every process does a number of calculations processing a subset of the initial problem and ultimately sends the sub-results back to rank 0 for post-processing.

Figure 3.1 shows the structure of an MPI program. The essential MPI calls using the C programming language are discussed in the following [36].

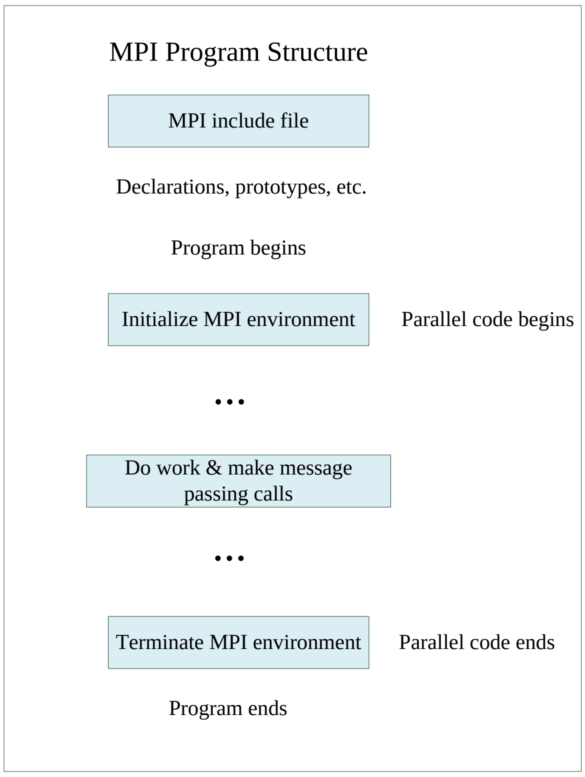
MPI Program Structure
MPI include file
Declarations, prototypes, etc.
Program begins
Initialize MPI environment
Parallel code begins
...
Do work & make message passing calls
...
Terminate MPI environment
Parallel code ends
Program ends

Figure 3.1: Structure of a MPI program [35]

Initializing the MPI environment:

- `MPI_Init( int *argc, char ***argv )`: Initializes the MPI environment. Must be called before any other MPI function.
- `MPI_Comm_size( MPI_Comm comm, int *size )`: Returns the number of processes in the specified communicator (e.g. `MPI_COMM_WORLD`)
- `MPI_Comm_rank( MPI_Comm comm, int *rank )`: Returns the rank (integer value) of the process within the specified communicator

After initializing the MPI environment the communication between different processes can be started. There are two types of communication in MPI: point-to-point message passing (e.g. `MPI_Send`) and collective (global) operations (e.g. `MPI_Bcast`). Point-to-point messages can be blocking or non-blocking. When using blocking calls the program waits until the data transfer is finished and proceeds afterwards. Non-blocking operations are sending and receiving data in the background while the process still executes other operations which introduces the potential for overlapping communication with computation, albeit in reality this is not always achieved [37]. Common point-to-point and collective operations:

- `MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`: Blocking send only returns after the data is stored in the send buffer and it is safe to write more data there.
- `MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`: Blocking receive returns after the data arrived and is ready-to-use by the process
- `MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)`: Non-blocking send returns almost immediately, i.e., the MPI does not wait for any calls or messages that confirm the data transmission. To ensure a correct data transfer *status* and *wait* mechanisms have to be used.
- `MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)`: Non-blocking receive returns almost immediately and starts receiving data in the background.
- `MPI_Wait(MPI_Request *request, MPI_Status *status)`: Waits for all non-blocking communications to be finished
- `MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )`: Sends a broadcast message from the process with rank number *root* to all other processes in the communicator.

- `MPI_Barrier( MPI_Comm comm)`: Synchronizes processes in the communicator. All processes are blocked until every process reaches the barrier.

After the MPI program finished the MPI environments needs to be appropriately shut down:

- `MPI_Finalize( void )`: Finalizes the MPI environment. No more MPI calls must be made after this command.

If errors or undefined conditions occur during the process execution the whole MPI program needs to be aborted.

- `MPI_Abort(MPI_Comm comm, int errorcode)`: Terminates all MPI processes in the specified communicator. A simple exit() call would only end one process while the rest of the processes keep running. To avoid this undefined state `MPI_Abort` is used.

The presented set of commands constitutes a subset of the provided API features, however, already with this small set it is possible to set up many MPI programs for various purposes. In general, communication between processes should be as limited as possible because it causes overhead and slows down the program execution. Often communication is responsible for limiting the parallel scalability and ultimately the execution performance.

3.1.2 MPI Implementations

The MPI standard describes the protocol and the semantics which have to be covered by a MPI implementation [38]. The behaviour of different MPI implementations should be as similar as possible to allow for portability. There are two widely used and free open-source MPI implementations at the moment: MPICH [39] and OpenMPI [40].

MPICH is a high quality open-source implementation of the latest MPI standard [41]. MPICH is used as base for a number of derivative implementations, e.g., Intel MPI, MVAPICH, Cray MPI, and IBM MPI. These implementations provide some specializations, for instance, the Intel MPI implementation provides optimized support for Xeon Phi coprocessors used within MPI environments. [42].

OpenMPI is an open-source MPI implementation based on the code of LAM/MPI, LA-MPI and FT-MPI [43]. It is widely used by the Top 500 supercomputers. One design goal was to support all widely used interconnects as TCP/IP, shared memory systems, Myrinet, Quadrics and InfiniBand. OpenMPI's process manager ORTE offers some advantages over the MPICH's Hydra process manager [44].

3.2 OpenMP

OpenMP was developed to be an industry standard API for shared memory programming [32]. Its first release for Fortran 1.0 was in 1997. Scalable applications need scalable hardware and software. Since the first multiprocessor architectures emerged the number of cores and the available memory increased with every new chip generation. This requires software to scale with the available hardware without software adjustments. Scalable hardware support and cache coherence are the basis of *scalable shared memory multiprocessor architectures* [37]. In these architectures every processor has direct memory access and can read and write every access of the memory. The first proprietary programming tools for such systems were not standardized and therefore not portable. One of the main OpenMP design goals was to bring software portability to scalable shared memory architectures without using message passing [45].

Message passing requires programmers to partition the simulation data explicitly [46]. On multiprocessor and multi-core systems with cache coherence data partitioning is not needed in that extent. Pthreads is too low level for many scientific simulations although it is a well established parallel execution model to provide task parallelism in low level software. Furthermore, applications must be designed with the aim of parallelism from the beginning to allow Pthreads to work properly. OpenMP allows incremental parallelism for existing software, in particular, it is tailored to accelerate compute-intensive loops which are predominant in scientific computing applications. [32].

More concretely, OpenMP is a set of compiler directives and callable runtime library routines. The directives extend the C, C++, and Fortran standard. Among the primary aims of OpenMP is to enable an easy access to shared memory parallel programming, which was very successful over the years as OpenMP is nowadays widely used in science and industry.

OpenMP provides directives to let the software developer indicate code regions to the complier which are intended to be executed in parallel [46]. In the way the instructions can be distributed among threads which will execute the code. OpenMP directives are instructions that are only understood by OpenMP supported compilers. For a regular compiler these directives look like comments and will be ignored thus also being backward compatible: Every OpenMP program can be built and run on any non-OpenMP-supporting platform, a property not supported by, for instance, MPI programs.

3.2.1 Structure of OpenMP Programs

The structure of an OpenMP program is called the fork-join model. When a parallel section starts, the master thread forks a number of slave threads and divides tasks among them. All threads run concurrently in the program process. However, unlike multiple processes running in parallel, all threads share the same memory space, heap, global variables, and shared memory. Therefore, resources and data used by multiple threads have to be locked to avoid race conditions and undefined program states.

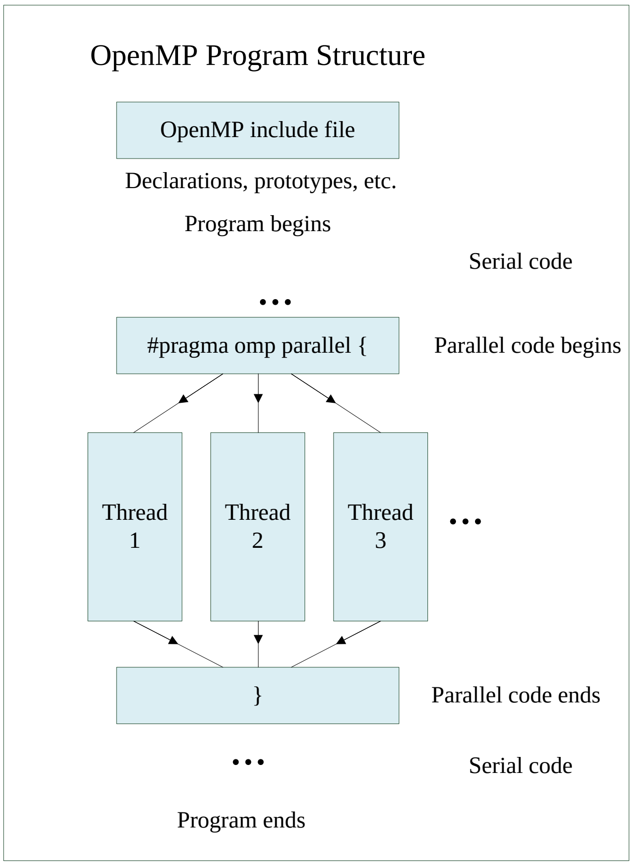

Figure 3.2: Fork-join structure of an OpenMP program [46]

Figure 3.2 shows the so-called fork-join structure of a typical OpenMP program. The most important directives used in the example program will be presented in the following.

All OpenMP directives used in a C program need to start with `#pragma omp`. A valid OpenMP instruction needs to appear after the pragma and before any clauses. For example:

- `#pragma omp parallel default(shared) private(beta,pi)`

The parallel region is the fundamental OpenMP construct. It is a block of instructions executed by multiple threads. When a parallel block starts one thread becomes the master thread with thread id 0. After the end of the parallel region only the master thread will be executed.

- `#pragma omp parallel { }`: Directive to start and end a parallel region.
- `omp_set_num_threads(int num_threads)`: Sets the number of threads to be started within the parallel region. The number can alternatively be defined using the `OMP_NUM_THREADS` environment variable.
- `int omp_get_num_threads(void)`: Returns the number of threads in a parallel region.
- `int omp_get_thread_num(void)`: Returns the thread id of the calling thread. Threads are numbered from 0 (master) to `omp_get_num_threads()`-1.

`For` loops are ideal blocks to be run in parallel. The big restriction of parallel for loops is that the program must not depend on the chronological order of the executed iterations.

- `#pragma omp for { }`: Directive to execute iterations of a for loop parallel
- `#pragma omp for schedule (static,dynamic) {}`: Allows to set the scheduling to different policies, such as, dynamic or static. Static scheduling divides the iteration space into (near)equal parts while dynamic scheduling distributes the iteration space into smaller chunks, gradually feeding those to threads once a thread is finished with processing the previous one. Static scheduling has less overhead but is only suitable to problems where the computational effort for each iteration remains nearly constant. On the contrary, dynamic scheduling allows to handle varying computational effort per iteration better and is thus typically used for load-unbalanced situations.

3.2.2 OpenMP on Coprocessors

OpenMP is the primary programming model to utilize the parallelization capabilities of Intel Xeon Phi coprocessors [12]. The programs can either be compiled to run solely on the coprocessor or a combination of CPU and coprocessor can be used. Figure 3.3 shows three different approaches of OpenMP on Intel Xeon Phi coprocessors.

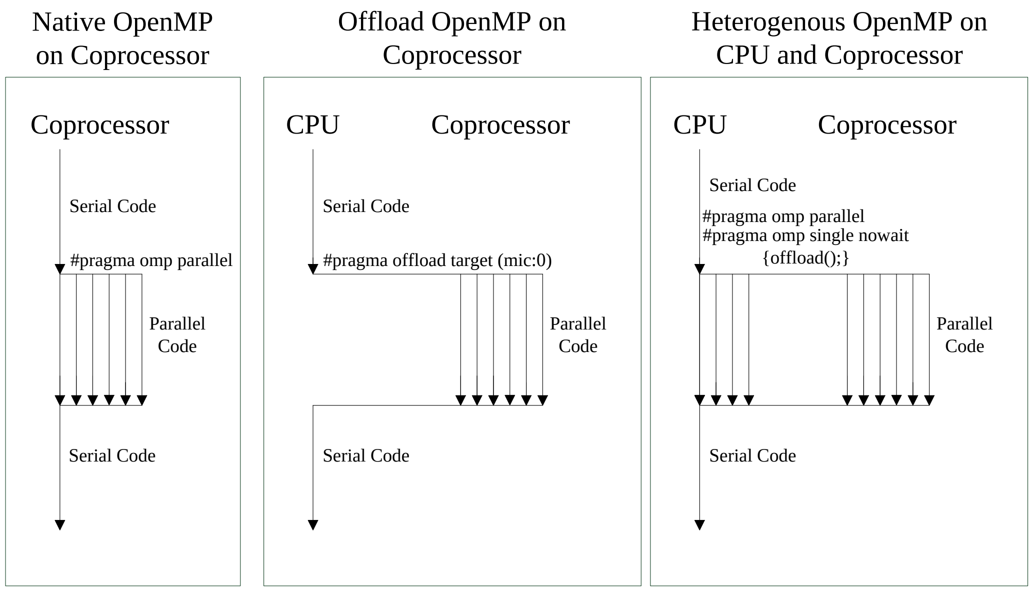

Figure 3.3: Types of OpenMP parallelism on Coprocessors [47]

Native OpenMP

Intel's MIC architecture is designed to natively support X86 code [48]. Thus, the coprocessor can be used to run X86 programs directly without changes of the source code. Intel designates Xeon Phi cards as coprocessors and not accelerators because they are very closely related to Intel's CPUs. Unlike GPUs, no vendor specific kernels or other device specific code needs to be added. It is good practice to first optimize OpenMP code on standard CPUs until the expected speed-up is measured [49]. In a second step, the program should be run on Xeon Phi coprocessors and be further optimized there. As the number of available cores can therefore quadruple further code optimizations are usually necessary.

Offload OpenMP

In offload mode, OpenMP uses the combination of a classical CPU with an coprocessor to accelerate applications [50]. The OpenMP directive `#pragma offload target (mic)` offloads a block of instructions onto a coprocessor, i.e., the parallel block is executed on the Xeon Phi. There common OpenMP directives can be used. Obviously, moving the data to and from the coprocessor creates overhead introduced by the latency and bandwidth limitations of the connecting bus. Therefore, compute intense program blocks should be processed by the coprocessor. When a coprocessor card is added to an existing hardware setup offload OpenMP is the easiest way to use the compute potential of the coprocessor.

Heterogeneous OpenMP

Heterogeneous OpenMP programs run calculations on different hardware at the same time [51]. When offloading is used, the main CPU is not needed for the most compute intense blocks of codes. But with modern CPUs with up to 20 cores the CPU also has significant parallel computing power. Therefore, the heterogeneous approach is to run a number of threads on the CPU while the rest of the threads runs on the coprocessor (cf. Figure 3.3, right). The parallel region is started on the CPU and with a `#pragma omp single nowait {offload();}` call the master thread offloads work to the coprocessor. When the threads exchange data in shared memory this can lead to significant overhead as the coprocessor has its own memory on the device. If heterogeneous OpenMP is used optimization and detailed performance analyses is necessary to avoid bottlenecks in the program execution. Although requiring significant effort to implement an efficient heterogeneous OpenMP program, it allows to utilize the entire computational processing power of a particular compute node, i.e., CPUs and coprocessors.

3.3 CUDA C

GPUs are massively parallel processors which support thousands of active threads (up to 20480 on GP100) [52]. This highly parallel computing platform requires a specialized programming model to efficiently express that kind of parallelism (most important data parallelism). Nvidia's CUDA represents such a tailored model. CUDA is a co-evolved hardware and software architecture which allows HPC developers to utilize the resources in a familiar programming environment [53]. The CUDA programming language extends the classical C language by several new instructions.

When scientists started to use GPUs for GPGPU computing they first used graphics programming APIs, such as OpenGL [53]. This approach limited the flexibility of the developed software and was an obstacle to HPC software developers who were often not familiar with graphics-focused programming. Nvidia's CUDA architecture enabled programming in C-like syntax and semantics, which was one of the primary reason for its success due to the broad C user base. Today, all current Nvidia GPUs can be accessed via CUDA. The Tesla product line is specifically developed for the HPC field and gained significant market share as accelerator for scientific simulations [54].

In real world applications where CUDA on accelerator cards is used speed-ups from 10x to 100x compared to conventional approaches are achieved [53]. Medical imaging was one of the first topics where performance increase of this magnitude changes the way diagnostics are used [55]. Today, GPGPU is of interest to all areas of compute-intensive science and engineering applications, such as mechanical-, chemical-, and electrical engineering as well as fluid dynamics, meteorology etc.

To clarify if program parts are executed on the compute node's CPU or the accelerator Nvidia introduced following definitions in its CUDA documentation [56]:

- Host: CPU of the compute node on which the accelerator is installed
- Device: accelerator
- Kernel: parallel parts of an application executed on the accelerator. Only one kernel is executed at a time. Many threads execute each kernel.

3.3.1 CUDA Execution Model

CUDA C allows the programmer to write straightforward C code that is run by thousands of parallel threads [53]. The C function which is executed on the GPU is called the kernel. The kernel is started usually on more than 10000 threads and the large number of threads (about ten times the number of CUDA cores) enables the SMs to schedule threads effectively to compensate the long latency of memory accesses and the shorter latency of ALU operations using pipelines. There are three refinements to running kernel functions across many parallel threads: hierarchical thread blocks, shared memory and barriers.

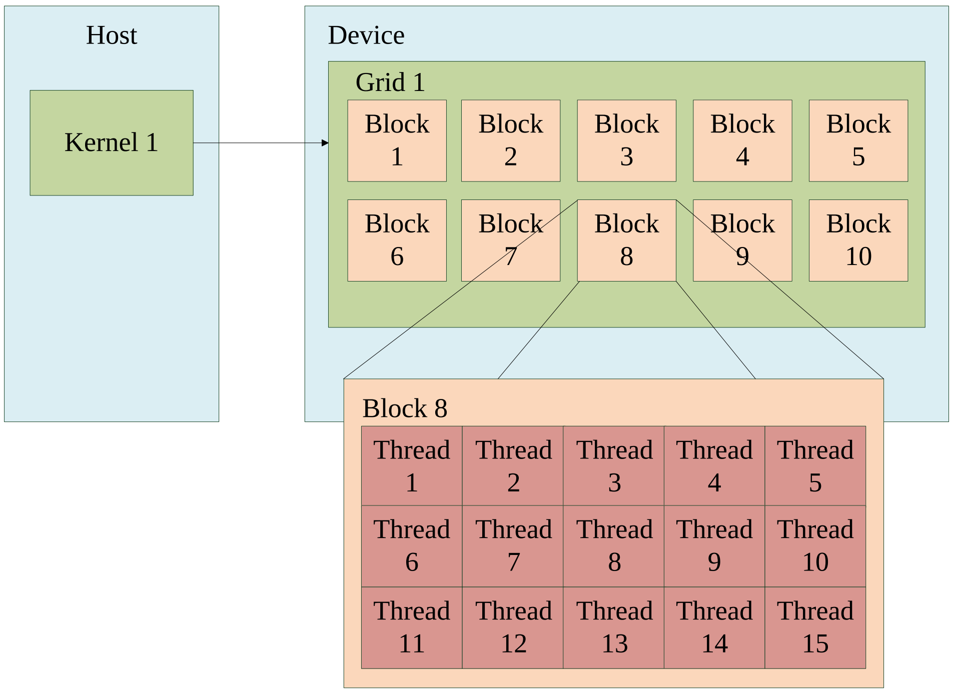

Figure 3.4: CUDA execution model [57]

Figure 3.4 shows the execution model of CUDA [58]. The kernel is called in the C program on the host CPU and then launched on the GPU. The array of threads is separated in thread blocks. One thread block can execute up to 512 threads (GP100 architecture, previous architectures 128 and 256 threads). Within a thread block cooperation and synchronisation of threads is possible. This is done via a shared memory mechanism of the thread block and barriers. A thread block is launched on a SM and does not migrate from there after launch.

The thread blocks are organized in a grid. A kernel is executed by a grid of thread blocks. The organisation of threads in thread blocks and grids allows the programmer to fine tune the parallelism to a specific problem. On low cost, low power GPUs one thread block with all synchronisation and shared memory can be run while on high end Tesla cards dozens of thread blocks are executed at the same time.

3.3.2 Structure of CUDA Programs

Programs using CUDA code that is run on accelerators are spilt in classic C functions and CUDA kernels [59]. The subroutines using CUDA C calls have to be compiled with Nvidia's CUDA compiler. The remainder of the code can be compiled with other compilers. Afterwards the compiled program parts (CUDA and non-CUDA) are linked together to form the actual application.

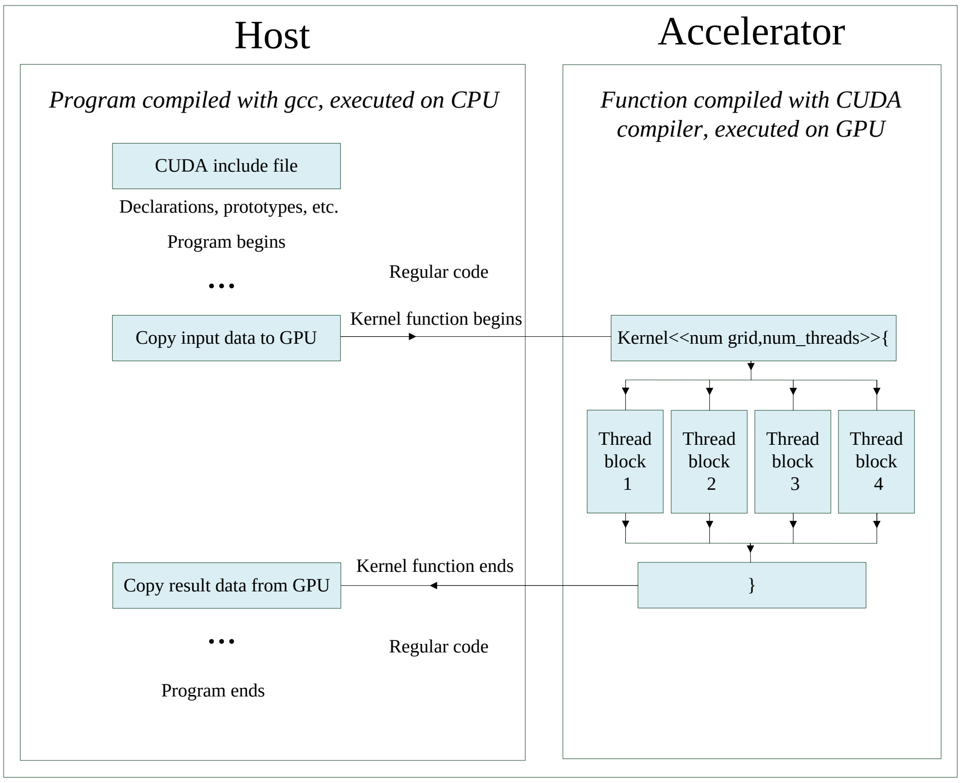

Figure 3.5: CUDA program structure [57]

Figure 3.5 schematically depicts the structure of a CUDA program. The execution starts on the host and before the parallel executed kernel can be started the input data has to be moved from the hosts' memory to the accelerator. This is achieved using `cudaMalloc()`, `cudaMemcopy()` and `cudaFree()` calls [59].

The programmer writes regular C code in the kernel for one sequential thread [60]. Parallelism is determined explicitly only by the dimensions of the thread blocks and the grid when launching the kernel. Thread creation, scheduling and thread termination are handled entirely by the underlying system. This is a more accessible approach to parallelism than using vector operations or other classical SIMD programming structures because vector operations are realized for single elements on thread level instead of a global scale.

4 Example Problem

In this chapter, a simplified implementation of a particle based simulation is presented. This code is used as a platform for comparing different parallelization approaches based on different languages and models. The individual peculiarities are identified as well as the parts of code, which have to be adopted or rewritten. The simulation is written in standard sequential C code. Afterwards different parallelization approaches, such as MPI and OpenMP are used to compare the required effort of parallelizing the code with the respective parallelization approach. Hybrid techniques with OpenMP offloading to coprocessors and offloading of the compute intense parts to GPUs is discussed at the end of the chapter.

The simulation is based on a two-dimensional finite grid within which particles are distributed. Each particle has an discrete spatial (i.e. x and y) coordinate and a non-discrete energy value. The simulation loop computes the new position and energy values of the particles at each iteration. This structure follows typical setups of particle Monte Carlo codes and, although simplified, allows to establish the basic work-flow with reduced complexity enabling to put the focus of the investigation on the different parallelization approaches. The code of the simulation loop is as follows.

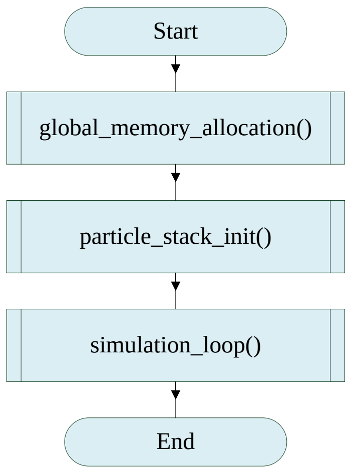

Figure 4.1: Structure of the example problem's serial implementation

```c
void simulation_loop(particle_t * stack, int size_x, int size_y,
    int num_particles, int num_steps) {

  int local_stacksize = num_particles;
  //Loop running number of steps of the simulation
  for(int iter = 0; iter < num_steps; iter ++)
  {
    double sum = 0;
    //Loop computing position and energy of every particle on grid
    for(int i=0; i < local_stacksize; i++)
    {
      stack[i].posx= (stack[i].posx + (rand() % size_x)) % size_x;

      stack[i].posy= (stack[i].posy + (rand() % size_y)) % size_y;

      stack[i].energy= sqrt( stack[i].posx + stack[i].posy );
      //Summing up energy of all particles on grid
      sum = sum + stack[i].energy;
    }
  } return;
}
```

For each iteration the simulation loop adds a random number to the x and y coordinates of the particles on the grid, to arbitrarily distribute the particles within the simulation domain. A modulo operation afterwards ensures that the particles stay within the grid. Afterwards the particle's *energy* is calculated from the x and y position on the grid. *Energy* in this example is no energy in the physical meaning, but a physically meaningless place-holder for further computations. Finally, the loop sums up the *energy* of all particles in the grid.

The `particle_t` datatype is a typedef structure which contains the parameters of one particle. There are the following three parameters:

- `int posx`: x position on the grid
- `int posy`: y position on the grid
- `double energy`: *energy* of the particle

The main program allocates memory for the particles. Afterwards, the function `particle_stack_init()` is called which assigns a random position to every particle on the grid and an random energy value. Then the simulation loop is started and the computation begins. Before shutting down in the end of the program the allocated memory is freed.

```c
int main(int argc, char* argv[])
{
  particle_t * global_particle_stack = NULL;

  //Global stack is generated with number of particles
  global_particle_stack = global_memory_allocation(num_particles);
  //Global stack is filled with random data
  particle_stack_init(global_particle_stack, num_particles);
  //Start simulation loop with number of particles and domain size
  simulation_loop(global_particle_stack, maxX, maxY, num_particles
    , num_iterations);

  free(global_particle_stack);

  return 0;
}
```

There are two main parameters in the main program which define the simulated environment and the simulation effort:

- `num_particles`: number of particles placed on the grid
- `num_iterations`: number of iterations of the simulation loop

4.1 MPI

In this section, the MPI is used to enable parallel execution of the example problem. The code of the simulation loop remains unchanged but the main program is adapted to initialize the MPI environment and to distribute the particles among the MPI processes.

After the initialization of the MPI environment the MPI rank and size is requested. The process with MPI rank 0 is declared the master process. The particles are sent to the slave processes from the master process. As the `particle_t` datatype is no standard datatype it has to be registered with the MPI environment.

```
int mpi_size, mpi_rank;
const int mpi_master = 0;

// initialize the MPI environment
MPI_Init(&argc, &argv);

// retrieve the number of MPI processes in the execution
// as well as the MPI rank (i.e. id) of the current process
MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);

//Register the particle struct with the MPI Backend
MPI_Datatype MPI_PARTICLE;
particle_t particle;
mpi_register_particle(&particle, &MPI_PARTICLE);
```

The master process allocates memory for the global particle stack holding all particles and initializes those. Afterwards, the global particle stack is split up into substacks. Figure 4.2 shows the spatial domain decomposition strategy used [61]. The following code snippet shows how the local particle stacks are filled with data from the global stack via the `stackSplit()` function. After splitting up the stack the data is sent to the slave processes via `MPI_Isend` call. The slave processes need to know how much memory they have to allocate for the incoming particle stack data. Therefore, the master sends out the local stack parameters via `MPI_Bcast` to all MPI processes.

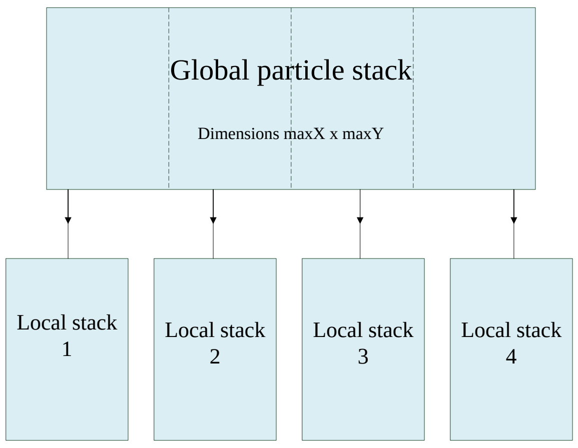

Figure 4.2: Spatial domain decomposition [62]

```
for(i = 0; i < mpi_size; i++) {
  //Allocate memory for the particle sub-stack of each sub-domain
  subStack[i] = calloc( substack_maxsize, sizeof(particle_t) );
  //Check which, and how many, particles in the global stack
   belong to this substack
  subStackSize[i] = stackSplit(global_particle_stack, subStack[i],
     startIndex_globalStack, stopIndex_globalStack,
   substack_maxsize);
}
//Send substacks to each of the helper processes:
MPI_Request sub_particles_send_requests[mpi_size-1];

for(int w = 1; w < mpi_size; w++) {
  MPI_Isend(subStack[w],subStackSize[w], MPI_PARTICLE, w,
   MPI_TAG_PARTICLESUBSTACK, MPI_COMM_WORLD,
  &sub_particles_send_requests[w-1]);
}

int parameter_array[SIZE_PARAMETER_ARRAY];
parameter_array[0] = maxX / mpi_size; //Size in X direction
parameter_array[1] = maxY; //Size in Y direction
parameter_array[2] = NUM_PARTICLES / mpi_size; //Size of substack
parameter_array[3] = NUM_ITERATIONS; //Number of iterations
//Transfer the parameters to the helper processes
MPI_Bcast( parameter_array, SIZE_PARAMETER_ARRAY, MPI_INT, 0,
   MPI_COMM_WORLD);
```

After sending the information to the slave processes the master process starts the simulation loop function and processes its own substack.

The slave processes, presented in the next code snippet, receive parameters defining their own share of work via `MPI_Bcast`. With this information the slave processes allocate memory for their local particle stacks. After `MPI_Irecv()` has received all particles the simulation loop is started. The simulation loop itself remains unchanged to the initial serial program.

```
int parameter_array[SIZE_PARAMETER_ARRAY];
//Recieve general information for all processes via broadcast
MPI_Bcast( parameter_array, SIZE_PARAMETER_ARRAY, MPI_INT, 0,
    MPI_COMM_WORLD);
//Create local particle stack
particle_t *local_particle_stack = calloc( parameter_array[2],
    sizeof(particle_t) );
//Recieve local particle stack from master process
MPI_Request receive_request;
MPI_Irecv(local_particle_stack, parameter_array[2], MPI_PARTICLE,
    mpi_master, MPI_TAG_PARTICLESUBSTACK, MPI_COMM_WORLD, &
    receive_request);
//Wait for MPI transmission to end
MPI_Wait(&receive_request, &status);

//
// Worker process starts its work
//
local_MAX_X = parameter_array[0]; //Size in X Direction
local_MAX_Y = parameter_array[1]; //Size in Y Direction
local_stacksize = parameter_array[2]; //Size of Substack
local_num_iterations = parameter_array[3]; //Number of Iterations
simulation_loop(local_particle_stack, local_MAX_X, local_MAX_Y,
    local_stacksize, local_num_iterations);
```

This simple example shows the effort a MPI implementation requires. The serial main function with 19 lines of code was replaced by the MPI main function with 187 lines of code. Additionally, the functions `stackSplit()` and `registerParticle()` added 59 more lines of code.

The big advantage of the MPI implementation is that, in principal, it supports MPI computing platforms of arbitrary sizes. When more processing power is needed the simulation can use more CPU cores if the cluster has available resources. As already indicated, the MPI requires the programmer to think about data locality from the beginning which, although requires initially a lot more effort as compared to, for instance, OpenMP, this initial effort usually pays off in the long run as proper data handling is one of the key ingredients for achieving high parallel efficiency on large core numbers.

4.2 OpenMP

The implementation of the OpenMP parallelization is only done in the simulation loop subroutine, as it hosts the central compute-intensive loop over the particle stack. The initial serial main program remains unchanged. On a side note, the MPI main program can also be used to use MPI for the distribution over multiple computing nodes and OpenMP for parallelism on the local shared memory node, paving the way for a hybrid MPI-OpenMP approach.

The following code snippet shows the OpenMP-parallelized implementation of the simulation loop. After the beginning of the parallel section (Line 3) the global particle stack is split into local particle stacks; one for each thread. This is done to ensure proper data placement, particularly important for Non-Uniform Memory Access (NUMA) systems: Each thread has its performance critical local substack within its memory domain, ensuring fast read and write access [37]. That being said, another effective but inefficient approach would be for all threads to *work* on the same master thread-owned global particle stack, albeit with expected reduced efficiency on NUMA systems: As all non-master-threads will have to access data in the master threads memory domain, overall performance will suffer from bandwidth and latency issues imposed by the memory links connecting the individual NUMA regions.

```
 1 void simulation_loop(particle_t * particle_stack, int size_x, int
      size_y, int num_particles, int num_steps) {
 2   #pragma omp parallel firstprivate (size_x, size_y, num_steps)
 3   { //Begin parallel section
 4     int local_stacksize = ceil (num_particles
 5                                 / omp_get_num_threads());
 6     int thread_num = omp_get_thread_num();
 7     printf("[simulation_loop]: Threadnumber: %d\n",thread_num);
 8     particle_t * local_stack = malloc(sizeof(particle_t)
 9                                       * local_stacksize);
10     memcpy(local_stack, &particle_stack[thread_num
11         *local_stacksize], local_stacksize * sizeof(particle_t));
12     for(int iter = 0; iter < num_steps; iter ++)
13     {
14       double sum = 0;
15       #pragma omp for schedule (static, local_stacksize)
16       for(int i=0; i < local_stacksize; i++)
17       {
18         local_stack[i].posx = (local_stack[i].posx +
19                               (rand() % size_x)) % size_x;
20         local_stack[i].posy = (local_stack[i].posy +
21                               (rand() % size_y)) % size_y;
```

```
        local_stack[i].energy = sqrt( local_stack[i].posx +
                                local_stack[i].posy );
        sum = sum + local_stack[i].energy;
    }
    }
  } //End parallel section
    return;
}
```

The OpenMP implementation adds 14 lines of code relative to the initial serial implementation. As already indicated, basic OpenMP functionality could also be achieved with just the two `#pragma` calls. But due to the already discussed NUMA-related data locality issues, it is usually useful to accept the extra overhead of replicating compute-critical data into local memory domains which is why this case is primarily considered in this investigation.

4.3 OpenMP on Xeon Phi

The OpenMP implementation running on a coprocessor is realized via OpenMP offloading routines. The normal OpenMP implementation from the previous section is extended to run on a coprocessor.

```
#pragma omp parallel firstprivate (size_x, size_y, num_steps)
{ //Begin parallel section
  int local_stacksize = ceil (num_particles
                                      / omp_get_num_threads());
  int thread_num = omp_get_thread_num();
  printf("[simulation_loop]: Threadnumber: %d\n",thread_num);
  particle_t * local_stack = malloc(sizeof(particle_t)
                                            * local_stacksize);
  memcpy(local_stack, &particle_stack[thread_num
        *local_stacksize], local_stacksize * sizeof(particle_t));
  #pragma offload target(mic) //Offload to coprocessor
    inout(local_stack: length(local_stacksize))
  for(int iter = 0; iter < num_steps; iter ++) {
    double sum = 0;
    #pragma omp for schedule (static, local_stacksize)
    for(int i=0; i < local_stacksize; i++) {
      local_stack[i].posx = (local_stack[i].posx + ...
      local_stack[i].posy = (local_stack[i].posy + ...
      local_stack[i].energy = sqrt( local_stack[i].posx + ...
      sum = sum + local_stack[i].energy;

```

The difference to the standard OpenMP implementation is the `offload` clause (Line 11-12). This call offloads the contained code to the coprocessor. The `inout()` part specifies which data is copied to and from the coprocessor. The `offload` call is placed there and not in the beginning of the parallel section for performance reasons. Splitting up the global particle stack and copying the data is typically faster on the CPU than on the coprocessor. The local particle stacks and their computation is then offloaded to the coprocessor.

This implementation can be called from the serial main routine or the MPI counterpart, again underlining the potential for an extension towards hybrid parallelization.

4.4 CUDA C

The CUDA C implementation of the simulation requires the serial simulation code to be exchanged by a function calling the CUDA kernel. The CUDA kernel is in a separate function run only on the GPU. The new simulation loop function allocates memory on the accelerator card and copies the data to the device memory.

```
__host__
void simulation_loop(particle_t * particle_stack, int size_x, int
    size_y, int num_particles, int num_steps) {
     int N = num_particles;
     #define THREADS_PER_BLOCK 128
     //Device copy of particle stack
     particle_t * dev_particle_stack;
     //Allocate device copie of particle stack
     cudaMalloc( (void **)& dev_particle_stack, num_particles *
    sizeof(particle_t) );

     //Copy data to device
     cudaMemcpy(dev_particle_stack, particle_stack, num_particles *
     sizeof(particle_t), cudaMemcpyHostToDevice);
     kernel<<< N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>> (
    dev_particle_stack, size_x, size_y , num_particles, num_steps);

     cudaMemcpy(particle_stack, dev_particle_stack, num_particles *
     sizeof(particle_t), cudaMemcpyDeviceToHost);

     cudaFree(dev_particle_stack);
     return;
}
```

The `__host__` statement defines that this function is only run on the host. This function and the kernel have to be compiled using the Nvidia CUDA compiler. The kernel call with `«Gridsize, Blocksize»` launches the kernel on the accelerator. The blocksize is fixed with 128 threads to allow the program to be executed on older CUDA accelerators, which had a maximum number of 128 threads. Modern accelerators increased this number to 256 or 512 threads per block. The gridsize is calculated from the number of particles in the simulation. The kernel is a new function and the `__global__` statement defines that it can be called from the host or the device.

```
__global__
void kernel( particle_t * particle_stack , int size_x , int size_y ,
    int num_particles , int num_steps)
{
  int iter;
  double sum;
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  //Set up random number generator curand. Calling function rand()
    from kernel is not easily possible and verly slowly
  curandState state;
  curand_init((unsigned long long)clock() + index , index , 0, &
   state);

  for(iter = 0; iter< num_steps; iter++)
  {
  double rand1 = curand_uniform_double(&state);
  double rand2 = curand_uniform_double(&state);
  particle_stack[index].posx = (particle_stack[index].posx + (int)
   rand1 % size_x)% size_x;
  particle_stack[index].posy = (particle_stack[index].posy + (int)
   rand2 % size_x)% size_x;
  particle_stack[index].energy = sqrt((float) ( particle_stack [
   index].posx + particle_stack [index].posy) );
    sum = sum + particle_stack[index].energy;
    __syncthreads();
  }
}
```

The standard `rand()` function is not available on the GPU, therefore, the cuRand random number generator is used to create random numbers.

The serial simulation loop implementation has 15 lines of code whereas the CUDA implementation requires 39 lines of code. To enable data processing on the accelerator data has to be copied to and from the GPU via CUDA `memcpy` calls.

The CUDA C implementation of the simulation loop can be called from the serial main or the MPI main. MPI can be used to distribute data to the computing nodes and the single nodes use GPU accelerators to compute the simulation results, again showing the support for hybrid parallelization. Exchanging data is more difficult because the simulation data is being processed on the GPU. Nvidia presented CUDA-aware MPI to address this problem and make synchronisation and data exchange easier on HPC clusters using GPU accelerators, however, for the sake of portability and clarity this is not further investigated in this work.

5 Conclusion and Outlook

This final chapter summarizes and analyses the results of the previous chapters and gives an outlook to the future of parallelization. In Chapter 5.1 the programming effort and portability of the different example problem implementations are discussed. Finally, Chapter 5.2 gives an overview over expected future developments in many-core programming.

5.1 Comparison of Programming Effort

The five implementations of the example problem in Chapter 4 are compared in programming effort. The chosen metric is *lines of code*, which is a straightforward way to describe the programming effort. Table 5.1 shows the lines of code needed for the different implementations.

Table 5.1: Lines of code in different implementations

	main routine	**simulation loop**	**further code**
Serial implementation	19	15	0
MPI implementation	187	15	59
OpenMP (with MPI)	19 (187)	29	0 (59)
OpenMP offload (with MPI)	19 (187)	31	0 (59)
CUDA (with MPI)	19 (187)	39	0 (59)

Evaluation of the different implementations and the gathered implementation effort:

- MPI: The MPI implementation requires the biggest development effort. If a HPC cluster has to be used for simulations there is basically no way around MPI to distribute data and exchange messages between the nodes. MPI can also be used on single multi-core compute nodes, making the MPI a versatile parallelization platform. However, depending on the degree of communication of an application, the speed-up achieved by adding further compute nodes will saturate at a certain point because of the communications overhead, which depends highly on the implementation.

- OpenMP: Parallelization of existing code with OpenMP requires comparatively less effort, however, significant optimization effort has to be considered to reach high parallel efficiency. OpenMP can be used to effectively and efficiently extend serial code step by step by parallel counterparts.
- Offload OpenMP: When OpenMP code is supposed to be offloaded to a coprocessor the OpenMP code should be first optimized on a regular CPU host, before moving tuning to a coprocessor system. The significant increase in the core numbers of a coprocessor will make further optimizations necessary to significantly outperform a HPC CPU-only host.
- CUDA C: Utilizing GPU accelerators requires the computations to be rewritten in CUDA C. GPUs have different functionalities than CPUs and therefore many routines cannot be used directly. The separate device memory requires the programmer to think about data locality and communication overheads.
- Hybrid implementations: As clusters with coprocessors and accelerators have already a large market share in the HPC market programmers have to become familiar with hybrid parallelization. Using MPI on a high level to communicate between compute nodes and OpenMP or CUDA C on the local node to utilize available coprocessors or accelerators is a viable option to utilize heterogeneous compute platforms..

MPI and OpenMP offer vendor-independent implementations. Both are available for all HPC clusters and stand-alone workstations. Offload OpenMP on coprocessors and CUDA C for Nvidia GPUs support only specific hardware. Therefore, implementations using those approaches are in a so-called vendor lock, meaning that the implementations cannot be used on other platforms outside the vendors hardware ecosystem. This fact might become problematic to research software developers as they might become dependent on the availability of a specific type of compute environment, especially as a lack of manpower and funding in research software projects renders repeated code porting to new platforms highly challenging and often simply not possible.

5.2 Future Developments and Outlook

In the near future, many-core applications will be more and more used to follow the hardware trend towards increasing core numbers. Medical imaging and other computationally challenging scientific simulations will rely more and more on these highly parallel platforms. However, to increase the usability new, more accessible language approaches, like OpenACC are being developed [63]. Using directives similar to OpenMP should make it easier for programmers new to the field of many-core programming to utilize accelerators and co-processors . On the contrary, OpenMP was extended to support offloading to coprocessors in specification 4.0 [64]. Researchers are working on approaches to run C code with offload OpenMP code on GPUs, to further the reach of OpenMP beyond CPUs and co-processors.

Hybrid parallelization has a big chance of becoming a standard for future implementations of scientific simulators. Today's high end multi core CPUs have more than 20 cores on a single chip (e.g. Intel Broadwell-EX). The gap to coprocessors is closing and there is no way around parallelization techniques to utilize this potential. This trend is expected to continue: The processor roadmaps predict rising core numbers and a transition from the multi core architecture to many core architectures [65]. For research software developers hybrid parallelization will have to become a standard method for utilizing all available, heterogeneous computing resources on single workstations or on clusters to support the ongoing quest for more accurate and faster computer simulations.

Bibliography

[1] K. Binder, D. Heermann, *Monte Carlo Simulation in Statistical Physics: An Introduction* (Springer Science & Business Media, 2010)

[2] A.B. Bortz, M.H. Kalos, J.L. Lebowitz, A New Algorithm for Monte Carlo Simulation of Ising Spin Systems, Journal of Computational Physics **17**(1), 10 (1975)

[3] J.F. Briesmeister, et al., MCNPTM - A General Monte Carlo N-Particle Transport Code, Version 4C, LA-13709-M, Los Alamos National Laboratory (2000)

[4] A. Smith, A. Doucet, N. de Freitas, N. Gordon, *Sequential Monte Carlo Methods in Practice* (Springer Science & Business Media, 2013)

[5] H. Gould, J. Tobochnik, W. Christian, *An Introduction to Computer Simulation Methods*, vol. 1 (Addison-Wesley New York, 1988)

[6] A. Heinecke, M. Klemm, H.J. Bungartz, From GPGPU to Many-Core: Nvidia Fermi and Intel Many Integrated Core Architecture, Computing in Science & Engineering **14**(2), 78 (2012)

[7] ViennaWD. "`http://viennawd.sourceforge.net`" (2016). [Online; accessed 5-October-2016]

[8] H. Esmaeilzadeh, E. Blem, R.S. Amant, K. Sankaralingam, D. Burger, Power Challenges May End the Multi-Core Era, Communications of the ACM **56**(2), 93 (2013)

[9] C. Isci, A. Buyuktosunoglu, C.Y. Cher, P. Bose, M. Martonosi, An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget, in *Proceedings of the 39th annual IEEE/ACM International Symposium on Microarchitecture* (IEEE Computer Society, 2006), pp. 347–358

[10] L. Chai, Q. Gao, D.K. Panda, Understanding the Impact of Multi-Core Architecture in Cluster Computing: A Case Study with Intel Dual-Core System, in *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)* (2007), pp. 471–478. DOI 10.1109/CCGRID.2007.119

[11] R. Vuduc, J. Choi, A Brief History and Introduction to GPGPU, in *Modern Accelerator Technologies for Geographic Information Science* (Springer, 2013), pp. 9–23

[12] J. Jeffers, J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming* (Newnes, 2013)

[13] B. Leback, D. Miles, M. Wolfe, Tesla vs. Xeon Phi vs. Radeon A Compiler Writer's Perspective, Technical News from The Portland Group (2013)

[14] J. Ghorpade, J. Parande, M. Kulkarni, A. Bawaskar, GPGPU Processing in CUDA Architecture, arXiv preprint arXiv:1202.4347 (2012)

[15] T. Rauber, G. Rünger, *Parallel Programming: For Multi-Core and Cluster Systems* (Springer Science & Business Media, 2013)

[16] Top500.org. Top 500 Super Computer List. "`https://www.top500.org/`" (2016). [Online; accessed 12-May-2016]

[17] J. Schutz, C. Webb, A Scalable X86 CPU Design for 90 nm Process, in *Solid-State Circuits Conference, 2004. Digest of Technical Papers. ISSCC. 2004 IEEE International* (IEEE, 2004), pp. 62–513

[18] A. Nalamalpu, N. Kurd, A. Deval, C. Mozak, J. Douglas, A. Khanna, F. Paillet, G. Schrom, B. Phelps, Broadwell: A Family of IA 14nm Processors, in *2015 Symposium on VLSI Circuits (VLSI Circuits)* (IEEE, 2015), pp. C314–C315

[19] Bloomberg. Intel Server Sales. "`http://www.bloomberg.com/news/articles/2015-07-15/intel-forecast-shows-server-demands-makes-up-for-pc-market-woes`" (2015). [Online; accessed 5-July-2016]

[20] B. Sinharoy, J. Van Norstrand, R.J. Eickemeyer, H.Q. Le, J. Leenstra, D.Q. Nguyen, B. Konigsburg, K. Ward, M. Brown, J.E. Moreira, et al., IBM POWER8 Processor Core Microarchitecture, IBM Journal of Research and Development **59**(1), 2 (2015)

[21] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, A. Ramirez, Tibidabo: Making the Case for an ARM-based HPC System, Future Generation Computer Systems (2013). DOI 10.1016/j.future.2013.07.013

[22] M.A. Laurenzano, A. Tiwari, A. Jundt, J. Peraza, W.A. Ward Jr, R. Campbell, L. Carrington, Characterizing the Performance-Energy Tradeoff of Small ARM Cores in HPC Computation, in *Euro-Par 2014 Parallel Processing* (Springer, 2014), pp. 124–137

[23] Intel. Intel Silvermont Architecture. "`https://software.intel.com/sites/default/files/managed/bb/2c/02_Intel_Silvermont_Microarchitecture.pdf`" (2015). [Online; accessed 27-April-2016]

[24] Intel. Intel Xeon E7 Servers. "`http://www.intel.com/content/www/us/en/processors/xeon/xeon-processor-e7-family.html`" (2016). [Online; accessed 16-July-2016]

[25] G. Chrysos, Intel Xeon Phi Coprocessor-The Architecture, Intel Whitepaper (2014)

[26] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: A Unified Graphics and Computing Architecture, IEEE micro **28**(2), 39 (2008)

[27] K. Rupp, J. Weinbub, F. Rudolf, Highly Productive Application Development with ViennaCL for Accelerators, in *AGU Fall Meeting Abstracts*, vol. 1 (2012), vol. 1, p. 1528

[28] Nvidia. Nvidia P100 Architecture. "`http://www.nvidia.com/object/tesla-p100.html`" (2016). [Online; accessed 22-July-2016]

[29] Nvidia. Nvidia GP100 Whitepaper. "`https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`" (2016). [Online; accessed 24-August-2016]

[30] P. Sharma, B. Kumar, P. Gupta, An Introduction to Cluster Computing using Mobile Nodes, in *2009 Second International Conference on Emerging Trends in Engineering & Technology* (IEEE, 2009), pp. 670–673

[31] W. Gropp, E. Lusk, N. Doss, A. Skjellum, A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, Parallel Computing **22**(6), 789 (1996). DOI 10.1016/0167-8191(96)00024-5

[32] L. Dagum, R. Menon, OpenMP: An Industry Standard API for Shared-Memory Programming, IEEE Computational Science and Engineering **5**(1), 46 (1998). DOI 10.1109/99.660313

[33] P. Rakić, D. Milašinović, Ž. Živanov, Z. Suvajdžin, M. Nikolić, M. Hajduković, MPI CUDA Parallelization of a Finite-Strip Program for Geometric Nonlinear Analysis: A Hybrid Approach, Advances in Engineering Software **42**(5), 273 (2011)

[34] V. Heuveline, M.J. Krause, J. Latt, Towards a Hybrid Parallelization of Lattice Boltzmann Methods, Computers & Mathematics with Applications **58**(5), 1071 (2009)

[35] R.L. Graham, *The MPI 2.2 Standard and the Emerging MPI 3 Standard* (Springer Berlin Heidelberg, Berlin, Heidelberg, 2009), pp. 2–2. DOI 10.1007/978-3-642-03770-2_2

[36] W. Gropp, E. Lusk, R. Thakur, *Using MPI-2: Advanced Features of the Message-Passing Interface* (MIT Press, Cambridge, MA, USA, 1999)

[37] G. Hager, G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*, 1st edn. (CRC Press, Inc., Boca Raton, FL, USA, 2010)

[38] J.J. Dongarra, S.W. Otto, M. Snir, D. Walker, An Introduction to the MPI Standard. Tech. rep., Knoxville, TN, USA (1995)

[39] High-Performance Portable MPI Implementation MPICH. `"http://www.mpich.org/"` (2016). [Online; accessed 12-September-2016]

[40] The OpenMP API Specification for Parallel Programming. `"http://openmp.org/wp/"` (2016). [Online; accessed 12-September-2016]

[41] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk, A. Skjellum, User's Guide to MPICH, a Portable Implementation of MPI, Argonne National Laboratory **9700**, 60439 (1995)

[42] M. Si, Y. Ishikawa, M. Tatagi, Direct MPI Library for Intel Xeon Phi Co-Processors, in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International* (2013), pp. 816–824. DOI 10.1109/IPDPSW.2013.179

[43] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, T.S. Woodall, *Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation* (Springer Berlin Heidelberg, Berlin, Heidelberg, 2004), pp. 97–104. DOI 10.1007/978-3-540-30218-6_19

[44] J. Hursey, J.M. Squyres, T.I. Mattox, A. Lumsdaine, The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI, in *2007 IEEE International Parallel and Distributed Processing Symposium* (2007), pp. 1–8. DOI 10.1109/IPDPS.2007.370605

[45] D. Molka, R. Schöne, D. Hackenberg, M.S. Müller, *Memory Performance and SPEC OpenMP Scalability on Quad-Socket x86_64 Systems* (Springer Berlin Heidelberg, Berlin, Heidelberg, 2011), pp. 170–181. DOI 10.1007/978-3-642-24650-0_15

[46] B. Chapman, G. Jost, R.v.d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)* (The MIT Press, 2007)

[47] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, *Parallel Programming in OpenMP* (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001)

[48] D. Schmidl, T. Cramer, S. Wienke, C. Terboven, M.S. Müller, *Assessing the Performance of OpenMP Programs on the Intel Xeon Phi* (Springer Berlin Heidelberg, Berlin, Heidelberg, 2013), pp. 547–558. DOI 10.1007/978-3-642-40047-6_56

[49] E. Saule, K. Kaya, Ü.V. Çatalyürek, *Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi* (Springer Berlin Heidelberg, Berlin, Heidelberg, 2014), pp. 559–570. DOI 10.1007/978-3-642-55224-3_52

[50] A. Duran, M. Klemm, The Intel Many Integrated Core Architecture, in *International Conference on High Performance Computing and Simulation (HPCS)* (2012), pp. 365–366. DOI 10.1109/HPCSim.2012.6266938

[51] E. Ayguadé, Badia, Extending OpenMP to Survive the Heterogeneous Multi-Core Era, International Journal of Parallel Programming **38**(5), 440 (2010). DOI 10.1007/s10766-010-0135-4

[52] D. Kirk, NVIDIA CUDA Software and GPU Parallel Computing Architecture, in *Proceedings of the 6th International Symposium on Memory Management* (ACM, New York, NY, USA, 2007), ISMM '07, pp. 103–104. DOI 10.1145/1296907.1296909

[53] D. Luebke, CUDA: Scalable Parallel Programming for High-Performance Scientific Computing, in *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro* (2008), pp. 836–838. DOI 10.1109/ISBI.2008.4541126

[54] M. Wang, H. Klie, M. Parashar, H. Sudan, *Solving Sparse Linear Systems on NVIDIA Tesla GPUs* (Springer Berlin Heidelberg, Berlin, Heidelberg, 2009), pp. 864–873. DOI 10.1007/978-3-642-01970-8_87

[55] L. Pan, L. Gu, J. Xu, Implementation of Medical Image Segmentation in CUDA, in *2008 International Conference on Information Technology and Applications in Biomedicine* (2008), pp. 82–85. DOI 10.1109/ITAB.2008.4570542

[56] C. Nvidia. Programming Guide (2008)

[57] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1st edn. (Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2013)

[58] J. Sanders, E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st edn. (Addison-Wesley Professional, 2010)

[59] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable Parallel Programming with CUDA, Queue **6**(2), 40 (2008). DOI 10.1145/1365490.1365500

[60] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, K. Skadron, A Performance Study of General-Purpose Applications on Graphics Processors using CUDA , Journal of Parallel and Distributed Computing **68**(10), 1370 (2008). DOI 10.1016/j.jpdc.2008.05.014

[61] P. Ellinghaus, J. Weinbub, M. Nedjalkov, S. Selberherr, I. Dimov, Distributed-Memory Parallelization of the Wigner Monte Carlo Method using Spatial Domain Decomposition, Journal of Computational Electronics **14**(1), 151 (2015). DOI 10.1007/s10825-014-0635-3

[62] J. Weinbub, P. Ellinghaus, M. Nedjalkov, Domain Decomposition Strategies for the Two-Dimensional Wigner Monte Carlo Method, Journal of Computational Electronics **14**(4), 922 (2015). DOI 10.1007/s10825-015-0730-0

[63] S. Wienke, P. Springer, C. Terboven, D. an Mey, *OpenACC — First Experiences with Real-World Applications* (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012), pp. 859–870. DOI 10.1007/978-3-642-32820-6_85

[64] C. Bertolli, S.F. Antao, A.E. Eichenberger, K. O'Brien, Z. Sura, A.C. Jacob, T. Chen, O. Sallenave, Coordinating GPU Threads for OpenMP 4.0 in LLVM, in *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC* (IEEE Press, Piscataway, NJ, USA, 2014), LLVM-HPC '14, pp. 12–21. DOI 10.1109/LLVM-HPC.2014.10

[65] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, P. Hanrahan, Larrabee: A Many-Core x86 Architecture for Visual Computing, ACM Trans. Graph. **27**(3), 18:1 (2008). DOI 10.1145/1360612.1360617

Hiermit erkläre ich, dass die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In– noch im Ausland in gleicher oder in ähnlicher Form in anderen Prüfungsverfahren vorgelegt.

Wien, 21.11.2016

....

Matthias Franz Glanz